CMSC 425
Game Programming[1]

David M. Mount
Department of Computer Science
University of Maryland
Spring 2017

Lecture 1: Introduction to Game Programming

Computer Game Programming: The famous game design Sid Meier once defined a *computer game* as "a series of interesting and meaningful choices made by the player in pursuit of a clear and compelling goal." A somewhat more concrete definition of a computer game, due to Mark Overmars, is "a software program in which one or more players make decisions through the control of game objects and resources, in pursuit of a goal." This course is concerned with the theory, practice, and technologies underlying the development of modern computer games of all varieties.

A Brief History: Today's computer games constitute a multi-billion dollar industry. There is an incredible variety of game genres: strategy, sports, shooter, role-playing, racing, adventure, and so on. Some games induce their players to dedicate long hours in pursuit of a distant goal, and others can be picked up and played for just a few minutes. Some games create astoundingly realistic and complex three-dimensional worlds and others involve little more than a simple 2-dimensional grid and very primitive computer graphics. Some games engage tens of thousands simultaneous users, and some involve a single user. How did we get here?

The Early Days: Computer games are as old as computers. One of the oldest examples was from 1958. It was a Pong-like game called *Tennis for Two*, which was developed by William Higinbotham of Brookhaven National Lab in 1958 and was played on an oscilloscope. Another example was a game developed in 1961 by a group of students at MIT, called *Spacewar*. It was programmed on the PDP 1. When the Arpanet (forerunner to the Internet) was developed, this program was available disseminated to a number of universities (where grad students like me would play them when their supervisors weren't watching). It is credited as the first influential computer game, but the influence was confined to academic and research institutions, not the general public.

Prior to the 1970s arcade games were mechanical, with the most popular being the many varieties of pinball. The computer game industry could be said to start with the first arcade computer game, called *Computer Space*. It was developed in 1971 by Nolan Bushnell and Ted Dabney, who founded Atari. One of Atari's most popular arcade games was *Pong*. There was a boom of 2-dimensional arcade games from the mid 1970s to the early 1980s, which was led by well-known games such as *Asteroids*, *Space Invaders*, *Galaxian*, and numerous variations In 1980, a popular game in Japan, called *Puck Man*, was purchased by Bally for US distribution. They recognized the enticement for vandalism by changing "Puck" into another well known 4-letter word, so they changed the name to "Pac-Man." They game became extremely successful.

The 70's and 80's: During the 1970s, computer games came into people's homes with the development of the Atari game console. Its popularity is remarkable, given that the early technology of the day supported nothing more sophisticated than Pong. One of the popular features of later game consoles, like the Atari 2600, is that it was possible to purchase additional cartridges, which looked like 8-track tapes, allowing users to upload new games.

The game industry expanded rapidly throughout the 1970s and early 1980s, but took an abrupt downturn in 1983. The industry came roaring back. One reason was the popularity of a game developed by Shiguero Miyamoto for Nintendo, called *Donkey Kong*, which featured a cute Italian "everyman" character, named *Mario*, who would jump over various obstacles to eventually save his lady from an evil kidnapping gorilla. Mario went on to become an icon of the computer game industry, and Donkey Kong generated many spin-offs involving Mario, notably *Super Mario Bros.* Eventually Donkey Kong was licensed to the Coleco company for release in their home game console and Super Mario Bros. was one of the top sellers on the Nintendo Entertainment System (NES).

Consoles, Handhelds, and MMOs: The 1990s saw the development of many game consoles with ever increasing processing and graphical capabilities. We mentioned the Nintendo NES above with Donkey Kong. Also, the early 1990s saw the release of the Sega Genesis and its flagship

game *Sonic the Hedgehog*. Another example is the Sony Playstation, with the very popular game *Final Fantasy* (version VII and on).

In the early 2000s came the rise of so called *sixth generation game consoles*, which include the very popular Sony Playstation 2 (and the hit game *Resident Evil*) and the Microsoft XBox and their seventh-generation follow-ups, the Playstation 3 and XBox 360 and Nintendo Wii, which dominate the market today.

A parallel thread through this has been the development of handheld game consoles. These include the Nintendo Game Boy, which was released in the late 1980s and early 1990s, the Nintendo DS in the mid 1990s, and the Playstation Portable in the mid 2000s.

Modern times have seen more games move to general purpose handheld devices, like the iPhone and Android. This has been accompanied by an infusion of games into social networking systems. With the advent of widely available game development systems, such as Unity and Unreal, there has been an explosion of independent, *indie*, games, and various systems for funding and distributing these projects.

Recent Trends: Game technology continues to develop in terms of the complexity and scale of games. Recently, with development of relatively inexpensive head-mounted displays, there has been a resurgence of interest in systems *virtual reality* and *augmented reality*. This technology increases the sense of immersion into the game's world. Improvements in sensor technology has led to more sophisticated controls based on voice recognition and gestures. The Microsoft Kinect and various devices for recognizing hand gestures are examples.

Another trend has been the increase opportunity for users to contribute to the games they play. For example, *open source* game software can be modified by users and shared. Games can also provide users the ability to modify and share artistic elements and design new game levels.

Elements of Computer Games: While computer games are fun to play, they can be terrifically challenging to implement. These challenges arise from the confluence of a number of elements that are critical to the execution and performance of the game. These include the following:

Real-time 3-dimensional computer graphics: Most high-end computer games involve the generation of photo-realistic imagery at the rate of 30–60 frames per second. This process is complicated by the following issues:

Large, complex geometric models: Large-scale models, such as factories, city-scapes, forests and jungles, and crowds of people, can involve vast numbers of geometric elements.

Complex geometry: Many natural objects (such as hair, fur, trees, plants, water, and clouds) have very sophisticate geometric structure, and they move and interact in complex manners.

Complex lighting: Many natural objects (such as human hair and skin, plants, and water) reflect light in complex and subtle ways.

Artificial intelligence: The game software controls the motions and behaviors of nonplayer entitites. Achieving realistic behavior involves an understanding of artificial intelligence.

Motion and Navigation: Nonplayer entities need to be able to plan their movement from one location to another. This can involve finding a shortest route from one location to another, moving in coordination with a number of other nonplayer entities, or generating natural-looking motion for a soccer player in a sports game.

Physics: The phyical objects of a game interact with one another in accordance with the laws of physics. Implicit in this is the capabiity to efficiently determine when objects collide with one another, and how they should move in response to these collisions.

Networking: Multiplayer online games use a network to communicate the current game state between players. Due to the latency inherent in network communication, the games states that individual players perceive are momentarily inconsistent. The game software must hide this latency and conceal these inconsistencies from the players.

Databases: The game state, especially for multiplayer online games, is maintained in a database. The database receives requests for updates of the game state from multiple sources, and it must maintain a consistent state throughout.

Security: Since the origin of games, there have been people who have sought ways of circumventing the games. This is particularly an issue in multiplayer games, where one player's cheating behavior degrades the experience for an honest player. For this reason, game software needs to be vigilant to detect and obstruct illicit game interaction.

The Scope of this Course: At some universities, game development constitutes a series of courses on various topics. Here, we will be able to focus on only a small part of the spectrum of relevant topics. While most game designers make use of sophisticate software tools (for graphics, modeling, AI, physics), it is not within the scope of this class to teach a particular set of tools (even though we will discuss game engines for the sake of project development). As in most upper-division computer science courses, our interest is not in how to *use* these tools, but rather how to *build* these systems. In particular, we will discuss the theory, practice, and technology that underlies the implementation of these systems.

This semester, we will touch upon only a subset of these issues. For each, we will discuss how concepts from computer science (and other areas such as mathematics and physics) can be applied to address the challenging elements that underlie game implementation.

Course Overview: In this course, we will provide an overview of what might be called the science and engineering of computer games. In particular, we will see how concepts developed in computer science can be applied to address the aforementioned elements of computer games. These include the following:

Game Engines: The organization, structure, and overall features of a typical game engine. Introduction to the Unity game engine.

Geometric Programming and Data Structures: Basic aspects of geometry and linear algebra and their applications to game programming. Bounding volumes and efficient collision detection.

Elements of Computer Graphics: Graphics systems and the graphics pipeline, model-view transformations and camera projection, lighting models, vertex and fragment shaders.

Modelling, and Animation: Shape representations and meshes, level of detail, terrain modeling, articulated models and skinning, animation, texture modeling, procedural generation and geometry synthesis.

AI and Algorithms for Games: Agent-based systems, decision making, finite-state machines, path planning, multiple-agent motion, flocking and emergent behavior.

Physics and Games: Newtonian dynamics, particle simulation, mass-spring models, collision detection and response, physics engines.

Networking and Games: TCP/IP, sockets programming, multiplayer gaming, latency hiding, distributed data consistency.

Security: Common methods of cheating in online games and approaches for detecting and counteracting them.

Lecture 2: Computer Game and Graphics System Architectures

Computer Game Architecture: A large computer game is a significant technical undertaking, involving a large number of interacting components. Of course, not all computer games require the same level of complexity. Different genres of games require different capabilities. The combination of components used for a simple casual 2-dimensional game is very different from a high-end AAA 3-dimensional game.

Computer Game Engine Architecture: One way to better understand the software structure underlying a generic game is to understand the structure of a typical game engines. Game engines arose in the mid-1990s. In particular, the software for the popular game *Doom* provided a separation between:

- core game components (such as the rendering system, collision detection system, audio system)
- art assets (models, textures, animations)
- rules of play

This separation made it easy for users to modify, or "modding", the game, and provided a framework for adding new elements. This model was extended to other games, including *Quake*, *Unreal*, and *Unreal Tournament* (all FPS games). At some point, these simple "modding systems" became generic enough that it was possible to implement a wide variety of very different games based on a common core set of components, the *game engine*. Examples of modern game engines include *Unity 3D* and *Unreal Engine 4*.

Game engines vary along a spectrum of ease of use and flexibility. Simple game engines can generate only a single type of game (e.g., 2-dimensional games), but are generally easy to pick up and use. Complex game engines can generate a great variety of games, but it can take quite a bit of time to master their various capabilities.

The following is a summary of the basic components of a modern game engine. We think of the engine as being designed in a number of layers, ranging from the lower levels (hardware and operating system) up to the higher levels (game specific entities like rules). Here is a summary of the levels, from low to high. These are illustrated in the figure below.

System: This includes low-level software for interacting with the operating system on which the game engine runs as well as the target system on which the game executes. Target systems can include general personal computers (running, say, Microsoft Windows, Linux, or Mac OS), game consoles (e.g., XBox, Playstation, Wii), or mobile devices (e.g., hand-held game consoles, tablets, and smart phones).

Third-Party SDKs and Middleware: These are libraries and sofware development toolkits (SDKs), usually provided from a third party. Examples include graphics (e.g., OpenGL and DirectX), physics (e.g., Havok, PhysX, and Bullet), basic algorithms and data structures (e.g., Java Class Library, C++ STL, Boost++), character animation (e.g., Granny), networking support (e.g., Unix sockets).

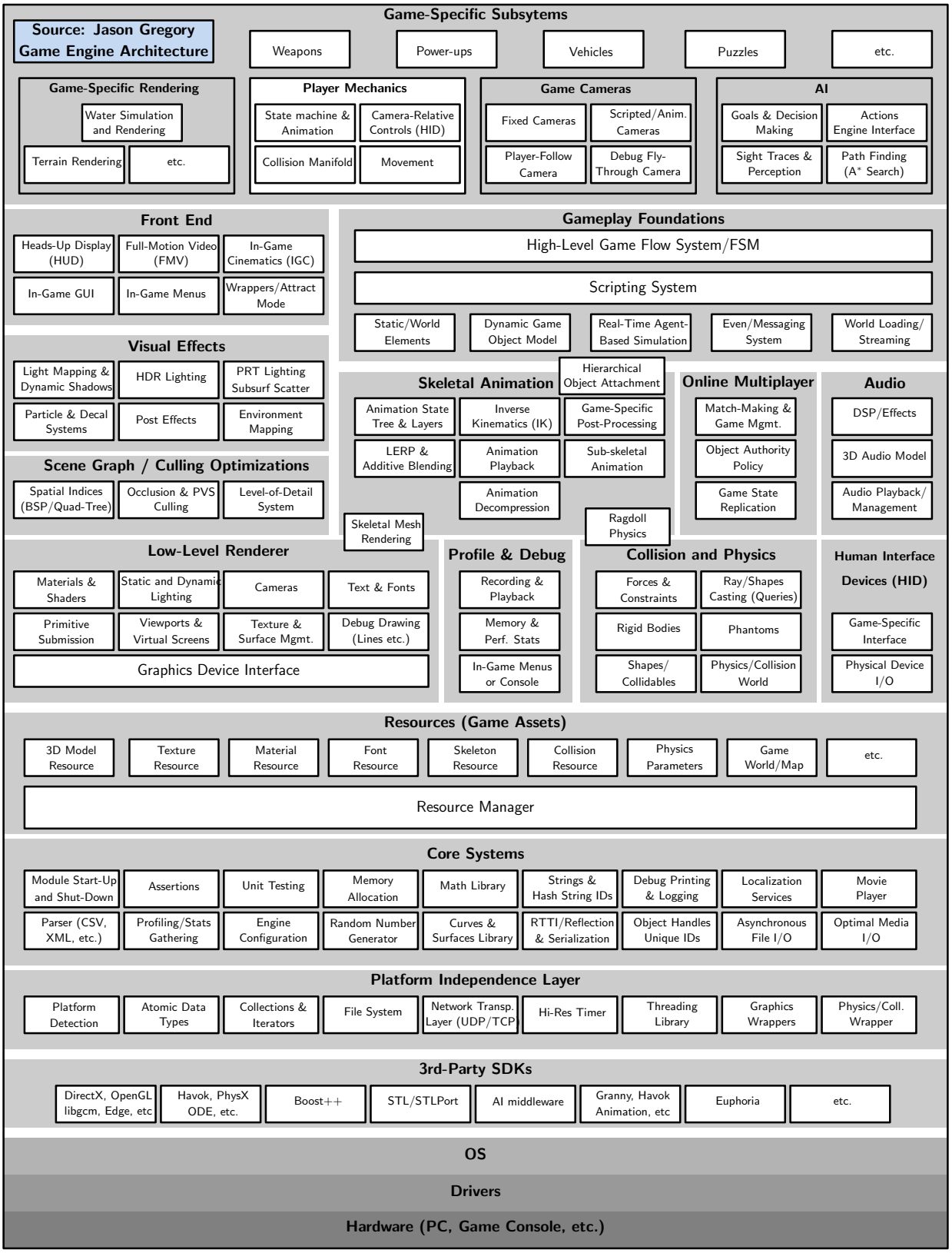
Platform Independence Layer: Since most games are developed to run on many different platforms, this layer provides software to translate between game specific operations and their system-dependent implementations.

Core System: These include basic tools necessary in any software development environment, including assertion testing, unit testing, memory allocation/deallocation, mathematics library, debugging aids, parsers and serializers (e.g., for xml-based import and export), file I/O, video playback.

Resource Manager: Large graphics programs involve accessing various resources, such as geometric models for characters and buildings, texture images for coloring these geometric models, maps representing the game's world. The job of the resource manager is to allow the program to load these resources. Since resources may be compressed to save space, this may also involve decompression.

Rendering Engine: This is one of the largest and most complex components of any real-time 3-dimensional game. This involves all aspects of drawing, and may involve close interaction with the graphics processing unit (GPU) for the sake of enhanced efficiency.

Low-Level Renderer: This comprises the most basic elements of producing images. Your program interacts with the GPU by asking it to render *objects*. Each object may be as simple


Source: Jason Gregory
Game Engine Architecture
Game-Specific Subsytems
Weapons
Power-ups
Vehicles
Puzzles
etc.
Game-Specific Rendering
Water Simulation and Rendering
Terrain Rendering
etc.
Player Mechanics
State machine & Animation
Camera-Relative Controls (HID)
Collision Manifold
Movement
Game Cameras
Fixed Cameras
Scripted/Anim. Cameras
Player-Follow Camera
Debug Fly-Through Camera
AI
Goals & Decision Making
Actions Engine Interface
Sight Traces & Perception
Path Finding (A* Search)
Front End
Heads-Up Display (HUD)
Full-Motion Video (FMV)
In-Game Cinematics (IGC)
In-Game GUI
In-Game Menus
Wrappers/Attract Mode
Gameplay Foundations
High-Level Game Flow System/FSM
Scripting System
Static/World Elements
Dynamic Game Object Model
Real-Time Agent-Based Simulation
Even/Messaging System
World Loading/ Streaming
Visual Effects
Light Mapping & Dynamic Shadows
HDR Lighting
PRT Lighting Subsurf Scatter
Particle & Decal Systems
Post Effects
Environment Mapping
Skeletal Animation
Hierarchical Object Attachment
Animation State Tree & Layers
Inverse Kinematics (IK)
Game-Specific Post-Processing
LERP & Additive Blending
Animation Playback
Sub-skeletal Animation
Animation Decompression
Online Multiplayer
Match-Making & Game Mgmt.
Object Authority Policy
Game State Replication
Audio
DSP/Effects
3D Audio Model
Audio Playback/ Management
Scene Graph / Culling Optimizations
Spatial Indices (BSP/Quad-Tree)
Occlusion & PVS Culling
Level-of-Detail System
Skeletal Mesh Rendering
Ragdoll Physics
Low-Level Renderer
Materials & Shaders
Static and Dynamic Lighting
Cameras
Text & Fonts
Primitive Submission
Viewports & Virtual Screens
Texture & Surface Mgmt.
Debug Drawing (Lines etc.)
Graphics Device Interface
Profile & Debug
Recording & Playback
Memory & Perf. Stats
In-Game Menus or Console
Collision and Physics
Forces & Constraints
Ray/Shapes Casting (Queries)
Rigid Bodies
Phantoms
Shapes/ Collidables
Physics/Collision World
Human Interface Devices (HID)
Game-Specific Interface
Physical Device I/O
Resources (Game Assets)
3D Model Resource
Texture Resource
Material Resource
Font Resource
Skeleton Resource
Collision Resource
Physics Parameters
Game World/Map
etc.
Resource Manager
Core Systems
Module Start-Up and Shut-Down
Assertions
Unit Testing
Memory Allocation
Math Library
Strings & Hash String IDs
Debug Printing & Logging
Localization Services
Movie Player
Parser (CSV, XML, etc.)
Profiling/Stats Gathering
Engine Configuration
Random Number Generator
Curves & Surfaces Library
RTTI/Reflection & Serialization
Object Handles Unique IDs
Asynchronous File I/O
Optimal Media I/O
Platform Independence Layer
Platform Detection
Atomic Data Types
Collections & Iterators
File System
Network Transp. Layer (UDP/TCP)
Hi-Res Timer
Threading Library
Graphics Wrappers
Physics/Coll. Wrapper
3rd-Party SDKs
DirectX, OpenGL libgcm, Edge, etc
Havok, PhysX ODE, etc.
Boost++
STL/STLPort
AI middleware
Granny, Havok Animation, etc
Euphoria
etc.
OS
Drivers
Hardware (PC, Game Console, etc.)

as a single triangle but is more typically a mesh consisting of many triangular elements. Objects are specified according to their coordinates in 3-dimensional space. Your program also informs the GPU what colors (or what image textures) to apply to these objects, where lights are positioned, and where the camera is positioned.

It is then the job of the GPU to perform the actual rendering (projection, coloring, shading) of the objects. In particular, it determines where each object projects onto the 2-dimensional image plane, which objects are visible and which are hidden from view, what is the color and brightness of each object. Your program needs to convey all this information to the GPU. This also includes elements like displaying text messages and subdividing the window into subwindows (called *viewports*) for the purposes of showing status information or maps.

Graphics Device Interface: Since the graphics device requires updating 30–100 times per second, but some operations (such as displaying messages to the user) occurs at a significantly different time scale (of multiple seconds), these components shield the programmer from some of the low-level timing issues when dealing with the graphics system.

Scene Graph: Game entities are naturally organized into hierarchical structures. This is true for dynamic and static objects. For example, a human body consists of a head, torso, arms, legs; an arm consists of a hand, lower-arm, and upper-arm; a hand consists of fingers. Thus, there is a natural structure in the form of a rooted tree.

In general, all the entities of the games world can be represented in a large tree, where the root represents the entire world, and the nodes of the tree implicitly represent the subtrees that are descended from these nodes. This makes it possible to perform operations easily on an entire portion of the tree. For example, we could "render the objects rooted at node u" or "rotate the object rooted at node v." We can also create multiple instances, as in "create 200 instances of the zombie object at node z."

This software component is responsible for creating, modifying, rendering, manipulating, and deleting elements of the scene graph. Another feature of using a scene graph is that it allows us to remove, or *cull*, entities that are not visible to the camera. For example, if the camera is located in a room represented by some node v, we need only render the objects lying within this room or that are visible through the room's doors and windows. Because game worlds are so large and complex, efficient rendering demands that we only attempt to draw the things that might be visible to the camera.

Visual Effects: This includes support for a number of complex effects such as:

- particle systems (which are used for rendering smoke, water, fire, explosions)
- decal systems (for painting bullet holes, damage scratches, powder marks from explosions, foot prints, etc)
- complex lighting, shadowing, and reflections

Others: This includes visual elements of the user interface, such as displaying menus or debugging aids (for the developer) and video playback for generating the back story.

Collisions and Physics: These components simulate the movement of objects over time, detect when objects collide with one another, and determine the appropriate response in the event of a collision (like knocking down the houses where the little pigs live). Except in very simple physical phenomena, like a free-falling body, physical simulation can be very difficult. For this reason, it is often handled by a third-party physics SDK.

Animation: While the game may direct a character to move from one location to another, the job of the animation system is to make this motion look natural, for example, by moving the arms and legs in a manner that is consistent with a natural walking behavior. The typical process for most animals (including humans) involves developing a skeletal representation of the object and wrapping flesh (which is actually a mixture of skin and clothing) around this skeleton. The skeleton is moved by specifying the changes in the angles of the various joints. This approach is called a *skin and bones* representation.

Input Handlers: These components process inputs from the user, including keyboard, mouse, and game controller inputs. Some devices also provide feedback to users (such as the vibration in some game controllers). Modern vision based systems, such as the XBox Kinect, add a whole new level of complexity to this process.

Audio: Audio components handle simple things like playback for background music, explosions, car engines, tire squealing, but they may also generate special audio effects, such as stereo effects to give a sense of location, echo effects to give a sense of context (inside versus outside), and other audio effects (such as creating a feeling of distance by filtering out high-frequency components).

Multiplayer/Networking: Multiplayer and online games require a number of additional supporting components. For example, multiplayer games may have a split-screen capability. Online games require support for basic network communication (serialization of structures into packets, network protocol issues, hiding network latency, and so on). This also includes issues in the design of game servers, such as services for maintaining registered users and their passwords, matching players with one another, and so on.

Gameplay Foundation System: The term *gameplay* refers to the rules that govern game play, that is, the player's possible actions and the consequences of these actions. Since this varies considerably from one game to another, designing a system that works for a wide variety of games can be quite daunting. Often, these systems may be designed to support just a single genre of games, such as FPS games. There are a number of subcomponents:

Game Worlds and Object Models: This constitutes the basic entities that make up the game's world. Here are some examples:

- static background objects (buildings, terrain, roads)
- (potentially) dynamic background objects (rocks, chairs, doors and windows)
- player characters (PCs) and non-player characters (NPCs)
- weapons and projectiles
- vehicles
- graphics elements (camera and lights)

The diversity of possible game objects is a major challenge to programmers trained in object-oriented methods. Objects share certain universal qualities (e.g., they can be created and destroyed), common qualities (e.g., they can be rendered), and more specialized qualities (e.g., gun-like objects can be shot, chair-like objects can be sat on).

Event System: Game objects need to communicate with one another. This is often handled by a form of message passing. When a message arrives, we can think of it as implicitly signaling an event to occur, which is to be processed by the entity. Game objects can register their interest in various events, which may affect their behavior. (For example, characters need to be made aware of explosions occurring in their vicinity. When such an explosion occurs, the system informs nearby characters by sending them a message..."you're toast, buddy.")

Scripting System: In order to allow game developers to rapidly write and test game code, rather than have them write in a low-level programming language, such as C++, it is common to have them produce their code in a scripting language, like Python. A sophisticated scripting system can handle the editing of scripts and reloading a script into a game while it is running.

Artificial Intelligence: These components are used to control the behavior of non-player characters. They are typically modeled as AI *agents*. As we will discuss later, an agent is an object that can sense its environment, maintain a memory (or state), and can respond in potentially complex ways depending on the current input and state.

Game Specific Systems: This is a catch-all for any components that are so specific to a given game that they don't fall into one of the other categories. This may include aspects like the mechanics controlling a player's state, the algorithms by which a camera moves throughout the world, the specific goals of the game's non-player characters, the properties of various weapons, and so on.

Interactive 3-dimensional Graphics: (Optional material)

In order to get a quick jump into game development, we will start by discussing the basic elements of interactive computer graphics systems. This will require that we understand a bit about how graphics processing units work and will lead us eventually into a discussion of OpenGL.

Anyone who has played a computer game is accustomed to interaction with a graphics system in which the principal mode of rendering involves 3-dimensional scenes. Producing highly realistic, complex scenes at interactive frame rates (at least 30 frames per second, say) is made possible with the aid of a hardware device called a *graphics processing unit*, or *GPU* for short. GPUs are very complex things, and we will only be able to provide a general outline of how they work.

Like the CPU (central processing unit), the GPU is a critical part of modern computer systems. (See Fig. 1 for a schematic representation.) Because of its central importance, the GPU is connected to the CPU and the system memory through a high-speed data transfer interface, which is current systems architecture terminology, is called the *north bridge*. The GPU has its own memory, separate from the CPU's memory, in which it stores the various graphics objects (e.g., vertex coordinates and textures) that it needs in order to do its job. The GPU is highly parallel, and it has a very high capacity connection with its memory. Part of this memory is called the *frame buffer*, which is a dedicated chunk of memory where the pixels associated with your monitor are stored.

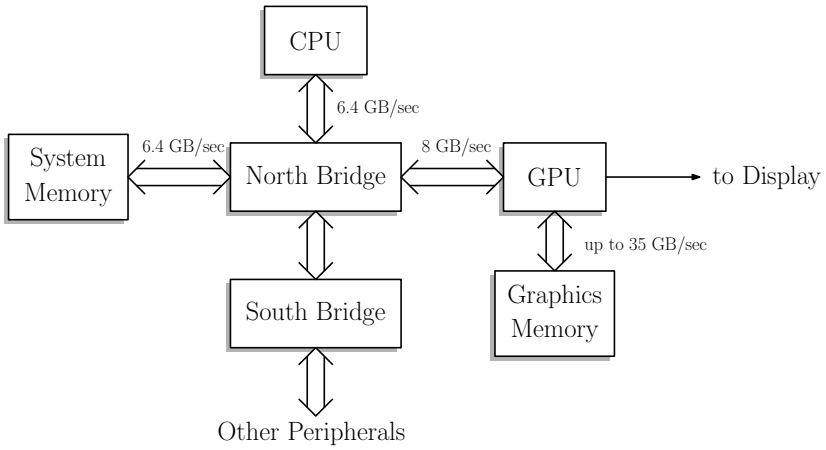


Fig. 1: Architecture of a simple GPU-based graphics system. (Adapted from NVIDIA GeForce documentation.)

Traditionally, GPUs are designed to perform a relatively limited fixed set of operations, but with blazing speed and a high degree of parallelism. Modern GPUs are *programmable*, in that they provide the user the ability to program various elements of the graphics process. For example, modern GPUs support programs called *vertex shaders* and *fragment shaders*, which provide the user with the ability to fine-tune the colors assigned to vertices and fragments.

Recently there has been a trend towards what are called *general purpose GPUs* (GPGPUs), which can perform not just graphics rendering, but general scientific calculations on the GPU. Since we are interested in graphics here, we will focus on the GPUs traditional role in the rendering process.

The Graphics Pipeline: The key concept behind all GPUs is the notion of the *graphics pipeline*. This is conceptual tool, where your user program sits at one end sending graphics commands to the GPU, and the frame buffer sits at the other end. A typical command from your program might be "draw a triangle in 3-dimensional space at these coordinates." The job of the graphics system is to convert this simple request to that of coloring a set of pixels on your display. The process of doing this is

quite complex, and involves a number of stages. Each of these stages is performed by some part of the pipeline, and the results are then fed to the next stage of the pipeline, until the final image is produced at the end.

Broadly speaking the pipeline can be viewed as involving a number of stages (see Fig. 2). Geometric objects, called *primitives*, are introduced to the pipeline from your program. Objects are described in terms of vectors in 3-dimensional space (for example, a triangle might be represented by three such vectors, one per vertex).

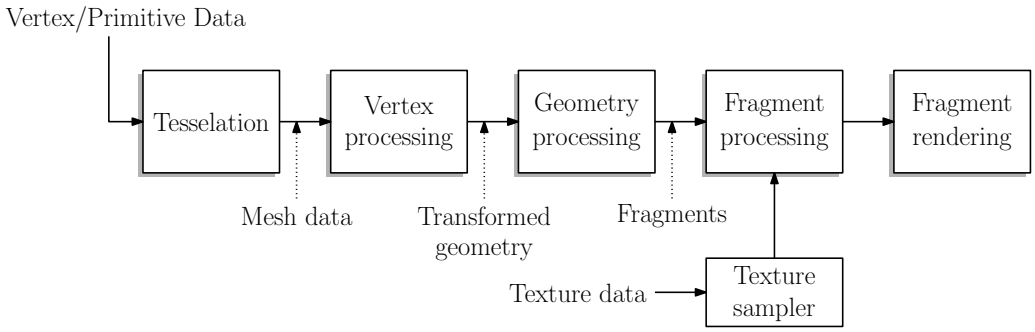


Fig. 2: Stages of the graphics pipeline.

Tesselation: Converts higher-order primitives (such as surfaces), displacement maps, and mesh patches to 3-dimensional vertex locations and stores those locations in *vertex buffers*, that is, arrays of vertex data.

Vertex processing: Vertex data is *transformed* from the user's coordinate system into a coordinate system that is more convenient to the graphics system. For the purposes of this high-level overview, you might imagine that the transformation projects the vertices of the three-dimensional triangle onto the 2-dimensional coordinate system of your screen, called *screen space*.

Geometry processing: This involves a number of tasks:

- *Clipping* is performed to snip off any parts of your geometry that lie outside the viewing area of the window on your display.
- *Back-face culling* removes faces of your mesh that lie on the side of an object that faces away from the camera.
- *Lighting* determines the colors and intensities of the vertices of your objects. Lighting is performed by a program called a *vertex shader*, which you provide to the GPU.
- *Rasterization* converts the geometric shapes (e.g., triangles) into a collection of pixels on the screen, called *fragments*.

Texture sampling: Texture images are sampled and smoothed and the resulting colors are assigned to individual fragments.

Fragment Processing: Each fragment is then run through various computations. First, it must be determined whether this fragment is *visible*, or whether it is hidden behind some other fragment. If it is visible, it will then be subjected to coloring. This may involve applying various coloring textures to the fragment and/or color blending from the vertices, in order to produce the effect of smooth shading.

Fragment Rendering: Generally, there may be a number of fragments that affect the color of a given pixel. (This typically results from translucence or other special effects like motion blur.) The colors of these fragments are then blended together to produce the final pixel color. Fog effects may also be involved to alter the color of the fragment. The final output of this stage is the *frame-buffer image*.

Lecture 3: Introduction to Unity

Unity: Unity is a widely-used cross-platform game develop system. It consists of a game engine and an integrated development environment (IDE). It can be used to develop games for many different platforms, including PCs, consoles, mobile devices, and deployment on the Web. In this lecture, we will present the basic elements of Unity. However, this is a complex system, and we will not have time to delve into its many features. A good starting point for learning about Unity is to try the many tutorials available on the Unity Tutorial Web site.

Unity Basic Concepts: The fundamental structures that make up Unity are the same as in most game engines. As with any system, there are elements and organizational features that are unique to this particular system.

Project: The *project* contains all the elements that makes up the game, including models, assets, scripts, scenes, and so on. Projects are organized hierarchically in the same manner as a file-system's folder structure.

Scenes: A *scene* contains a collection of game objects that constitute the world that the player sees at any time. A game generally will contain many scenes. For example, different levels of a game would be stored as different scenes. Also, special screens (e.g., an introductory screen), would be modeled as scenes that essentially have only a two-dimensional content.

Packages: A *package* is an aggregation of game objects and their associated meta-data. Think of a package in the same way as library packages in Java. They are related objects (models, scripts, materials, etc.). Here are some examples:

- a collection of shaders for rendering water effects
- particle systems for creating explosions
- models of race cars for a racing game
- models of trees and bushes to create a woodland scene

Unity provides a number standard packages for free, and when a new project is created, you can select the packages that you would like to have imported into your project.

Prefabs: A *prefab* is a template for grouping various assets under a single header. Prefabs are used for creating multiple instances of a common object. Prefabs are used in two common ways. First, in designing a level for your game you may have a large number of copies of a single element (e.g., street lights). Once designed, a street light prefab can be instantiated and placed in various parts of the scene. If you decide to want to change the intensity of light for all the street lights, you can modify the prefab, and this will cause all the instances to change. A second use is to generate dynamic game objects. For example, you could model an explosive shell shot from a cannon as a prefab. Each time the cannon is shot a new prefab shell would be instantiated (through one of your scripts). In this way each new instance of the shell will inherit all the prefabs properties, but it will have its own location and state.

Game Objects: The *game objects* are all the "things" that constitute your scene. Game objects not only include concrete objects (e.g., a chair in a room), but also other elements that reside in space such as light sources, audio sources, and cameras. Empty game objects are very useful, since they can to serve as parent nodes in the hierarchy. Every game object (even empty ones) has a position and orientation space. This, it can be moved, rotated and scaled. (As mentioned above, whenever a transformation is applied to a parent object, it is automatically propagated to all of this object's descendants descendants.)

Game objects can be used for invisible entities that are used to control a game's behavior. (For example, suppose that you want a script to be activated whenever the player enters a room. You could create an invisible portal object covering the door to the room that triggers an event

whenever the player passes through it.) Game objects can be enabled or disabled. (Disabled objects behave as if they do not exist.) It is possible to associate various elements with game objects, called components, which are described below.

Components: As mentioned above, each game object is defined by a collection of associated elements. These are called *components*. The set of components that are associated with a game object depend on the nature of object. For example, a light source object is associated with color and intensity of the light source. A camera object is associated with various properties of how the projection is computed (wide-angle or telephoto). Physical objects of the scene are associated with many different components. For example, these might include:

- A *mesh filter* and *mesh renderer* are components that define the geometric surface model for the object and the manner in which it is drawn, respectively.
- A *rigid body* component that specifies how the object moves physically in space by defining elements like the object's mass, drag (air resistance), and whether the object is affected by gravity.
- A *collider* which is an imaginary volume that encloses the object and is used to determine whether the object collides with other objects from the scene. (In theory, the object's mesh describes it shape and hence be used for computing collisions, but for the sake of efficiency, it is common to use a much simpler approximating shape, such as a bounding box or a bounding sphere, when detecting collisions.)
- Various *surface materials*, which describe the object's color, texture, and shading.
- Various *scripts*, which control how the object behaves and how it reacts to its environment. One example of a script is a *player controller*, which is used for the player object and describes how the object responds to user inputs.

The various components that are associated with an game object can be viewed and edited in the Inspector window (described below).

Assets: An *asset* is any resource that will be used as part of an object's component. Examples include meshes (for defining the shapes of objects), materials (for defining shapes), physics materials (for defining physical properties like friction), and scripts (for defining behaviors).

Scripts: A *script* is a chunk of code that defines the behavior of game objects. Scripts are associated with game objects. There are various types of scripts classes, depending on the type of behavior being controlled. Because interactive game programming is *event-driven*, a typical script is composed as a collection of functions, each of which is invoked in response to a particular event. Each of these functions performs some simple action (e.g., moving the game object, creating/destroying game objects, triggering events for other game objects), and then returns control to the system.

Overview of the Unity IDE: Having described the basic concepts underlying Unity, let us next take a quick look at the Unity user interface. As mentioned above, Unity provides a integrated development environment in which to edit and develop your game. While the user interface is highly configurable, there are a few basic windows which are the most frequently used (see Fig. 3).

Scene View: This window shows all the elements of the current scene. (See description below for what a scene is.) Most editing of the scene is done through the scene view, because it provides access to low-level and hidden aspects of the objects. For example, this view will show you where the camera and light sources are located. In contrast, the Game View, shows the game as it would appear to the player.

Game View: This window shows the elements of the scene as they would appear to the player.

Inspector: At any time there is an *active* game object (which the designer selects by clicking on the object or on its entry in the hierarchy). This window provides all the component information associated with this object. At a minimum, this includes its position and orientation in space. However it also has entries for each of the components associated with this object.

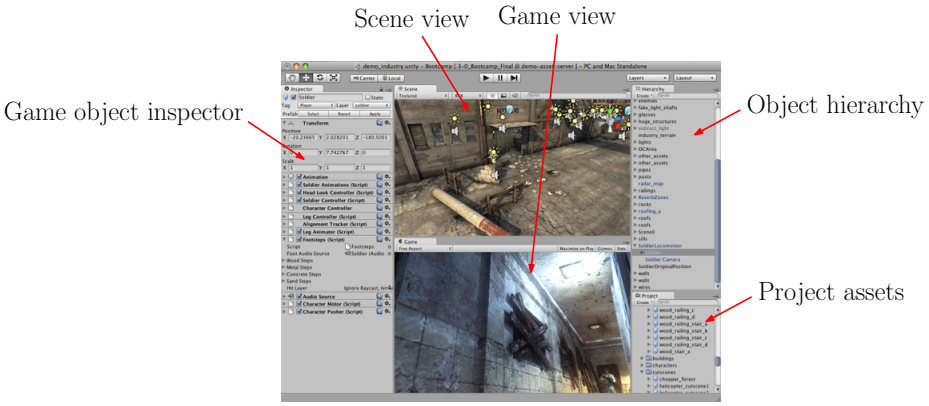


Fig. 3: Unity IDE.

Hierarchy: This window shows all the game objects that constitute the current scene. (Scenes are discussed below). As its name suggests, game objects are stored hierarchically in a tree structure. This makes it possible so that transformations applied to a parent object are then propagated to all of its descendents. For example, if a building is organized as a collection of rooms (descended from the building), and each room is organized as a collection of pieces of furniture (descended from the room), then moving the building will result in all the rooms and pieces of furniture moving along with it.

Project: The project window contains all of the assets that are available for you to use. Typically, these are organized into folders, for example, according to the asset type (models, materials, audio, prefabs, scripts, etc.).

Scripting in Unity: As mentioned above, scripting is used to describe how game objects behave in response to various events, and therefore it is an essential part of the design of any game. Unity supports three different scripting languages: C#, UnityScript (a variant of JavaScript), and Boo (a variant of Python). (I will use C# in my examples. At a high level, C# is quite similar to Java, but there are minor variations in syntax and semantics.) Recall that a script is an example of a component that is associated with an game object. In general, a game object may be associated with multiple scripts. (In Unity, this is done by selecting the game object, adding a component to it, and selecting an existing predefined script or "New Script" as the component type.)

Geometric Elements: Unity supports a number of objects to assist with geometric processing. We will discuss these objects in greater detail in a later lecture, but here are a few basic facts.

Vector3: This is standard (x, y, z) vector. As with all C# objects, you need to invoke "new" when creating a Vector3 object. The following generates a Vector3 variable u with coordinates $(1, 2, -3)$:

```
Vector3 u = new Vector3(1, 2, -3);
```

The orientation of the axes follows Unity's (mathematically nonstandard) convention that the y-axis is directed upwards, the x-axis points to the viewer's right, and the z-axis points to the viewer's forward direction. (Of course, as soon as the camera is repositioned, these orientations change.)

It is noteworthy that Unity's axis forms what is called a *left-handed coordinate system*, which means that $x \times y = -z$ (as opposed to $x \times y = z$, which holds in most mathematics textbooks as well as other 3D programming systems, such as UE4 and Blender).

To make it a bit more natural in programming, Unity provides function calls that return the unit vectors in natural directions. For example, Vector3.right return $(1, 0, 0)$, Vector3.up returns $(0, 1, 0)$, and Vector3.forward returns $(0, 0, 1)$. Others include left, down, back, and zero.

Ray: Rays are often used in geometric programming to determine the object that lies in a given direction from a given location. A ray is specified by giving its origin and direction. The following creates a ray starting at the origin and directed along the x-axis.

```
Ray ray = new Ray(Vector3.zero, Vector3.right);
```

We will discuss how to perform ray-casting queries in Unity and how these can be applied in your programs.

Quaternion: A quaternion is a structure that represents a rotation in 3-dimensional space. There are many ways to provide Unity with a rotation. The two most common are through the use of *Euler angles*, which means specifying three rotation angles, one about the x-axis, one about the y-axis, and one about the z-axis. The other is by specifying a Vector3 as an axis of rotation and a rotation angle about this axis. For example, the following both create the same quaternion, which performs a 30° rotation about the vertical (that is, y) axis.

```
Quaternion q1 = new Quaternion(0, 30, 0);
Quaternion q2 = Quaternion.AngleAxis(30, Vector3.up);
```

Transform: Every game object in Unity is associated with an object called its transform. This object stores the position, rotation, and scale of the object. You can use the Transform object to query the object's current position (transform.position) and rotation (transform.eulerAngles).

You can also modify the transform to reposition the object in space. These are usually done indirectly through functions that translate (move) or rotate the object in space. Here are examples:

```
transform.Translate(new Vector3(0, 1, 0)); // move up one unit
transform.Rotate(0, 30, 0); // rotate 30 degrees about y-axis
```

You might wonder whether these operations are performed relative to the global coordinate system or the object's local coordinate system. The answer is that there is an optional parameter (not shown above) that allows you to select the coordinate system about which the operation is to be interpreted.

Recall that game object's in Unity reside within a hierarchy, or tree structure. Transformations applied to an object apply automatically to all the descendants of this object as well. The tree structure is accessible through the transform. For example, transform.parent returns the transform of the parent (and transform.parent.gameObject returns the associated Unity game object). You can set a transforms parent using transform.SetParent(t), where t is the transform of the parent object. It is also possible to enumerate the children and all the descendants of a transform.

Structure of a Typical Script: A game object may be associated with a number of scripts. Ideally, each script is responsible for a particular aspect of the game object's behavior. The basic template for a Unity script, called MainPlayer, is given in the following code block.

Observe a few things about this code fragment. First, the first using statements provides access to class objects defined for the Unity engine, and the second provides access to built-in collection data structures (ArrayList, Stack, Queue, HashTable, etc.) that are part of C#. The main class is MainPlayer, and it is a subclass of MonoBehaviour. All script objects in Unity are subclasses of MonoBehaviour.

Many script class objects will involve: (1) some sort of initialization and (2) some sort of incremental updating just prior to each refresh cycle (when the next image is drawn to the display). The template facilitates this by providing you with two blank functions, Start and Update, for you to fill in. Of course, there is no requirement to do so. For example, your script may require no explicit initializations (and thus there is no need for Start), or rather than being updated with each refresh cycle, your script may be updated in response to specific user inputs or collisions (and so there would be no need for Update).

Script template

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {
    void Start () {
        // ... insert initialization code here
    }
    void Update () {
        // ... insert code to be repeated every update cycle
    }
}
```

Awake versus Start: There are two Unity functions for running initializations for your game objects, Start (as shown above) and Awake. Both functions are called at most once for any game object. Awake will be called first, and is called as soon as the object has been initialized by Unity. However, the object might not yet appear in your scene because it has not been *enabled*. As soon as your object is enabled, the Start is called.

Let me digress a moment to discuss enabling/disabling objects. In Unity can "turned-on" or "turned-off" in two different ways (without actually deleting them). In particular, objects can be *enabled* or *disabled*, and objects can be *active* or *inactive*. (Each game object in Unity has two fields, enabled and active, which can be set to true or false.) The difference is that disabling an object stops it from being rendered or updated, but it does not disable other components, such as colliders. In contrast, making an object *inactive* stops all its components.

For example, suppose that some character in your game is spawned only after a particular event takes place (you cast a magic spell). The object can initially be declared to be *disabled*, and later when the spawn event occurs, you ask Unity to enable it. Awake will be called on this object as soon as the game starts. Start will be called as soon as the object is enabled. If you later disable and object and re-enable it, Start *will not* be called again. (Both functions are called at most once.)

To make your life simple, it is almost always adequate to use just the Start function for one-time initializations. If there are any initializations that *must* be performed just as the game is starting, then Awake is one to use.

Controlling Animation Times: As mentioned above, the Update function is called with each update-cycle of your game. This typically means that every time your scene is redrawn, this function is called. Redraw functions usually happen very quickly (e.g., 30–100 times per second), but they can happen more slowly if the scene is very complicated. The problem that this causes is that it affects the speed that objects appear to move. For example, suppose that you have a collection of objects that spin around with constant speed. With each call to Update, you rotate them by 3°. Now, if Update is called twice as frequently on one platform than another, these objects will appear to spin twice as fast. This is not good!

The fix is to determine how much time as elapsed since the last call to Update, and then scale the rotation amount by the elapsed time. Unity has a predefined variable, Time.deltaTime, that stores the amount of elapsed time (in seconds) since the last call to Update. Suppose that we wanted to rotate an object at a rate of 45° per second about the vertical axis. The Unity function transform.Rotate will do this for us. We provide it with a vector about which to rotate, which will usually be $(0, 1, 0)$ for the up-direction, and we multiply this vector times the number of degrees we wish to have in the rotation. In order to achieve a rotation of 45° per second, we would take the vector $(0, 45, 0)$ and scale it by Time.deltaTime in our Update function. For example:

```
void Update () {
    transform.Rotate (new Vector3 (0, 45, 0) * Time.deltaTime);
}
```

By the way, it is a bit of a strain on the reader to remember which axis points in which direction. The Vector3 class defines functions for accessing important vectors. The call Vector3.up returns the vector $(0, 1, 0)$. So, the above call would be equivalent to transform.Rotate (Vector3.up * 45 * Time.deltaTime).

Update versus FixedUpdate: While we are discussing timing, there is another issue to consider. Sometimes you want the timing between update calls to be predictable. This is true, for example, when updating the physics in your game. If acceleration is changing due to gravity, you would like the effect to be applied at regular intervals. The Update function does not guarantee this. It is called at the refresh rate for your graphics system, which could be very high on a high-end graphics platform and much lower for a mobile device.

Unity provides a function that is called in a predictable manner, called FixedUpdate. When dealing with the physics system (e.g., applying forces to objects) it is better to use FixedUpdate than Update. When using FixedUpdate, the corresponding elapsed-time variable is Time.fixedDeltaTime. (I've read that Time.fixedDeltaTime is 0.02 seconds, but I wouldn't bank on that.)

While I am discussing update functions, let me mention one more. LateUpdate() is called after all Update functions have been called but before redrawing the scene. This is useful to order script execution. For example a follow-camera should always be updated in LateUpdate because it tracks objects that might have moved due to other Update function calls.

Accessing Components: As mentioned earlier, each game object is associated with a number of defining entities called its *components*. The most basic is the transform component, which describes where the object is located. Most components have constant values, and can be set in the Unity editor (for example, by using the AddComponent command. However, it is often desirable to modify the values of components at run time. For example, you can alter the buoyancy of a balloon as it loses air or change the color of a object to indicate the presence of damage.

Unity defines class types for each of the possible components, and you can access and modify this information from within a script. First, in order to obtain a reference to a component, you use the command GetComponent. For example, to access the rigid-body component of the game object associated with this script, you could invoke. Recall that this component controls the physics properties of the object.

```
Rigidbody rb = GetComponent<Rigidbody>(); // get rigidbody component
```

This returns a reference rb to this object's rigid body component, and similar calls can be made to access any of the other components associated with a game object. (By the way, this call was not really needed. Because the rigid body is such a common component to access, every MonoBehaviour object has a member called rigidbody, which contains a reference to the object's rigid body component, or null if there is none.)

Public Variables and the Unity Editor: One of the nice things that the Unity IDE provides you with is the ability to modify the member variables of your game objects directly within the editor. For example, suppose that you have a moving object that has an adjustable parameter. Consider the following code fragment that is associated with a floating ball game object. The script defines a public member variable floatSpeed to control the speed at which a ball floats upwards.

```
public class BallBehavior : MonoBehaviour {
```

```
    public float floatSpeed = 10.0f; // how fast ball floats up
    public float jumpForce = 4.0f;   // force applied when ball jumps
    ...
}
```

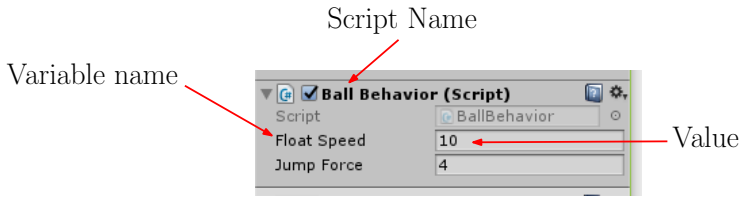


Fig. 4: Editing a public variable in the inspector.

When you are running the program in the Unity editor, you can adjust the values of floatSpeed and jumpForce (while the program is running) until you achieve the desired results. If you like, you can then fix their values in the script and make them private.

Note that there are three different ways that the member variable floatSpeed could be set:

(1) It could be initialized as part of its declaration, public floatSpeed = 6.0f;

(2) It could be set by you in the Unity editor (as above)

(3) It could be initialized in the script, e.g., in the Start() function.

Note that (3) takes precedence over (2), which takes precedence over (1).

By the way, you can do this not only for simple variables as above, but you can also use this mechanism for passing game objects through the public variables of a script. Just make the game object variable public, then drag the game object from the hierarchy over the variable value in the editor.

Object References by Name or Tag: Since we are on the subject of how to obtain a reference from one game object to another, let us describe another mechanism for doing this. Each game object is associated with a *name*, and its can be also be associated with one more *tags*. Think of a name as a unique identifier for the game object (although, I don't think there is any requirement that this be so), whereas a tag describes a type, which might be shared by many objects.

Both names and tags are just a string that can be associated with any object. Unity defines a number of standard tags, and you can create new tags of your own. When you create a new game object you can specify its name. In each object's inspector window, there is a pull-down menu that allows you associate any number of tags with this object.

Here is an example of how to obtain a the main camera reference by its name:

```
GameObject camera = GameObject.Find ("Main Camera");
```

Suppose that we assign the tag "Player" with the player object and "Enemy" with the various enemy objects. We could access the object(s) through the tag using the following commands:

```
GameObject player = GameObject.FindWithTag("Player");
GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");
```

In the former case, we assume that there is just one object with the given tag. (If there is none, then null is returned. If there is more than one, it returns one one of them.) In the latter case, all objects having the given tag are returned in the form of an array.

Note that there are many other commands available for accessing objects and their components, by name, by tag, or by relationship within the scene graph (parent or child). See the Unity documentation

for more information. The Unity documentation warns that these access commands can be relatively *slow*, and it is recommended that you execute them once in your Start() or Awake() functions and save the results, rather than invoking them with every update cycle.

Accessing Members of Other Scripts: Often, game objects need to access member variables or functions in other game objects. For example, your enemy game object may need to access the player's transform to determine where the player is located. It may also access other functions associated with the player. For example, when the enemy attacks the player, it needs to invoke a method in the player's script that decreases the player's health status.

```
public class PlayerController : MonoBehaviour {
    public void DecreaseHealth() { ... } // decrease player's health
}

public class EnemyController : MonoBehaviour {
    public GameObject player; // the player object
    void Start () {
        GameObject player = GameObject.Find("Player");
    }
    void Attack () { // inflict health loss on player
        player.GetComponent<PlayerController>().DecreaseHealth();
    }
}
```

Note that we placed the call to GameObject.Find in the Start function. This is because this operation is fairly slow, and ideally should be done sparingly.

Colliders and Triggers: Games are naturally *even driven*. Some events are generated by the user (e.g., input), some occur at regular time intervals (e.g., Update()), and finally others are generated within the game itself. An important class of the latter variety are collision events. Collsions are detected by a component called a *collider*. Recall that this is a shape that (approximately) encloses a given game object.

Colliders come in two different types, *colliders* and *triggers*. Think of colliders as solid physical objects that should not overlap, whereas a *trigger* is an invisible barrier that sends a signal when crossed.

For example, when a rolling ball hits a wall, this is a collider type event, since the ball should not be allowed to pass through the wall. On the other hand, if we want to detect when a player enters a room, we can place an (invisible) trigger over the door. When the player passes through the door, the trigger event will be generated.

There are various event functions for detecting when an object enters, stays within, or exits, collider/trigger region. These include, for example:

- For colliders: void OnCollisionEnter(), void OnCollisionStay(), void OnCollisionExit()
- For triggers: void OnTriggerEnter(), void OnTriggerStay(), void OnTriggerExit()

More about RigidBody: Earlier we introduced the rigid-body component. What can we do with this component? Let's take a short digression to discuss some aspects of rigid bodies in Unity. We can use this reference to alter the data members of the component, for example, the body's mass:

```
rb.mass = 10f;     // change this body's mass
```

Unity objects can be controlled by physics forces (which causes them to move) or are controlled directly by the user. One advantage of using physics to control an object is that it will automatically avoid collisions with other objects. In order to move a body that is controlled by physics, you do not set its velocity. Rather, you apply forces to it, and these forces affect it velocity. Recall that from physics, a

force is a vector quantity, where the direction of the vector indicates the direction in which the force is applied.

```
rb.AddForce(Vector3.up * 10f);     // apply an upward force
```

Sometimes it is desirable to take over control of the body's movement yourself. To turn off the effect of physics, set the rigid body's type to *kinematic*.

```
rb.isKinematic = true;  // direct motion control---bypass physics
```

Once a body is kinematic, you can directly set the body's velocity directly, for example.

```
rb.velocity = Vector3(0f, 10f, 0f);  // move up 10 units/second
```

(If the body had not been kinematic, Unity would issue an error message if you attempted to set its velocity in this way.) By the way, since the y-axis points up, the above statement is equivalent to setting the velocity to `Vector3.up * 10f`. This latter form is, I think, more intuitive, but I just wanted to show that these things can be done in various ways.

Kinematic and Static: In general, Physics computations can be expensive, and Unity has a number of tricks for optimizing the process. As mentioned earlier, an object can be set to *kinematic*, which means that your scripts control the motion directly, rather than physics forces. Note that this only affects motion. Kinematic objects still generate events in the event of collisions and triggers.

Another optimization involves static objects. Because many objects in a scene (such as buildings) never move, you can declare such objects to be *static*. (This is indicated by a check box in the upper right corner of the object's Inspector window.) Unity does not perform collision detection between two static objects. (It checks for collisions/triggers between static-to-dynamic and dynamic-to-dynamic, but not static-to-static.) This can save considerable computation time since many scenes involve relatively few moving objects. Note that you can alter the static-dynamic property of an object, but the documentation warns that this is somewhat risky, since the physics engine precomputes and caches information for static objects, and this information might be erroneous if an object changes this property.

Event Functions: Because script programming is *event-driven*, most of the methods that make up MonoBehaviour scripts are event callbacks, that is, functions that are invoked when a particular event occurs. Examples of events include (1) initialization, (2) physics events, such as collisions, and (3) user-input events, such as mouse or keyboard inputs.

Unity passes the control to each script intermittently by calling a determined set of functions, called *Event Functions*. The list of available functions is very large, here are the most commonly used ones:

Initialization: Awake and Start as mentioned above.

Regular Update Events: Update functions are called regularly throughout the course of the game. These include redraw events (Update and LateUpdate) and physics (or any other regular time events) (FixedUpdate).

GUI Events: These events are generated in response to user-inputs regarding the GUI elements of your game. For example, if you have a GUI element like a push-down button, you can be informed when a mouse button has been pressed down, up, or is hovering over this element with callbacks such as void OnMouseDown(), void OnMouseUp(), void OnMouseOver().They are usually processed in Update or FixedUpdate.

Physics Events: These include the collider and trigger functions mentioned earlier (such as OnCollisionEnter, OnTriggerEnter).

There are *many* things that we have not listed. For further information, see the Unity user manual.

Instantiating Prefabs: (Coming soon)

Coroutines: (Optional material)

Anyone who has worked with event-driven programming for some time has faced the frustration that, while we programmers like to think *iteratively*, in terms of loops, our event-driven graphics programs are required to work *incrementally*. The event-driven structure in an intrinsic part of interactive computer graphics has the same general form: wake up (e.g., at every refresh cycle), make a small incremental change to the state, and go back to sleep (and let the graphics system draw the new scene). In order to make our iterative programs fit within the style, we need to "unravel" our loops to fit this awkward structure.

Unity has an interesting approach to helping with this issue. Through a language mechanism, called *coroutines*, it is possible to implement code that *appears* to be iterative, and yet behaves *incrementally*. When you call a function, it runs to completion before returning. This effectively means that the function cannot run a complete loop, e.g., for some animation, that runs over many refresh cycles. As an example, consider the following code snippet that gradually makes a object transparent until it becomes invisible. Graphics objects are colored using a 4-element vector, called RGBA. The R, G, and B components describe the red, gree, and blue components (where 1 denotes the brightest value and 0 the darkest). The A component is called the colors *alpha* value, which encodes its level of opacity (where 1 is fully opaque and 0 is fully transparent).

```
void Fade() {  // gradually fade from opaque to transparent
  for (float f = 1f; f >= 0; f -= 0.1f) {
    Color c = renderer.material.color;
    c.a = f;
    renderer.material.color = c;
    // ... we want to redraw the object here
  }
}
```

Unfortunately, this code does not work, because we cannot just interrupt the loop in the middle to redraw the new scene. A *coroutine* is like a function that has the ability to pause execution and return (to Unity) but then to continue where it left off on the following frame. To do this, we use the yield return construct from C#, which allows us to call a function multiple times, but each time it is called, it starts not from the beginning, but from where it left off. Such a function has a return type of an iterator, but we will not worry about this for this example.

```
IEnumerator Fade() { // gradually fade from opaque to transparent
  for (float f = 1f; f >= 0; f -= 0.1f) {
    Color c = renderer.material.color;
    c.a = f;
    renderer.material.color = c;
    yield return null; // return to Unity to redraw the scene
  }
}
```

The next time this function is called, it resumes just *after* the return statement in order to start the next iteration of the loop. (Pretty cool!)

If you want to control the timing, so that this happens, say, once every tenth of a second, you can add a delay into the return statement, "yield return new WaitForSeconds(0.1f)".

Lecture 4: Geometry and Geometric Programming

Geometry for Game Programming and Graphics: For the next few lectures, we will discuss some of the basic elements of geometry. There are many areas of computer science that involve computation with geometric entities. This includes not only computer graphics, but also areas like computer-aided design, robotics, computer vision, and geographic information systems. While software systems like Unity provide support to do geometry for you, there are good reasons for learning this material. First, for those of you who will go on to design the successor to Unity, it is important to understand the fundamentals underlying gometric programming. Second, as a game programmer, you will find that there things that Unity *cannot* help with. In such cases, you will need to write scripts to do the geometry yourself.

Computer graphics deals largely with the geometry of lines and linear objects in 3-space, because light travels in straight lines. For example, here are some typical geometric problems that arise in designing programs for computer graphics.

Transformations: You are asked to render a twirling boomerang flying through the air. How would you represent the boomerang's rotation and translation over time in 3-dimensional space? How would you compute its exact position at a particular time?

Geometric Intersections: Given the same boomerang, how would you determine whether it has hit another object?

Orientation: You have been asked to design the AI for a non-player agent in a flight combat simulator. You detect the presence of a enemy aircraft in a certain direction. How should you rotate your aircraft to either attack (or escape from) this threat?

Change of coordinates: We know the position of an object on a table with respect to a coordinate system associated with the table. We know the position of the table with respect to a coordinate system associated with the room. What is the position of the object with respect to the coordinate system associated with the room?

Reflection and refraction: We would like to simulate the way that light reflects off of shiny objects and refracts through transparent objects.

Such basic geometric problems are fundamental to computer graphics, and over the next few lectures, our goal will be to present the tools needed to answer these sorts of questions. There are various formal geometric systems that arise naturally in game programming and computer graphics. The principal ones are:

Affine Geometry: The geometry of simple "flat things": points, lines, planes, line segments, triangles, etc. There is no defined notion of distance, angles, or orientations, however.

Euclidean Geometry: The geometric system that is most familiar to us. It enhances affine geometry by adding notions such as distances, angles, and orientations (such as clockwise and counterclockwise).

Projective Geometry: In Euclidean geometry, there is no notion of infinity (in the same way that in standard arithmetic, you cannot divide by zero). But in graphics, we often need to deal with infinity. (For example, two parallel lines in 3-dimensional space can meet at a common *vanishing point* in a perspective rendering. Think of the point in the distance where two perfectly straight train tracks appear to meet. Computing this vanishing point involves points at infinity.) Projective geometry permits this.

Affine Geometry: Affine geometry is basic to all geometric processing. It's basic elements are:

- *Scalars*, which we can just think of as being real numbers

- *Points*, which define locations in space
- *Free vectors* (or simply *vectors*), which are used to specify direction and magnitude, but have no fixed position.

The term "free" means that vectors do not necessarily emanate from some position (like the origin), but float freely about in space. There is a special vector called the *zero vector*, $\vec{0}$, that has no magnitude, such that $\vec{v} + \vec{0} = \vec{0} + \vec{v} = \vec{v}$.

Note that we did *not* define a *zero point* or "origin" for affine space. This is an intentional omission. No point special compared to any other point. (We will eventually have to break down and define an origin in order to have a coordinate system for our points, but this is a purely representational necessity, not an intrinsic feature of affine space.)

You might ask, why make a distinction between points and vectors?[2] Although both can be represented in the same way as a list of coordinates, they represent very different concepts. For example, points would be appropriate for representing a vertex of a mesh, the center of mass of an object, the point of contact between two colliding objects. In contrast, a vector would be appropriate for representing the velocity of a moving object, the vector normal to a surface, the axis about which a rotating object is spinning. (As computer scientists the idea of different abstract objects sharing a common representation should be familiar. For example, stacks and queues are two different abstract data types, but they can both be represented as a 1-dimensional array.)

Because points and vectors are conceptually different, it is not surprising that the operations that can be applied to them are different. For example, it makes perfect sense to multiply a vector and a scalar. Geometrically, this corresponds to stretching the vector by this amount. It also makes sense to add two vectors together. This involves the usual head-to-tail rule, which you learn in linear algebra. It is not so clear, however, what it means to multiply a point by a scalar. (For example, the top of the Washington monument is a point. What would it mean to multiply this point by 2?) On the other hand, it does make sense to add a vector to a point. For example, if a vector points straight up and is three meters long, then adding this to the top of the Washington monument would naturally give you a point that is three meters above the top of the monument.

We will use the following notational conventions. Points will usually be denoted by lower-case Roman letters such as p, q, and r. Vectors will usually be denoted with lower-case Roman letters, such as u, v, and w, and often to emphasize this we will add an arrow (e.g., $\vec{u}$, $\vec{v}$, $\vec{w}$). Scalars will be represented as lower case Greek letters (e.g., α, β, γ). In our programs, scalars will be translated to Roman (e.g., a, b, c). (We will sometimes violate these conventions, however. For example, we may use c to denote the center point of a circle or r to denote the scalar radius of a circle.)

Affine Operations: The table below lists the valid combinations of scalars, points, and vectors. The formal definitions are pretty much what you would expect. Vector operations are applied in the same way that you learned in linear algebra. For example, vectors are added in the usual "tail-to-head" manner (see Fig. 5). The difference $p - q$ of two points results in a free vector directed from q to p. Point-vector addition $r + \vec{v}$ is defined to be the translation of r by displacement $\vec{v}$. Note that some operations (e.g. scalar-point multiplication, and addition of points) are explicitly not defined.

$vector \leftarrow scalar \cdot vector$, $\quad vector \leftarrow vector / scalar$	scalar-vector multiplication
$vector \leftarrow vector + vector$, $\quad vector \leftarrow vector - vector$	vector-vector addition
$vector \leftarrow point - point$	point-point difference
$point \leftarrow point + vector$, $\quad point \leftarrow point - vector$	point-vector addition

[2] Unity does not distinguish between them. The data type Vector3 is used to represent both points and vectors.

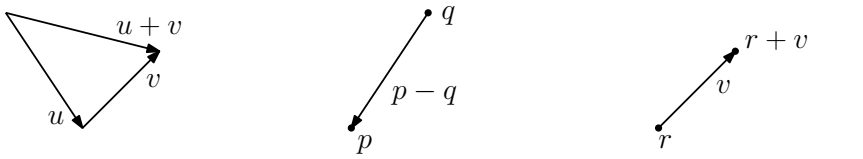


Fig. 5: Affine operations.

Affine Combinations: Although the algebra of affine geometry has been careful to disallow point addition and scalar multiplication of points, there is a particular combination of two points that we will consider legal. The operation is called an *affine combination*.

Let's say that we have two points p and q and want to compute their midpoint r, or more generally a point r that subdivides the line segment $\overline{pq}$ into the proportions α and $1-\alpha$, for some $\alpha \in [0,1]$. (The case $\alpha = 1/2$ is the case of the midpoint). This could be done by taking the vector $q-p$, scaling it by α, and then adding the result to p. That is,

$$r = p + \alpha(q-p),$$

(see Fig. 6(a)). Another way to think of this point r is as a *weighted average* of the endpoints p and q. Thinking of r in these terms, we might be tempted to rewrite the above formula in the following (technically illegal) manner:

$$r = (1-\alpha)p + \alpha q,$$

(see Fig. 6(b)). Observe that as α ranges from 0 to 1, the point r ranges along the line segment from p to q. In fact, we may allow to become negative in which case r lies to the left of p, and if $\alpha > 1$, then r lies to the right of q (see Fig. 6(c)). The special case when $0 \le \alpha \le 1$, this is called a *convex combination*.

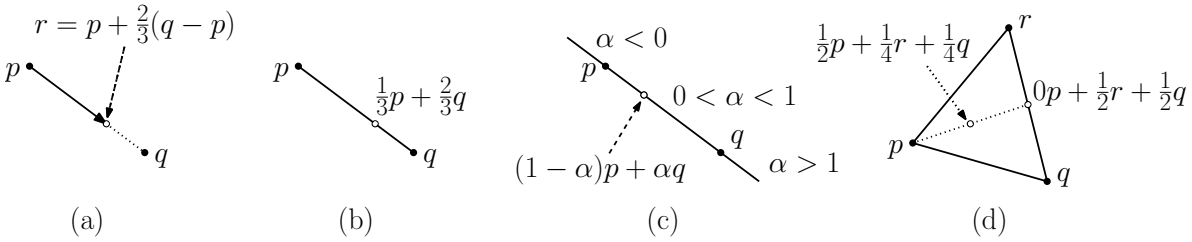


Fig. 6: Affine combinations.

In general, we define the following two operations for points in affine space.

Affine combination: Given a sequence of points $p_1, p_2, \ldots, p_n$, an affine combination is any sum of the form

$$\alpha_1 p_1 + \alpha_2 p_2 + \ldots + \alpha_n p_n,$$

where $\alpha_1, \alpha_2, \ldots, \alpha_n$ are scalars satisfying $\sum_i \alpha_i = 1$.

Convex combination: Is an affine combination, where in addition we have $\alpha_i \ge 0$ for $1 \le i \le n$.

Affine and convex combinations have a number of nice uses in graphics. For example, any three noncollinear points determine a plane. There is a 1–1 correspondence between the points on this plane and the affine combinations of these three points. Similarly, there is a 1–1 correspondence between the

points in the triangle determined by the these points and the convex combinations of the points (see Fig. 6(d)). In particular, the point $\frac{1}{3}p + \frac{1}{3}q + \frac{1}{3}r$ is the *centroid* of the triangle.

We will sometimes be sloppy, and write expressions like $\frac{1}{2}(p+q)$, which really means $\frac{1}{2}p + \frac{1}{2}q$. We will allow this sort of abuse of notation provided that it is clear that there is a legal affine combination that underlies this operation.

To see whether you understand the notation, consider the following questions. Given three points in the 3-space, what is the union of all their affine combinations? (Ans: the plane containing the 3 points.) What is the union of all their convex combinations? (Ans: The triangle defined by the three points and its interior.)

Euclidean Geometry: In affine geometry we have provided no way to talk about angles or distances. Euclidean geometry is an extension of affine geometry which includes one additional operation, called the *inner product*.

The inner product is an operator that maps two vectors to a scalar. The product of $\vec{u}$ and $\vec{v}$ is denoted commonly denoted $(\vec{u}, \vec{v})$. There are many ways of defining the inner product, but any legal definition should satisfy the following requirements

Positiveness: $(\vec{u}, \vec{u}) \geq 0$ and $(\vec{u}, \vec{u}) = 0$ if and only if $\vec{u} = \vec{0}$.

Symmetry: $(\vec{u}, \vec{v}) = (\vec{v}, \vec{u})$.

Bilinearity: $(\vec{u}, \vec{v} + \vec{w}) = (\vec{u}, \vec{v}) + (\vec{u}, \vec{w})$, and $(\vec{u}, \alpha\vec{v}) = \alpha(\vec{u}, \vec{v})$. (Notice that the symmetric forms follow by symmetry.)

See a book on linear algebra for more information. We will focus on a the most familiar inner product, called the *dot product*. To define this, we will need to get our hands dirty with coordinates. Suppose that the d-dimensional vector $\vec{u}$ is represented by the coordinate vector $(u_0, u_1, \ldots, u_{d-1})$. Then define

$$\vec{u} \cdot \vec{v} = \sum_{i=0}^{d-1} u_i v_i,$$

Note that inner (and hence dot) product is defined only for vectors, not for points.

Using the dot product we may define a number of concepts, which are not defined in regular affine geometry (see Fig. 7). Note that these concepts generalize to all dimensions.

Length: of a vector $\vec{v}$ is defined to be $\sqrt{\vec{v} \cdot \vec{v}}$, and is denoted by $\|\vec{v}\|$ (or as $|\vec{v}|$).

Normalization: Given any nonzero vector $\vec{v}$, define the *normalization* to be a vector of unit length that points in the same direction as $\vec{v}$, that is, $\vec{v}/\|\vec{v}\|$. We will denote this by $\widehat{v}$.

Distance between points: $\text{dist}(p, q) = \|p - q\|$.

Angle: between two nonzero vectors $\vec{u}$ and $\vec{v}$ (ranging from 0 to π) is

$$\text{ang}(\vec{u}, \vec{v}) \;=\; \cos^{-1}\left(\frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|}\right) \;=\; \cos^{-1}(\widehat{u} \cdot \widehat{v}).$$

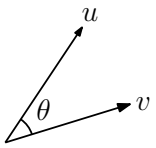
This is easy to derive from the law of cosines. Note that this does not provide us with a signed angle. We cannot tell whether $\vec{u}$ is clockwise our counterclockwise relative to $\vec{v}$. We will discuss signed angles when we consider the cross-product.

Orthogonality: $\vec{u}$ and $\vec{v}$ are *orthogonal* (or perpendicular) if $\vec{u} \cdot \vec{v} = 0$.

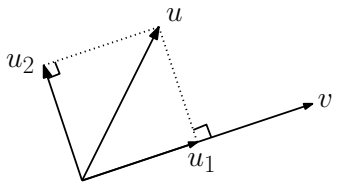
Orthogonal projection: Given a vector $\vec{u}$ and a nonzero vector $\vec{v}$, it is often convenient to decompose $\vec{u}$ into the sum of two vectors $\vec{u} = \vec{u}_1 + \vec{u}_2$, such that $\vec{u}_1$ is parallel to $\vec{v}$ and $\vec{u}_2$ is orthogonal to $\vec{v}$.

$$\vec{u}_1 \;\leftarrow\; \frac{(\vec{u} \cdot \vec{v})}{(\vec{v} \cdot \vec{v})}\vec{v}, \qquad\qquad \vec{u}_2 \;\leftarrow\; \vec{u} - \vec{u}_1.$$

(As an exercise, verify that $\vec{u}_2$ is orthogonal to $\vec{v}$.) Note that we can ignore the denominator if we know that $\vec{v}$ is already normalized to unit length. The vector $\vec{u}_1$ is called the *orthogonal projection* of $\vec{u}$ onto $\vec{v}$. If we think of $\vec{v}$ as being a normal vector to a plane, then the projection of u onto this plane is called *orthogonal complement* of u with respect to v, and is given by $\vec{u}_2 \leftarrow \vec{u} - \vec{u}_1$.



Angle between vectors



Orthogonal projection and its complement

Fig. 7: The dot product and its uses.

Doing it with Unity: Unity does not distinguish between points and vectors. Both are represented using Vector3. Unity supports many of the vector operations by overloading operators. Given vectors u, v, and w, all of type Vector3, the following operators are supported:

```
u = v + w; // vector addition
u = v - w; // vector subtraction
if (u == v || u != w) { ... } // vector comparison
u = v * 2.0f; // scalar multiplication
v = w / 2.0f; // scalar division
```

You can access the components of a Vector3 using as either using axis names, such as, u.x, u.y, and u.z, or through indexing, such as u[0], u[1], and u[2].

The Vector3 class also has the following members and static functions.

```
float x = v.magnitude; // length of v
Vector3 u = v.normalize; // unit vector in v's direction
float a = Vector3.Angle (u, v); // angle (degrees) between u and v
float b = Vector3.Dot (u, v); // dot product between u and v
Vector3 u1 = Vector3.Project (u, v); // orthog proj of u onto v
Vector3 u2 = Vector3.ProjectOnPlane (u, v); // orthogonal complement
```

Some of the Vector3 functions apply when the objects are interpreted as points. Let p and q be points declared to be of type Vector3. The function Vector3.Lerp is short for *linear interpolation*. It is essentially a two-point special case of a convex combination. (The combination parameter is assumed to lie between 0 and 1.)

```
float b = Vector3.Distance (p, q); // distance between p and q
Vector3 midpoint = Vector3.Lerp(p, q, 0.5f); // convex combination
```

Lecture 5: More on Geometry and Geometric Programming

More Geometric Programming: In this lecture we continue the discussion of basic geometric programming from the previous lecture. We will discuss coordinate systems for affine and Euclidean geometry, cross-product and orientation testing, and affine transformations.

Local and Global Frames of Reference: Last time we introduced the basic elements of affine and Euclidean geometry: points and (free) vectors. However, as of yet we have no mechanism for representing

these objects. Recall that points are to be thought of as locations in space and (free) vectors represent direction and magnitude, but are not tied down to a particular location in space. We seek a "frame of reference" from which to describe vectors and points. This is called a *coordinate frame*.

There is a *global coordinate frame* (also called the *world frame*) from which all geometric objects are described. It is convenient in geometric programming to define various *local frames* as well. For example, suppose we have a vehicle driving around a city. We might attach a local frame to this vehicle in order to describe the relative positions of objects and characters within the vehicle. The position of the vehicle itself is then described relative to the global frame. This raises the question of how to convert between the *local coordinates* used to define objects within the vehicle to their *global coordinates*.

Bases, Vectors, and Coordinates: The first question is how to represent points and vectors in affine space. We will begin by recalling how to do this in linear algebra, and generalize from there. We know from linear algebra that if we have 2-linearly independent vectors, $\vec{u}_0$ and $\vec{u}_1$ in 2-space, then we can represent *any* other vector in 2-space uniquely as a *linear combination* of these two vectors (see Fig. 8(a)):

$$\vec{v} = \alpha_0\vec{u}_0 + \alpha_1\vec{u}_1,$$

for some choice of scalars α_0, α_1.

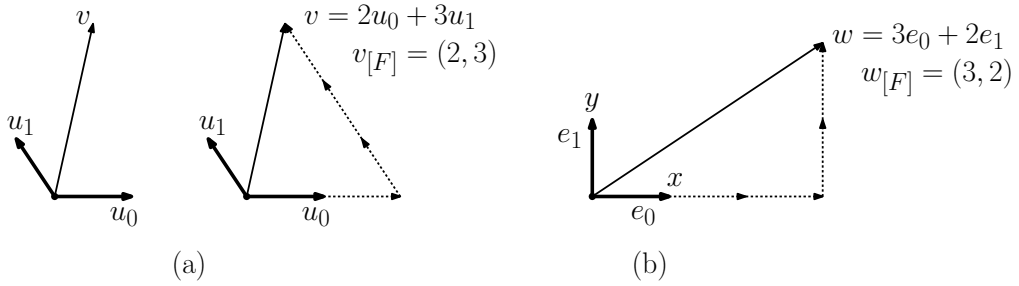


Fig. 8: Bases and linear combinations in linear algebra (a) and the standard basis (b).

Thus, given any such vectors, we can use them to represent any vector in terms of a pair of scalars (α_0, α_1). In general d linearly independent vectors in dimension d is called a *basis*. The most convenient basis to work with consists of two vectors, each of unit length, that are orthogonal to each other. Such a collection of vectors is said to be *orthonormal*. The *standard basis* consisting of the x- and y-unit vectors is an example of such a basis (see Fig. 8(b)).

Note that we are using the term "vector" in two different senses here, one as a geometric entity and the other as a sequence of numbers, given in the form of a row or column. The first is the object of interest (i.e., the abstract data type, in computer science terminology), and the latter is a representation. As is common in object oriented programming, we should "think" in terms of the abstract object, even though in our programming we will have to get dirty and work with the representation itself.

Coordinate Frames and Coordinates: Now let us turn from linear algebra to affine geometry. Again, let us consider just 2-dimensional space. To define a coordinate frame for an affine space we would like to find some way to represent any object (point or vector) as a sequence of scalars. Thus, it seems natural to generalize the notion of a basis in linear algebra to define a basis in affine space. Note that free vectors alone are not enough to define a point (since we cannot define a point by any combination of vector operations). To specify position, we will designate an *arbitrary* point, denoted O, to serve as the *origin* of our coordinate frame. Let $\vec{u}_0$ and $\vec{u}_1$ be a pair of linearly independent vectors. We already know that we can represent any vector uniquely as a linear combination of these two basis

vectors. We can represent any point p by adding a vector to O (in particular, the vector $p - O$). It follows that we can represent any point p in the following form:

$$p = \alpha_0 \vec{u}_0 + \alpha_1 \vec{u}_1 + O,$$

for some pair of scalars α_0 and α_1. This suggests the following definition.

Definition: A *coordinate frame* for a d-dimensional affine space consists of a point (which we will denote O), called the *origin* of the frame, and a set of d linearly independent *basis vectors*.

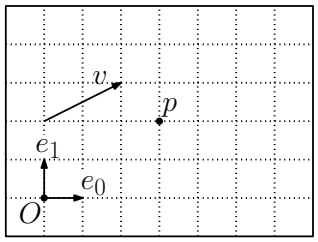
Given the above definition, we now have a convenient way to express both points and vectors. As with linear algebra, the most natural type of basis is orthonormal. Given an orthonormal basis consisting of origin O and unit vectors $\vec{e}_0$ and $\vec{e}_1$, we can express any point p and any vector $\vec{v}$ as:

$$p \;=\; \alpha_0 \cdot \vec{e}_0 + \alpha_1 \cdot \vec{e}_1 + O \qquad \text{and} \qquad \vec{v} \;=\; \beta_0 \cdot \vec{e}_0 + \beta_1 \cdot \vec{e}_1$$

for scalars α_0, α_1, β_0, and β_1.

In order to convert this into a coordinate system, let us entertain the following "notational convention." Define $1 \cdot O = O$ and $0 \cdot O = \vec{0}$ (the zero vector). Note that these two expressions are blatantly illegal by the rules of affine geometry, but this convention makes it possible to express the above equations in a common (homogeneous) form (see Fig. 9):

$$p \;=\; \alpha_0 \cdot \vec{e}_0 + \alpha_1 \cdot \vec{e}_1 + 1 \cdot O \qquad \text{and} \qquad \vec{v} \;=\; \beta_0 \cdot \vec{e}_0 + \beta_1 \cdot \vec{e}_1 + 0 \cdot O.$$



$$p = 3 \cdot \vec{e}_0 + 2 \cdot \vec{e}_1 + 1 \cdot O$$
$$\Rightarrow p_{[F]} = (3, 2, 1)$$

$$v = 2 \cdot \vec{e}_0 + 1 \cdot \vec{e}_1 + 0 \cdot O$$
$$\Rightarrow v_{[F]} = (2, 1, 0)$$

Fig. 9: Coordinate frames and (affine) homogeneous coordinates.

This suggests a nice method for expressing both points and vectors using a common notation. For the given coordinate frame $F = (\vec{e}_0, \vec{e}_1, O)$ we can express the point p and the vector $\vec{v}$ as

$$p_{[F]} \;=\; (\alpha_0, \alpha_1, 1) \qquad \text{and} \qquad \vec{v}_{[F]} \;=\; (\beta_0, \beta_1, 0)$$

(see Fig. 9).

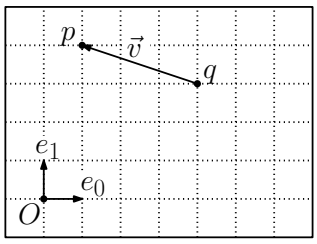
These are called *(affine) homogeneous coordinates*. In summary, to represent points and vectors in d-space, we will use coordinate vectors of length $d + 1$. Points have a last coordinate of 1, and vectors have a last coordinate of 0. (Some conventions place the homogenizing coordinate first rather than last. There are actually good reasons for doing this. But we will stick with standard engineering conventions and place it last.)

Properties of homogeneous coordinates: The choice of appending a 1 for points and a 0 for vectors may seem to be a rather arbitrary choice. Why not just reverse them or use some other scalar values? The reason is that this particular choice has a number of nice properties with respect to geometric operations.

For example, consider two points p and q whose coordinate representations relative to some frame F are $p_{[F]} = (1, 4, 1)$ and $q_{[F]} = (4, 3, 1)$, respectively (see Fig. 10). Consider the vector

$$\vec{v} = p - q.$$

If we apply the difference rule that we defined last time for points, and then convert this vector into it coordinates relative to frame F, we find that $\vec{v}_{[F]} = (-3, 1, 0)$. Thus, to compute the coordinates of $p - q$ we simply take the component-wise difference of the coordinate vectors for p and q. The 1-components nicely cancel out, to give a vector result.



$$p_{[F]} = (1, 4, 1)$$
$$q_{[F]} = (4, 3, 1)$$
$$\vec{v}_{[F]} = (1 - 4, 4 - 3, 1 - 1) = (-3, 1, 0)$$

Fig. 10: Coordinate arithmetic.

In general, a nice feature of this representation is the last coordinate behaves exactly as it should. Let u and v be either points or vectors. After a number of operations of the forms $u + v$ or $u - v$ or αu (when applied to the coordinates) we have:

- If the last coordinate is 1, then the result is a *point*.
- If the last coordinate is 0, then the result is a *vector*.
- Otherwise, this is not a legal affine operation.

This fact can be proved rigorously, but we won't worry about doing so.

Cross Product: The cross product is an important vector operation in 3-space. You are given two vectors and you want to find a third vector that is orthogonal to these two. This is handy in constructing coordinate frames with orthogonal bases. There is a nice operator in 3-space, which does this for us, called the *cross product*.

The cross product is usually defined in standard linear 3-space (since it applies to vectors, not points). So we will ignore the homogeneous coordinate here. Given two vectors in 3-space, $\vec{u}$ and $\vec{v}$, their *cross product* is defined as follows (see Fig. 11(a)):

$$\vec{u} \times \vec{v} = \begin{pmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{pmatrix}.$$

A nice mnemonic device for remembering this formula, is to express it in terms of the following symbolic determinant:

$$\vec{u} \times \vec{v} = \begin{vmatrix} \vec{e}_x & \vec{e}_y & \vec{e}_z \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix}.$$

Here $\vec{e}_x$, $\vec{e}_y$, and $\vec{e}_z$ are the three coordinate unit vectors for the standard basis. Note that the cross product is only defined for a pair of free vectors and only in 3-space. Furthermore, we ignore the homogeneous coordinate here. The cross product has the following important properties:

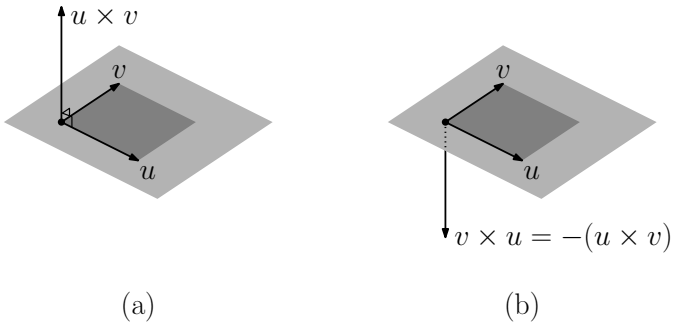


Fig. 11: Cross product.

Skew symmetric: $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$ (see Fig. 12(b)). It follows immediately that $\vec{u} \times \vec{u} = 0$ (since it is equal to its own negation).

Nonassociative: Unlike most other products that arise in algebra, the cross product is *not* associative. That is

$$(\vec{u} \times \vec{v}) \times \vec{w} \neq \vec{u} \times (\vec{v} \times \vec{w}).$$

Bilinear: The cross product is linear in both arguments. For example:

$$\begin{aligned} \vec{u} \times (\alpha\vec{v}) &= \alpha(\vec{u} \times \vec{v}), \\ \vec{u} \times (\vec{v} + \vec{w}) &= (\vec{u} \times \vec{v}) + (\vec{u} \times \vec{w}). \end{aligned}$$

Perpendicular: If $\vec{u}$ and $\vec{v}$ are not linearly dependent, then $\vec{u} \times \vec{v}$ is perpendicular to $\vec{u}$ and $\vec{v}$, and is directed according the right-hand rule.

Angle and Area: The length of the cross product vector is related to the lengths of and angle between the vectors. In particular:

$$|\vec{u} \times \vec{v}| = |u||v| \sin\theta,$$

where θ is the angle between $\vec{u}$ and $\vec{v}$. The cross product is usually not used for computing angles because the dot product can be used to compute the cosine of the angle (in any dimension) and it can be computed more efficiently. This length is also equal to the area of the parallelogram whose sides are given by $\vec{u}$ and $\vec{v}$. This is often useful.

The cross product is commonly used in computer graphics for generating coordinate frames. Given two basis vectors for a frame, it is useful to generate a third vector that is orthogonal to the first two. The cross product does exactly this. It is also useful for generating surface normals. Given two tangent vectors for a surface, the cross product generate a vector that is normal to the surface.

Orientation: Given two real numbers p and q, there are three possible ways they may be ordered: $p < q$, $p = q$, or $p > q$. We may define an orientation function, which takes on the values $+1$, 0, or -1 in each of these cases. That is, $\mathrm{Or}_1(p, q) = \mathrm{sign}(q - p)$, where $\mathrm{sign}(x)$ is either -1, 0, or $+1$ depending on whether x is negative, zero, or positive, respectively. An interesting question is whether it is possible to extend the notion of order to higher dimensions.

The answer is yes, but rather than comparing two points, in general we can define the orientation of $d+1$ points in d-space. We define the *orientation* to be the sign of the determinant consisting of their homogeneous coordinates (with the homogenizing coordinate given first). For example, in the plane and 3-space the orientation of three points p, q, r is defined to be

$$\mathrm{Or}_2(p, q, r) = \mathrm{sign}\ \det \begin{pmatrix} 1 & 1 & 1 \\ p_x & q_x & r_x \\ p_y & q_y & r_y \end{pmatrix}, \qquad \mathrm{Or}_3(p, q, r, s) = \mathrm{sign}\ \det \begin{pmatrix} 1 & 1 & 1 & 1 \\ p_x & q_x & r_x & s_x \\ p_y & q_y & r_y & s_y \\ p_z & q_z & r_z & s_z \end{pmatrix}.$$

What does orientation mean intuitively? The orientation of three points in the plane is +1 if the triangle PQR is oriented counter-clockwise, -1 if clockwise, and 0 if all three points are collinear (see Fig. 12). In 3-space, a positive orientation means that the points follow a right-handed screw, if you visit the points in the order $PQRS$. A negative orientation means a left-handed screw and zero orientation means that the points are coplanar. Note that the order of the arguments is significant. The orientation of (p, q, r) is the negation of the orientation of (p, r, q). As with determinants, the swap of any two elements reverses the sign of the orientation.

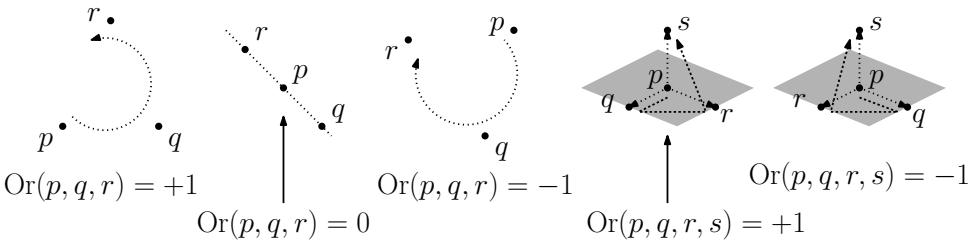


Fig. 12: Orientations in 2 and 3 dimensions.

You might ask why put the homogeneous coordinate first? The answer a mathematician would give you is that is really where it should be in the first place. If you put it last, then positive oriented things are "right-handed" in even dimensions and "left-handed" in odd dimensions. By putting it first, positively oriented things are always right-handed in orientation, which is more elegant. Putting the homogeneous coordinate last seems to be a convention that arose in engineering, and was adopted later by graphics people.

The value of the determinant itself is the area of the parallelogram defined by the vectors $q - p$ and $r - p$, and thus this determinant is also handy for computing areas and volumes. Later we will discuss other methods.

Orientation testing is a very useful tool, but it is (surprisingly) not very widely known in the areas of computer game programming and computer graphics. For example, suppose that we have a bullet path, represented by a line segment $\overline{pq}$. We want to know whether the linear extension of this segment intersects a target triangle, $\triangle abc$. We can determine this using three orientation tests. To see the connection, consider the three directed edges of the triangle $\overrightarrow{ab}$ $\overrightarrow{bc}$ and $\overrightarrow{ca}$. Suppose that we place an observer along each of these edges, facing the direction of the edge. If the line passes through the triangle, then all three observers will see the directed line $\overrightarrow{pq}$ passing in the same direction relative to their edge (see Fig. 13). (This might take a bit of time to convince yourself of this. To make it easier, imagine that the triangle is on the floor with a, b, and c given in counterclockwise order, and the line is vertical with p below the floor and q above. The line hits the triangle if and only if all three observers, when facing the direction of their respective edges, see the line on their left. If we reverse the roles of p and q, they will all see the line as being on their right. In any case, they all agree.)

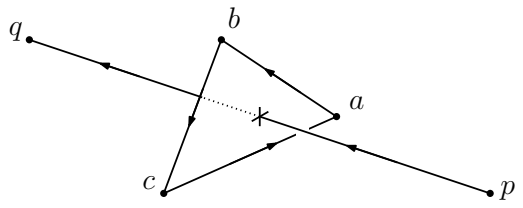


Fig. 13: Using orientation testing to determine line-triangle intersection.

It follows that the line passes through the triangle if and only if

$$\mathrm{Or}_3(p, q, a, b) \;=\; \mathrm{Or}_3(p, q, b, c) \;=\; \mathrm{Or}_3(p, q, c, d).$$

(By the way, this tests only whether the infinite line intersects the triangle. To determine whether the segment intersects the triangle, we should also check that p and q lie on opposite sides of the triangle. Can you see how to do this with two additional orientation tests?)

Lecture 6: Affine Transformations and Rotations

Affine Transformations: So far we have been stepping through the basic elements of geometric programming. We have discussed points, vectors, and their operations, and coordinate frames and how to change the representation of points and vectors from one frame to another. Our next topic involves how to map points from one place to another. Suppose you want to draw an animation of a spinning ball. How would you define the function that maps each point on the ball to its position rotated through some given angle?

We will consider a limited, but interesting class of transformations, called *affine transformations*. These include (among others) the following transformations of space: translations, rotations, uniform and nonuniform scalings (stretching the axes by some constant scale factor), reflections (flipping objects about a line) and shearings (which deform squares into parallelograms). They are illustrated in Fig. 14.

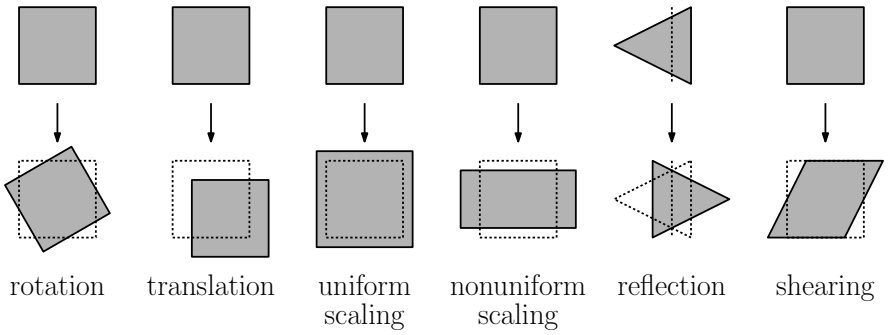


Fig. 14: Examples of affine transformations.

These transformations all have a number of things in common. For example, they all map lines to lines. Note that some (translation, rotation, reflection) preserve the lengths of line segments and the angles between segments. These are called *rigid transformations*. Others (like uniform scaling) preserve angles but not lengths. Still others (like nonuniform scaling and shearing) do not preserve angles or lengths.

Formal Definition: Formally, an *affine transformation* is a mapping from one affine space to another (which may be, and in fact usually is, the same space) that preserves affine combinations. For example, this implies that given any affine transformation T and two points p and q, and any scalar α,

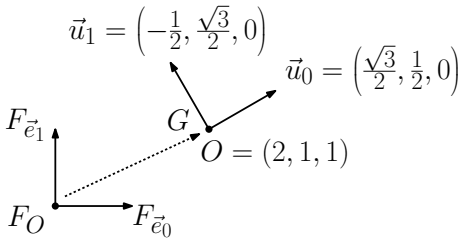
$$r \;=\; (1-\alpha)p + \alpha q \qquad \Rightarrow \qquad T(r) \;=\; (1-\alpha)T(p) + \alpha T(q).$$

For example, if r is the midpoint of segment $\overline{pq}$, then $T(r)$ is the midpoint of the transformed line segment $\overline{T(p)T(q)}$.

Matrix Representations of Affine Transformations: The above definition is rather abstract. It is possible to present any affine transformation T in d-dimensional space as a $(d+1) \times (d+1)$ matrix. For

example, suppose that we have a d-dimensional frame F consisting of an origin point $F.O$ and basis vectors $F.\vec{e}_0$ through $F.\vec{e}_{d-1}$. To express T in the form of a matrix, we determine where each of these frame components is mapped, and then generate a matrix whose first d columns are the images of the basis vectors and whose last component is the image of the origin point. (It follows, therefore, that the last row of such a matrix must be $(0, \ldots, 0, 1)$, because the basis vectors must map to vectors, which must end in 0 according to the rules of affine homogeneous coordinates, and the origin point maps to a point, which must end in 1 by these same rules.)

For example, consider the affine transformation (in 2-dimensional space) shown in Fig. 15, which rotates by 30° degrees about the origin and translates 2 units to the right and 1 unit up. This transformation maps the origin $F.O$ to the point O with homogeneous coordinates $(2, 1, 1)$, the x-axis is mapped to the vector $\vec{u}_0 = (\cos 30°, \sin 30°, 0) = (\sqrt{3}/2, 1/2, 0)$, and the y-axis is mapped to $(-\sin 30°, \cos 30°, 0) = (-1/2, \sqrt{3}/2, 0)$. By forming a matrix whose columns are consist of $\vec{u}_0$, $\vec{u}_1$, and O, as shown in the figure.



$$\begin{bmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} & 2 \\ \frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Fig. 15: Generating a homogeneous matrix for an affine transformation.

Here are a number of concrete examples of how this applies to various transformations. Rather than considering this in the context of 2-dimensional transformations, let's consider it in the more general setting of 3-dimensional transformations. The two dimensional cases can be extracted by just ignoring the rows and columns for the z-coordinates.

Translation: Translation by a fixed vector $\vec{v}$ maps any point p to $p + \vec{v}$. Note that, since free vectors have no position in space, they are not altered by translation (see Fig. 16(a)).

Suppose that relative to the standard frame, $v[F] = (\alpha_x, \alpha_y, \alpha_z, 0)$ are the homogeneous coordinates of $\vec{v}$. The three unit vectors are unaffected by translation, and the origin is mapped to $O + \vec{v}$, whose homogeneous coordinates are $(\alpha_x, \alpha_y, \alpha_z, 1)$. Thus, by the rule given earlier, the homogeneous matrix representation for this translation transformation is

$$T(\vec{v}) = \begin{pmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Scaling: *Uniform scaling* is a transformation which is performed relative to some central fixed point. We will assume that this point is the origin of the standard coordinate frame. (We will leave the general case of scaling about an arbitrary point in space as an exercise.) Given a scalar β, this transformation maps the object (point or vector) with coordinates $(\alpha_x, \alpha_y, \alpha_z, \alpha_w)$ to $(\beta\alpha_x, \beta\alpha_y, \beta\alpha_z, \alpha_w)$ (see Fig. 16(b)).

In general, it is possible to specify separate scaling factors for each of the axes. This is called *nonuniform scaling*. The unit vectors are each stretched by the corresponding scaling factor, and

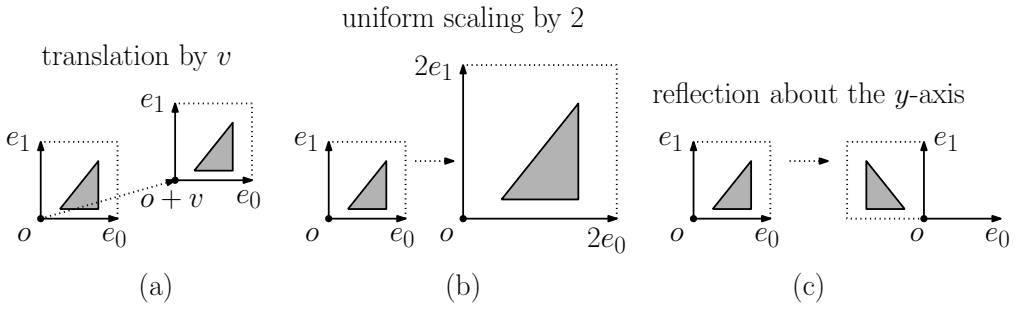


Fig. 16: Derivation of transformation matrices.

the origin is unmoved. Thus, the transformation matrix has the following form:

$$S(\beta_x, \beta_y, \beta_z) = \begin{pmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Observe that both points and vectors are altered by scaling.

Reflection: In its most general form, a reflection in the plane is given a line and maps points by flipping the plane about this line. A reflection in 3-space is given a plane, and flips points in space about this plane. In this case, reflection is just a special case of scaling, but where the scale factor is negative. A common simple version of this is when the plane about which the reflection is performed is one of the coordinate planes (corresponding to $x = 0$, $y = 0$, or $z = 0$).

For example, to reflect points about the yz-coordinate plane (that is, the plane $x = 0$), we can scale the x-coordinate by -1 (see Fig. 16(c)). Using the scaling matrix above, we have the following transformation matrix:

$$F_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The cases for the other two coordinate frames are similar. Reflection about an arbitrary line (in 2-space) or a plane (in 3-space) is left as an exercise.

Rotation: In its most general form, rotation is defined to take place about some fixed point, and around some fixed vector in space. We will consider the simplest case where the fixed point is the origin of the coordinate frame, and the vector is one of the coordinate axes. There are three basic rotations: about the x, y and z-axes. In each case the rotation is counterclockwise through an angle θ (given in radians). The rotation is assumed to be in accordance with a right-hand rule: if your right thumb is aligned with the axes of rotation, then positive rotation is indicated by the direction in which the fingers of this hand are pointing. To produce a clockwise rotation, simply negate the angle involved.

Consider a rotation about the z-axis. The z-unit vector and origin are unchanged. The x-unit vector is mapped to $(\cos\theta, \sin\theta, 0, 0)$, and the y-unit vector is mapped to $(-\sin\theta, \cos\theta, 0, 0)$ (see Fig. 17(a)). Thus the rotation matrix is:

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Observe that both points and vectors are altered by rotation. For the other two axes we have:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \qquad R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

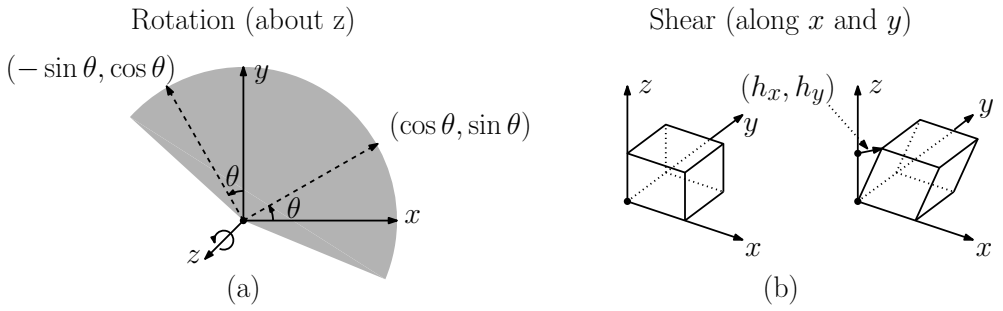


Fig. 17: Rotation and shearing.

If (as with Unity) the coordinate frame is left-handed, then the directions of all the rotations are reversed as well (clockwise, rather than counter-clockwise). Rotations about the coordinate axes are often called *Euler angles*. Rotations can generally be performed around any vector, called the *axis of rotation*, but the resulting transformation matrix is significantly more complex than the above examples.

Shearing: (Optional) A shearing transformation is perhaps the hardest of the group to visualize. Think of a shear as a transformation that maps a square into a parallelogram by sliding one side parallel to itself while keeping the opposite side fixed. In 3-dimensional space, it maps a cube into a parallelepiped by sliding one face parallel while keeping the opposite face fixed (see Fig. 17(b)). We will consider the simplest form, in which we start with a unit cube whose lower left corner coincides with the origin. Consider one of the axes, say the z-axis. The face of the cube that lies on the xy-coordinate plane does not move. The face that lies on the plane $z = 1$, is translated by a vector (h_x, h_y). In general, a point $p = (p_x, p_y, p_z, 1)$ is translated by the vector $p_z(h_x, h_y, 0, 0)$. This vector is orthogonal to the z-axis, and its length is proportional to the z-coordinate of p. This is called an *xy-shear*. (The yz- and xz-shears are defined analogously.)

Under the xy-shear, the origin and x- and y-unit vectors are unchanged. The z-unit vector is mapped to $(h_x, h_y, 1, 0)$. Thus the matrix for this transformation is:

$$H_{xy}(h_x, h_y) = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Shears involving any other pairs of axes are defined analogously.

$$H_{yz}(h_y, h_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ h_y & 1 & 0 & 0 \\ h_z & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad H_{zx}(h_z, h_x) = \begin{pmatrix} 1 & h_x & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & h_z & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Transformations in Unity: Recall that all game objects in Unity (in particular, Monobehaviour objects) are associated with a member called transform, which is of type Transform. This object controls the

position and orientation of the object. If the object is *not* being controlled by the physics engine (that is, if it is kinematic), then you can control its movement through your scripts. (Otherwise, you should just let the physics engine do its job.)

Any Transform object supports the following operations for the two most common rigid transformations, translation and rotation:

- void Translate(Vector3 translation, Space relativeTo = Space.Self):

  This performs translation of the current object by the vector translation. The second argument specifies the coordinate frame relative to which the rotation is performed. By default, it is with respect to the object's own coordinate frame.

  For example, if in your update method you wanted to translate the current object forward by some given linear speed (in units per second), you could use

  ```
  transform.Translate(Vector3.forward * speed * Time.deltaTime);
  ```

- void Rotate(Vector3 eulerAngles, Space relativeTo = Space.Self):

  This performs an Euler-angle based rotation in degrees (see below). Specifically, it rotates by eulerAngles.z degrees around the z- axis, eulerAngles.x degrees around the x-axis, and eulerAngles.y degrees around the y-axis (in that order). (Given Unity's coordinate system, this means roll, then pitch, then yaw.)

  The second argument specifies the coordinate frame relative to which the rotation is performed. By default, it is with respect to the object's own coordinate frame. (I believe that, because of Unity's convention of using left-handed coordinate frames, a positive rotation corresponds to a *clockwise* rotation, but I am not entirely sure about this.)

  For example, if in your update method you wanted to rotate the current object by some given angular speed (in degrees per second) about its own vertical axis, you could use

  ```
  transform.Rotate(Vector3.up, speed * Time.deltaTime);
  ```

Rotation and Orientation in 3-Space: One of the trickier problems 3-d geometry is that of parameterizing rotations and the orientation of frames. We have introduced the notion of orientation before (e.g., clockwise or counterclockwise). Here we mean the term in a somewhat different sense, as a directional position in space. Describing and managing rotations in 3-space is a somewhat more difficult task (at least conceptually), compared with the relative simplicity of rotations in the plane. We will explore two methods for dealing with rotation, *Euler angles* and *quaternions*.

Euler Angles: Leonard Euler was a famous mathematician who lived in the 18th century. He proved many important theorems in geometry, algebra, and number theory, and he is credited as the inventor of graph theory. Among his many theorems is one that states that the composition any number of rotations in three-space can be expressed as a single rotation in 3-space about an appropriately chosen vector. Euler also showed that any rotation in 3-space could be broken down into exactly three rotations, one about each of the coordinate axes.

Suppose that you are a pilot, such that the x-axis points to your left, the y-axis points ahead of you, and the z-axis points up (see Fig. 18). (This is the coordinate frame that I prefer, which is also used by the Unreal engine. Note that Unity swaps the z and y axes.) Then a rotation about the x-axis, denoted by ϕ, is called the *pitch*. A rotation about the y-axis, denoted by θ, is called *roll*. A rotation about the z-axis, denoted by ψ, is called *yaw*. Euler's theorem states that any position in space can be expressed by composing three such rotations, for an appropriate choice of (ϕ, θ, ψ).

The order in which the rotations are performed is significant. In Unity (using the command transform.Rotate(x, y, z)), the order is the z-axis first, x-axis second, and y-axis third. Recalling that Unity switches the rolls of the z and y axes relative to the above figure, this means that it performs the operations in the order roll, then pitch, then yaw.

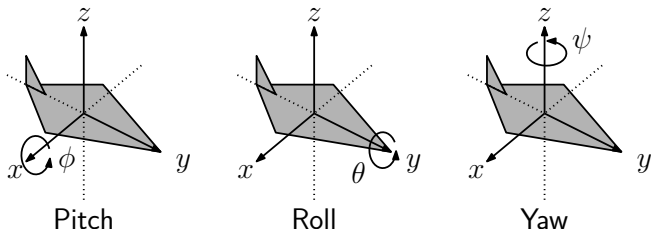


Fig. 18: Euler angles: pitch, roll, and yaw.

Shortcomings of Euler angles: There are some problems with Euler angles. One issue is the fact that this representation depends on the choice of coordinate system. In the plane, a 30-degree rotation is the same, no matter what direction the axes are pointing (as long as they are orthonormal and right-handed). However, the result of an Euler-angle rotation depends very much on the choice of the coordinate frame and on the order in which the axes are named. (Later, we will see that quaternions do provide such an intrinsic system.)

Another problem with Euler angles is called *gimbal lock*. Whenever we rotate about one axis, it is possible that we could bring the other two axes into alignment with each other. (This happens, for example if we rotate x by $90°$.) This causes problems because the other two axes no longer rotate independently of each other, and we effectively lose one degree of freedom. Gimbal lock as induced by one ordering of the axes can be avoided by changing the order in which the rotations are performed. But, this is rather messy, and it would be nice to have a system that is free of this problem.

Quaternions: We will now delve into a subject, which at first may seem quite unrelated. But keep the above expression in mind, since it will reappear in most surprising way. This story begins in the early 19th century, when the great mathematician William Rowan Hamilton was searching for a generalization of the complex number system.

Imaginary numbers can be thought of as linear combinations of two basis elements, 1 and i, which satisfy the multiplication rules $1^2 = 1$, $i^2 = -1$ and $1 \cdot i = i \cdot 1 = i$. (The interpretation of $i = \sqrt{-1}$ arises from the second rule.) A complex number $a + bi$ can be thought of as a vector in 2-dimensional space (a, b). Two important concepts with complex numbers are the *modulus*, which is defined to be $\sqrt{a^2 + b^2}$, and the *conjugate*, which is defined to be $(a, -b)$. In vector terms, the modulus is just the length of the vector and the conjugate is just a vertical reflection about the x-axis. If a complex number is of modulus 1, then it can be expressed as $(\cos\theta, \sin\theta)$. Thus, there is a connection between complex numbers and 2-dimensional rotations. Also, observe that, given such a unit modulus complex number, its conjugate is $(\cos\theta, -\sin\theta) = (\cos(-\theta), \sin(-\theta))$. Thus, taking the conjugate is something like negating the associated angle.

Hamilton was wondering whether this idea could be extended to three dimensional space. You might reason that, to go from 2D to 3D, you need to replace the single imaginary quantity i with two imaginary quantities, say i and j. Unfortunately, this this idea does not work. After many failed attempts, Hamilton finally came up with the idea of, rather than using two imaginaries, instead using three imaginaries i, j, and k, which behave as follows:

$$i^2 = j^2 = k^2 = ijk = -1 \qquad ij = k,\ jk = i,\ ki = j.$$

Combining these, it follows that $ji = -k$, $kj = -i$ and $ik = -j$. The skew symmetry of multiplication (e.g., $ij = -ji$) was actually a major leap, since multiplication systems up to that time had been commutative.)

Hamilton defined a *quaternion* to be a generalized complex number of the form

$$\mathbf{q} = q_0 + q_1 i + q_2 j + q_3 k.$$

Thus, a quaternion can be viewed as a 4-dimensional vector $\mathbf{q} = (q_0, q_1, q_2, q_3)$. The first quantity is a scalar, and the last three define a 3-dimensional vector, and so it is a bit more intuitive to express this as $\mathbf{q} = (s, u)$, where $s = q_0$ is a scalar and $u = (q_1, q_2, q_3)$ is a vector in 3-space. We can define the same concepts as we did with complex numbers:

Conjugate: $\mathbf{q}^* = (s, -u)$

Modulus: $|\mathbf{q}| = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} = \sqrt{s^2 + (u \cdot u)}$

Unit Quaternion: $\mathbf{q}$ is said to be a unit quaternion if $|\mathbf{q}| = 1$

Quaternion Multiplication: Consider two quaternions $\mathbf{q} = (s, u)$ and $\mathbf{p} = (t, v)$:

$$\begin{aligned} \mathbf{q} &= (s, u) = s + u_x i + u_y j + u_z k \\ \mathbf{p} &= (t, v) = t + v_x i + v_y j + v_z k. \end{aligned}$$

If we multiply these two together, we'll get lots of cross-product terms, such as $(u_x i)(v_y j)$, but we can simplify these by using Hamilton's rules. That is, $(u_x i)(v_y j) = u_x v_h (ij) = u_x v_h k$. If we do this, simplify, and collect common terms, we get a very messy formula involving 16 different terms. (The derivation is left as an exercise.) The formula can be expressed somewhat succinctly in the following form:

$$\mathbf{qp} = (st - (u \cdot v), sv + tu + u \times v).$$

Note that the above expression is in the quaternion scalar-vector form. The first term $st - (u \cdot v)$ evaluates to a scalar (recalling that the dot product returns a scalar), and the second term $(sv + tu + u \times v)$ is a sum of three vectors, and so is a vector. It can be shown that quaternion multiplication is associative, but not commutative.

Quaternion Multiplication and 3-d Rotation: Before considering rotations, we first define a *pure quaternion* to be one with a 0 scalar component

$$\mathbf{p} = (0, v).$$

Any quaternion of nonzero magnitude has a multiplicative *inverse*, which is defined to be

$$\mathbf{q}^{-1} = \frac{1}{|\mathbf{q}|^2}\mathbf{q}^*.$$

(To see why this works, try multiplying $\mathbf{q}\mathbf{q}^{-1}$, and see what you get.) Observe that if $\mathbf{q}$ is a unit quaternion, then it follows that $\mathbf{q}^{-1} = \mathbf{q}^*$.

As you might have guessed, our objective will be to show that there is a relation between rotating vectors and multiplying quaternions. In order apply this insight, we need to first show how to represent rotations as quaternions and 3-dimensional vectors as quaternions. After a bit of experimentation, the following does the trick:

Vector: Given a vector $v = (v_x, v_y, v_z)$ to be rotated, we will represent it by the pure quaternion $(0, v)$.

Rotation: To represent a rotation by angle θ about a unit vector u, you might think, we'll use the scalar part to represent θ and the vector part to represent u. Unfortunately, this doesn't quite work. After a bit of experimentation, you will discover that the right way to encode this rotation is with the quaternion $\mathbf{q} = (\cos(\theta/2), (\sin(\theta/2))u)$. (You might wonder, why we do we use $\theta/2$, rather than θ. The reason, as we shall see below, is that "this is what works.")

Rotation Operator: Given a vector v represented by the quaternion $\mathbf{p} = (0, v)$ and a rotation represented by a unit quaternion $\mathbf{q}$, we define the *rotation operator* to be:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{-1} \;=\; \mathbf{q}\mathbf{p}\mathbf{q}^{*}.$$

(The last equality results from the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$, if $\mathbf{q}$ is a unit quaternion). We claim that the result of this operation will always be a pure quaternion, and so it is possible to interpret the result as a vector. In particular, this vector will be the result of applying the rotation $\mathbf{q}$ to v.

We will give a formal justification of this later, but for now, let's consider what this gives us. Let us apply the above quaternion multiplication rule and use the fact that $\mathbf{q}^{-1} = \mathbf{q}^{*}$ for a unit quaternion $\mathbf{q} = (s, u)$. Letting $\mathbf{p} = (0, v)$ denote the object to be rotated and expanding/simplifying we obtain:

$$R_{\mathbf{q}}(\mathbf{p}) \;=\; (0,\ (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)). \tag{1}$$

(We leave the derivation as an exercise, but a few nontrivial facts regarding dot products and cross products need to applied.) It is not obvious that this has anything to do with rotation, but later we will show that this corresponds exactly to rotating v about the axis u by θ degrees.

Unity supports an object called Quaternion that encapsulates a quaternion. You can generate a quaternion that performs a rotation by x degrees about a given 3-dimensional vector $\vec{u}$ using the command Quaternion.AngleAxis(x, u). For example, the following command sets the current object's rotation to a rotation by a 30° about the vertical axis.

```
transform.rotation = Quaternion.AngleAxis(30, Vector.up);
```

Example: Consider the 3-d "roll" rotation shown in Fig. 19. This rotation can be achieved by performing a rotation about the y-axis by $\theta = 90$ degrees. Thus $\theta = \pi/2$, and the axis of rotation is $\widehat{u} = (0, 1, 0)$, and so we have $s = \cos(\theta/2) = 1/\sqrt{2}$ and $u = (\sin(\theta/2))\widehat{u} = (0, 1/\sqrt{2}, 0)$, and hence

$$\mathbf{q} \;=\; (\cos(\theta/2), (\sin(\theta/2))u) \;=\; \left(\cos\left(\frac{\pi}{4}\right), \sin\left(\frac{\pi}{4}\right)(0,1,0)\right) \;=\; \left(\frac{1}{\sqrt{2}}, \left(0, \frac{1}{\sqrt{2}}, 0\right)\right).$$

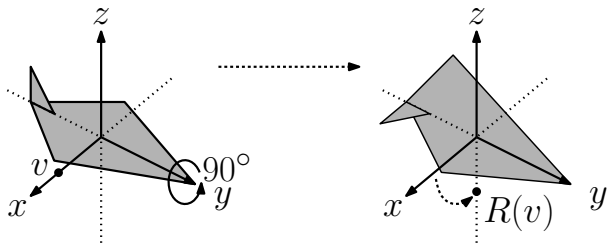


Fig. 19: Rotation example.

Let us consider how the x-unit vector $v = (1, 0, 0)$ is transformed under this rotation. To reduce this to a quaternion operation, we encode v as a pure quaternion $\mathbf{p} = (0, v) = (0, (1, 0, 0))$. Observe that

$$s^2 - (u \cdot u) = \frac{1}{2} - \frac{1}{2} = 0, \qquad (u \cdot v) = 0, \qquad \text{and} \qquad (u \times v) = \left(0, 0, \frac{-1}{\sqrt{2}}\right).$$

By applying the rotation operator, by Eq. (1), we have

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= (0,\ (s^2 - (u \cdot u))v + 2u(u \cdot v) + 2s(u \times v)) \\ &= (0,\ 0v + 2u0 + 2s(0, 0, -1/\sqrt{2})) \\ &= (0,\ \vec{0} + \vec{0} + (2/\sqrt{2})(0, 0, -1/\sqrt{2})) \\ &= (0, (0, 0, -1)). \end{aligned}$$

Interpreting $\mathbf{p}$ as a vector $(0, 0, -1)$, we see that, as expected, quaternion rotation rotates the vector $v = (1, 0, 0)$ by 90° to $(0, 0, -1)$ (see Fig. 19).

Why Quaternions Work: (Optional) In order to understand why the above quaternion operation implements rotation, we begin with the concept of *angular displacement*, which involves rotating a given vector v about a given rotation axis u (any unit vector) by a certain number of degrees θ.

Let $R(v)$ denote this rotated vector (see Fig. 20(a)). In order to derive this, we begin by decomposing v as the sum of its components that are parallel to and orthogonal to u, respectively.

$$v_\parallel \;=\; (u \cdot v)u \qquad v_\perp \;=\; v - v_\parallel \;=\; v - (u \cdot v)u.$$

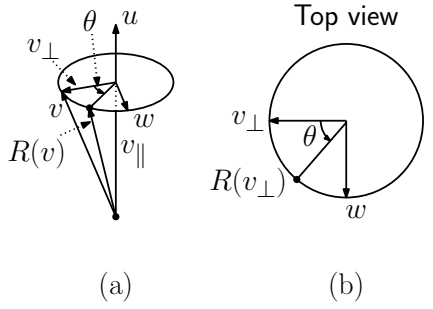


Fig. 20: Angular displacement.

Note that $v_\parallel$ is unaffected by the rotation, but $v_\perp$ is rotated to a new position $R(v_\perp)$. To determine this rotated position, we will first construct a vector that is orthogonal to $v_\perp$ lying in the plane of rotation.

$$w \;=\; u \times v_\perp \;=\; u \times (v - v_\parallel) \;=\; (u \times v) - (u \times v_\parallel) \;=\; u \times v.$$

The last step follows from the fact that u and $v_\parallel$ are parallel, and so the cross product is zero. Clearly w is orthogonal to both $v_\perp$ and u. Furthermore, because $v_\perp$ is orthogonal to the unit vector u, it follows from basic properties of the cross product that w is the same length as $v_\perp$.

Now, consider the plane spanned by $v_\perp$ and w (see Fig. 20(b)). We have

$$R(v_\perp) \;=\; (\cos\theta)v_\perp + (\sin\theta)w.$$

From this and the fact that $R(v_\parallel) = v_\parallel$, we have

$$\begin{aligned} R(v) &= R(v_\parallel) + R(v_\perp) \;=\; v_\parallel + (\cos\theta)v_\perp + (\sin\theta)w \\ &= (u \cdot v)u + (\cos\theta)(v - (u \cdot v)u) + (\sin\theta)w \\ &= (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \end{aligned}$$

In summary, we have the following formula expressing the effect of the rotation of vector v by angle θ about a rotation axis u:

$$R(v) \;=\; (\cos\theta)v + (1 - \cos\theta)u(u \cdot v) + (\sin\theta)(u \times v). \tag{2}$$

This expression is the image of v under the rotation. Notice that, unlike Euler angles, this is expressed entirely in terms of intrinsic geometric functions (such as dot and cross product), which do not depend on the choice of coordinate frame. This is a major advantage of this approach over Euler angles.

Now that we know how to express rotation in terms of vector operations, let's see how this relates to the quaternion rotation operation. Let us see if we can express this in a more suggestive form. Since $\mathbf{q}$ is of unit magnitude, we can express it as

$$\mathbf{q} = \left(\cos\frac{\theta}{2}, \left(\sin\frac{\theta}{2}\right)u\right), \qquad \text{where } \|u\| = 1.$$

Plugging this into Eq. (1) and applying some standard trigonometric identities, we obtain

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{p}) &= \left(0, \left(\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2}\right)v + 2\left(\sin^2\frac{\theta}{2}\right)u(u\cdot v) + 2\cos\frac{\theta}{2}\sin\frac{\theta}{2}(u\times v)\right) \\ &= (0,\ (\cos\theta)v + (1-\cos\theta)u(u\cdot v) + \sin\theta(u\times v)). \end{aligned}$$

Observe that the vector part of this quaternion is *identical* to the angular displacement equation for $R(v)$ presented in Eq. (2), implying that the quaternion rotation operator achieves the desired rotation.

Composing Rotations: (Optional) We have shown that each unit quaternion corresponds to a rotation in 3-space. This is an elegant representation, but can we manipulate rotations through quaternion operations? The answer is yes. In particular, the action of multiplying two unit quaternions results in another unit quaternion. Furthermore, the resulting product quaternion corresponds to the composition of the two rotations. In particular, given two unit quaternions $\mathbf{q}$ and $\mathbf{q}'$, a rotation by $\mathbf{q}$ followed by a rotation by $\mathbf{q}'$ is equivalent to a single rotation by the product $\mathbf{q}'' = \mathbf{q}'\mathbf{q}$. That is,

$$R_{\mathbf{q}'}R_{\mathbf{q}} = R_{\mathbf{q}''} \qquad \text{where } \mathbf{q}'' = \mathbf{q}'\mathbf{q}.$$

This follows from the associativity of quaternion multiplication, and the fact that $(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}^{-1}\mathbf{q}'^{-1}$, as shown below.

$$\begin{aligned} R_{\mathbf{q}'}(R_{\mathbf{q}}(\mathbf{p})) &= \mathbf{q}'(\mathbf{q}\mathbf{p}\mathbf{q}^{-1})\mathbf{q}'^{-1} = (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}^{-1}\mathbf{q}'^{-1}) \\ &= (\mathbf{q}'\mathbf{q})\mathbf{p}(\mathbf{q}\mathbf{q}')^{-1} = \mathbf{q}''\mathbf{p}\mathbf{q}''^{-1} \\ &= R_{\mathbf{q}''}(\mathbf{p}). \end{aligned}$$

Lecture 7: Geometric Programming: Sample Solutions

Samples: In the last few lectures, we have been discussing affine and Euclidean geometry, coordinate frames and affine transformations, and rotations. In this lecture, we work through a few examples of how to apply these concepts to solve a few concrete problems that might arise in the context of game programming. **Caveat**: I have not tested the Unity code given here, so don't trust it!

Shot-Gun Simulator:

Problem: You have been asked to implement a new weapon that behaves something like a shot gun. It has a wide ange of effectiveness, but it is only effective at relatively small distances. Suppose that the end of the muzzle of the gun is located at a point p (in 3-dimensional space), and it has been aimed at a target point t (see Fig. 21(a)). The gun shoots a spray of pellets within angle θ the central axis between p and t, but bullets are only effective up to a distance of r from the end of the muzzle. (Let's assume that θ is given in degrees and is strictly smaller than 90°.) Given a point q, write a procedure to determine whether the point q will be hit when the gun is fired.

Solution: We need to determine (1) whether q lies within the infinite cone about the central axis $\overline{pt}$ and (2) whether q is close enough to be hit. How do we determine the first condition? We can construct two vectors, one from p to t and one from p to q, and determine whether the angle between them is at most θ degrees (see Fig. 21(b)).

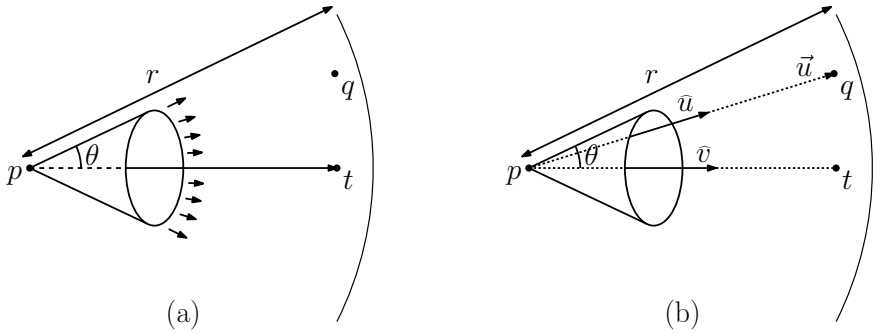


Fig. 21: Shot gun.

First, define a vector $\vec{v}$ to be the vector from p to t, that is, $\vec{v} \leftarrow t - p$. Next, define the vector $\vec{u}$ to be a vector that is directed from p to q, thus, $\vec{u} \leftarrow q - p$. Let us normalize these vectors to unit length, by defining $\widehat{u} \leftarrow \text{normalize}(\vec{u}) = \vec{u}/\ell(u)$, where $\ell(u) = \|u\| = \sqrt{u \cdot u}$. (Here we have used the property of dot product, that the dot product of a vector with itself is the vector's squared length.) We can do the same for $\vec{v}$.

In order for q to lie within the cone, we compute the angle between these vectors. Recall, that we can compute the cosine of two unit vectors by taking their dot product. Since the cosine is a monotonically *decreasing* function (for the angles in the range from 0 to 180°), this is equivalent to the condition $\widehat{u} \cdot \vec{v} \geq \cos\theta$.

Be careful! Remember that θ is given in degrees and the cosine function assumes that the argument is given in radians. To convert from degrees to radians we multiply by $\pi/180$. So the correct expression is

$$\widehat{u} \cdot \widehat{v} \;\geq\; \cos\left(\theta \cdot \frac{\pi}{180}\right).$$

To solve (2), it suffices to test whether If $\ell(u) > r$ then q is too far away to be hit.

To summarize, we have the following test.

$$\begin{array}{rcl}
\vec{v} & \leftarrow & t - p; \qquad \vec{u} \;\leftarrow\; q - p \\
\ell(v) & \leftarrow & \|\vec{v}\| \;=\; \sqrt{\vec{v} \cdot \vec{v}}; \qquad \ell(u) \;\leftarrow\; \|\vec{u}\| \;=\; \sqrt{\vec{u} \cdot \vec{u}} \\
\widehat{v} & \leftarrow & \text{normalize}(\vec{v}) \;=\; \vec{v}/\ell(v); \qquad \widehat{u} \;\leftarrow\; \text{normalize}(\vec{u}) \;=\; \vec{u}/\ell(u) \\
c_1 & \leftarrow & \widehat{u} \cdot \widehat{v} \\
c_2 & \leftarrow & \cos\left(\theta \cdot \dfrac{\pi}{180}\right) \\
\text{return} & & \textbf{true} \text{ iff } (c_1 \geq c_2 \;\text{ and }\; \ell(u) \leq r).
\end{array}$$

A Unity implementation of this procedure (which I haven't tested) can be found in the following code block.

Projectile Shooting: Your game involves a shooting an object (and arrow, rock, grenade, or other projectile) in a certain direction. Your boss wants you to write a program to determine where the projectile will land, as part of an aiming tool for inexperienced players.

Suppose that the projectile is launched from a location that is h meters. Following Unity's convention the projectile starts above on the vertical (y) axis, at coordinates $(0, h, 0)$. Suppose that the projectile is launched with a velocity given by the vector $\vec{v}_0 = (v_{0,x}, v_{0,y}, v_{0,z})$. Let's assume that the arrow is shot upwards, that is, $v_{0,y} > 0$. To simplify matters, let's assume that the projectile is shot in the forward (z) direction. Thus $v_{0,x} = 0$ and $v_{0,z} > 0$. We want to determine the distance ℓ from the shooter where the projectile hits the ground.

Does a shot gun fired at p toward t hit point q?

```
bool HitMe (Vector3 p, Vector3 t, float theta, float r, Vector3 q) {
  Vector3 v = t - p;                 // vector from muzzle end to target
  Vector3 u = q - p;                 // vector from muzzle end to q
  float lu = u.magnitude;            // distance to q
  Vector3 vv = v.normalized;         // directional vector to t
  Vector3 uu = u.normalized;         // directional vector to q
  float c1 = Vector3.Dot (uu, vv);   // cosine of angle between vectors
  float c2 = Mathf.Cos (theta * Mathf.PI / 180); // cosine of hit cone
  return (c1 >= c2) && (lu <= r);    // target within cone and distance
}
```

Let $t = 0$ denote the time at which the object is shot. After consulting a standard textbook on Physics, we are reminded that (on Earth at least) the force of gravity results in an acceleration of $g \approx 9.8m/s^2$.

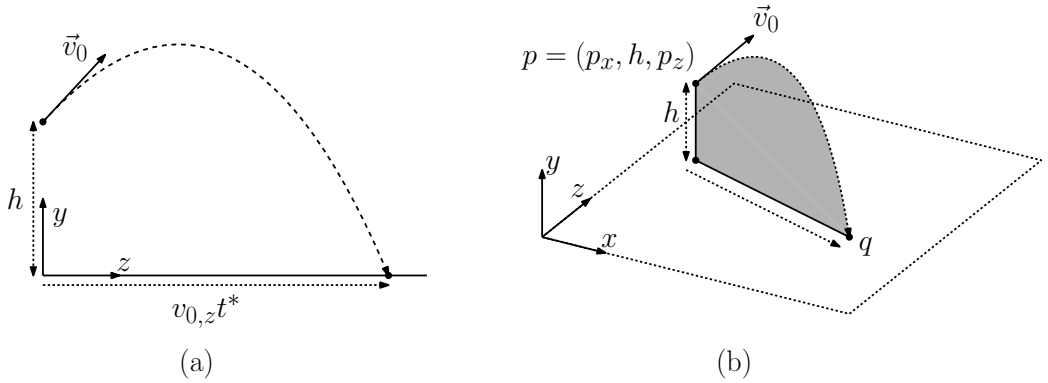


Fig. 22: Projectile shooting

After consulting your physics text, you find out that after t time units have elapsed, the position of the projectile is $p(t) = (z(t), y(t))$, where

$$z(t) \;=\; v_{0,z}t \quad \text{and} \quad y(t) \;=\; h + v_{0,y}t - \frac{1}{2}gt^2.$$

(Assuming no wind resistance, the projectile's motion with the z-axis is constant over time as $v_{0,z}$. It's motion with respect to the y-axis follows a downward parabolic arc as a function of time t.)

Time of Impact: Letting $a = g/2$, $b = -v_{0,y}$, and $c = -h$, we seek the value of t such that $at^2 + bt + c = 0$. (We have intentionally negated the coefficients so that $a > 0$.) By the quadratic formula we have

$$t \;=\; \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \;=\; \frac{v_{0,y} \pm \sqrt{v_{0,y}^2 + 2gh}}{g}.$$

Note that the quantity under the square-root sign is positive and is larger than $v_{0,y}$, which implies that both roots exist, one is positive and one is negative. Clearly, we want the positive root. Thus, we take the "+" root from the "±" option, which yields $t^* = \left(v_{0,y} + \sqrt{v_{0,y}^2 + 2gh}\right)/g$.

Location of Impact: We know that the projectile moves at a rate of $v_{0,z}$ units per second horizontally. Therefore, at time $t = t^*$ it has traveled a distance of $v_{0,z}t^*$ units. Since we started at the origin, the location we hit the ground is $(x, y, z) = (0, 0, v_{0,z}t^*)$.

Generalizing this: We assumed that the projectile was shot along the z-axis. Note that the generalization is very easy. We compute t^* in the same manner as above, since it depends only on the height and vertical velocity. Then we apply the same reasoning for x as for z. The projectile travels a distance of $v_{0,x}t^*$ units along the x-axis, so its final position is $(x, y, z) = (v_{0,x}t^*, 0, v_{0,z}t^*)$. We also assumed that the projective was shot from h units above the origin. If, instead it had been shot from some arbitrary point (p_x, h, p_z), then we would displace the final location by this amount, as $q = (p_x + v_{0,x}t^*, 0, p_z + v_{0,z}t^*)$. This is where we would draw our spot for aiming tool (see Fig. 22(b)).

Aiming Tool for Projectile

```
Vector3 HitSpot (Vector3 p, Vector3 v) {
  float h = p.y;                      // height above ground
  float a = 9.8f/2.0f;                // quadratic coefficients
  float b = -v.y;
  float c = -h;
  float t = (-b + Math.Sqrt(b*b - 4*a*c)) / (2*a); // time in the air
  return new Vector3 (p.x + v.x*t, 0, p.z + v.z*t); // where we hit
}
```

Shooting and Arrow: Before leaving the topic of shooting the projectile, it is worth observing that the Unity Physics engine can simulate the motion of the projectile. This will look fine if the projectile is a ball. However, if the projectile is an arrow, the arrow will not "turn" properly in the air in the manner that arrows do. Unity will simply translate the arrow through space, without rotating it (see Fig. 23(a)). This raises the question, "How can we rotate the arrow to point in the direction it is traveling?" (see Fig. 23(b))

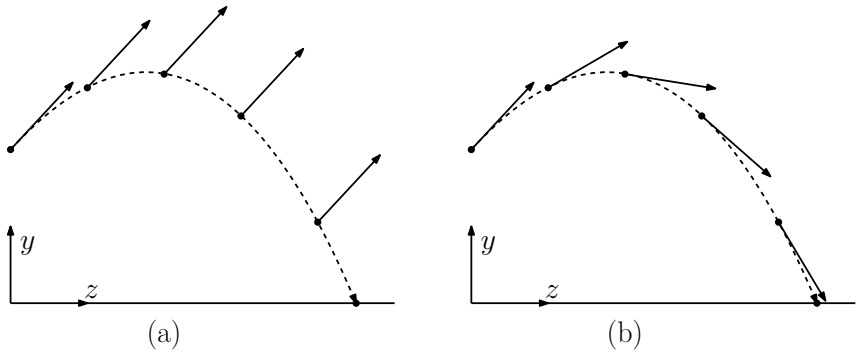


Fig. 23: Arrow shooting

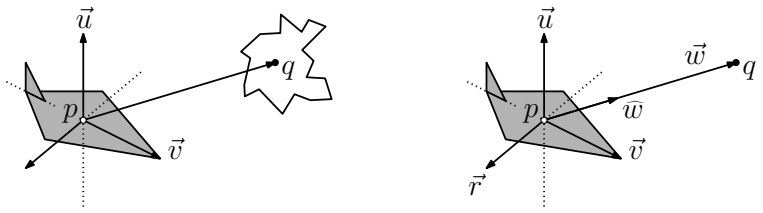
This may seem to be a complicated question. Clearly, this is not a rotation about the axes, and so Euler angles would not be the right approach. We need to use quaternions. We want to specify a quaterion that will cause the arrow to rotate in the direction it is moving. Luckily for us, Unity provides a function that does almost what we need. The Unity function Quaternion.LookRotation(Vector3 forward) generates a rotation (represented as a quaternion) that aligns the forward (z) axis with the argument. Thus, to align the arrow with the direction it is heading, we can use the following Unity commands:

```
RigidBody rb = getComponent<RigidBody> ();
transform.rotation = Quaternion.LookRotation (rb.velocity);
```

This will rotate the object so its orientation matches its velocity, as desired.

Evasive Action:

Problem: In your latest game, you are simulating the AI for some alien space ships. Each space ship is associated with its current position, a point p, and two unit-length vectors. The first vector $\vec{v}$ indicates the direction in which the space ship is flying. The second vector $\vec{u}$ is orthogonal to $\vec{v}$ and indicates the direction that is up relative to the pilot flying the space ship (see Fig. 24(a)). There are various obstacles to be avoided (asteroids, and such) and the AI system needs to issue turning commands to avoid these obstacles. Given an obstacle at some point q, the question that we want to determine is whether we should turn (yaw) to the left or right and whether we should turn (pitch) up or down to avoid the collision. (We won't worry about the actual number of degrees of rotation for now, just the direction.)



(a) (a)

Fig. 24: Evasive action.

Solution: First observe that $\vec{w} \leftarrow q - p$ defines a vector that is directed from the space ship to the obstacle (see Fig. 24(b)). To convert this into a unit vector (since we just care about the direction), let us normalize it to unit length. (Recall that normalizing a vector involves dividing a vector by its length.) We can compute the length of a vector as the square root of it dot product with itself. Thus, we have

$$\begin{aligned} \vec{w} &\leftarrow q - p \\ \widehat{w} &\leftarrow \text{normalize}(\vec{w}) = \frac{\vec{w}}{\|\vec{w}\|} = \frac{\vec{w}}{\sqrt{\vec{w} \cdot \vec{w}}} = \frac{\vec{w}}{\sqrt{w_x^2 + w_y^2 + w_z^2}}. \end{aligned}$$

What we want to know is whether this vector is pointing to the space-ship pilot's left or right (in which case we will turn the opposite direction), or is above or below (in which case we will pitch in the opposite direction).

Let's first tackle the problem of whether to turn pitch up or down. We can determine this by checking whether the angle between the up-vector $\vec{u}$ and $\widehat{w}$. If this angle is smaller than 90°, then the obstacle is above us and we should pitch downward. Otherwise, we should pitch upward. Given that both vectors have unit length, we can compute the cosine of the angle between them by the dot product. If the dot product is positive, the angle is smaller than 90°, thus the obstacle is above, and we turn down. Otherwise, we turn down. We have

$$\begin{aligned} \widehat{w} \cdot \vec{u} \geq 0 &\quad \Rightarrow \quad \text{(obstacle above) pitch downwards} \\ \widehat{w} \cdot \vec{u} < 0 &\quad \Rightarrow \quad \text{(obstacle below) pitch upwards.} \end{aligned}$$

(By the way, since we are only checking the sign of this dot product, not its magnitude, it was not really necessary to normalize $\vec{w}$ to unit length. We could have substituted $\vec{w}$ for $\widehat{w}$ above without affecting the correctness of the result.)

Next, let's consider whether to turn left or right. We would like to perform a similar type of computation, but to do so, we should generate a vector that indicates left and right relative to

the pilot of the ship. Such a vector will be orthogonal both to the direction that we are flying and to the up direction. We can obtain such a vector using the cross-product. In particular, define a vector $\vec{r} \leftarrow \vec{v} \times \vec{u}$. By the right-hand rule, this vector will point to the pilot's right. By our assumption that $\vec{v}$ and $\vec{u}$ are orthogonal to each other and of unit length, it follows from the definition of the cross product that $\vec{r}$ will also be of unit length.

We reason in an analogous manner to the up-down case. If the angle between $\widehat{w}$ and $\vec{r}$ is smaller than 90°, then the obstacle is to our right, and we turn left to avoid it. Otherwise, we turn right. This is equivalent to testing whether the cosine of the angle is positive or negative. Thus, we have

$$\begin{aligned} \vec{r} &\leftarrow \vec{v} \times \vec{u} \\ \widehat{w} \cdot \vec{r} \geq 0 &\Rightarrow \text{(obstacle to the right) yaw to the left} \\ \widehat{w} \cdot \vec{r} < 0 &\Rightarrow \text{(obstacle to the left) yaw to the right.} \end{aligned}$$

A Unity implementation of this procedure (which I haven't tested) can be found in the following code block.

Turning a ship at position p to avoid obstacle at q

```
void Evade (Vector3 p, Vector3 v,  Vector3 u, Vector3 q) {
  Vector3 w = q - p;                // vector from pilot to obstacle
  float l = w.magnitude;            // distance to obstacle
  Vector3 ww = w.normalized;        // directional vector to obstacle
  if (Vector3.Dot (ww, u) >= 0)     // obstacle is above?
    PitchDown ();
  else
    PitchUp ();
  Vector3 r = Vector3.Cross(v, u); // vector to pilot's right
  if (Vector3.Dot (ww, r) >= 0)     // obstacle is to the right
    YawToLeft ();
  else
    YawToRight ();
}
```

We have not discussed how to perform the pitch or yaw operations. In Unity, these could be expressed as rotations about the vectors $\vec{r}$ and $\vec{u}$, respectively.

Lecture 8: Geometric Data Structures: Enclosures and Spatial Indices

Geometric Objects and Queries: Someone once defined a *computer game* as a "database with a fun interface". Large games involve the storage and maintenance of a huge number of geometric objects, many of which change dynamically over time, and the game software needs to be able to access this information efficiently. Access to these structures takes form of *queries* (asking questions about the objects of the database) and *updates* (making changes to these objects).

What sorts of geometric queries might we be interested in asking? This depends a great deal about the application at hand. Queries typically involve determining what things are "close by." One reason is that nearby objects are more likely to have interesting interactions in a game (collisions or attacks). Of course, there are other sorts of interesting geometric properties. For example, in a shooting game, it may be of interest to know which other players have a line-of-sight to a given entity.

While we will focus on purely geometric data in this lecture, it is worth noting that geometry of an object may not be the only property of interest. For example, the query "locate all law enforcement vehicles within a half-mile radius of the player's car", might be quite reasonable for a car-theft game. Such queries involve both geometric properties (half-mile radius) and nongeometric properties (law enforcement). Such hybrid queries may involve a combination of multiple data structures.

Bounding Enclosures: When storing complex objects in a spatial data structure, it is common to first approximate the object by a simple enclosing structure. Bounding enclosures are often handy as a means of approximating an object as a filter in collision detection. If the bounding enclosures do not collide, then the objects to not collide. If they do, then we strip away the enclosures and apply a more extensive intersection test to the actual objects. Examples of bounding structures include:

Axis-aligned bounding boxes: This is an enclosing rectangle whose sides are parallel to the coordinate axes (see Fig. 25(a)). They are commonly called *AABBs* (axis-aligned bounding boxes). They are very easy to compute. (The corners are based on the minimum and maximum x- and y-coordinates.) An AABB can be represented by two points, for example, the lower-left point p^- and the upper-right point p^+. AABBs are preserved under translation, but not under rotation.

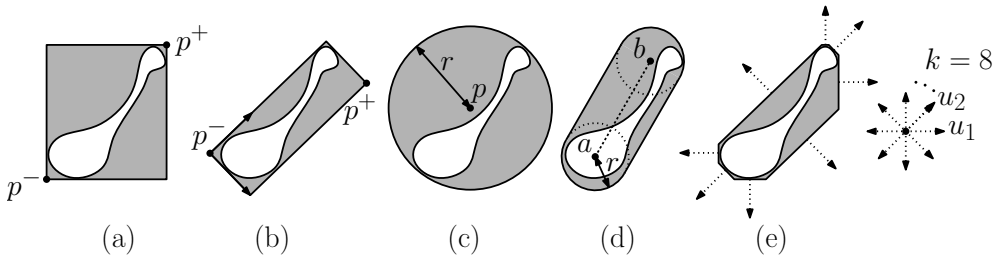


Fig. 25: Examples of common enclosures: (a) AABB, (b) general BB, (c) sphere, (d) capsule, (e) 8-DOP.

General bounding boxes: The principal shortcoming of axis-parallel bounding boxes is that it is not possible to rotate the object without recomputing the entire bounding box. In contrast, general (arbitrarily-oriented) bounding boxes can be rotated without the need to recompute them (see Fig. 25(b)).

A natural approach to represent such a box is to describe the box as an AABB but relative to a different coordinate frame. For example, we could define a frame whose origin is one of the corners of the box and whose axes are aligned with the box's sides. By applying an appropriate affine transformation, we can map the general box to an AABB.

Computing the minimum bounding box is not simple. It will be the AABB for an appropriate rotation of the body, but determining the best rotation (especially in 3-space) is quite tricky.

Bounding spheres: These are among the most popular bounding enclosures. A sphere can be represented by a center point p and a radius r (see Fig. 25(b)). Spheres are invariant under rigid transformations, that is under translation and rotation. Unfortunately, they are not well suited to skinny objects.

Minimum bounding spheres are tricky to compute exactly. A commonly used heuristic is to first identify (by some means) a point p that lies near the center of the body, and then set the radius just large enough so that a sphere centered at p encloses the body. Identifying the point p is tricky. One heuristic is set p to be the center of gravity of the body. Another is to compute two points a and b on the body that are farthest apart from each other. This is called the *diametrical pair*. Define p to be the midpoint of the segment $\overline{ab}$.

Bounding ellipsoids: The main problem with spheres (and problem that also exists with axis-parallel bounding boxes) is that skinny objects are not well approximated by a sphere. A ellipse (or generally, an ellipsoid in higher dimensions) is just the image of a sphere under an affine transformations.

As with boxes, ellipsoids may either be axis-parallel (meaning that the principal axes of the ellipse are parallel to the coordinate axes) or arbitrary.

As with spheres, minimum bounding ellipsoids are difficult to compute exactly. Various heuristics can be employed. For example, you can use the diametrical pair as defining the principal axis of the ellipse, but this is not generally optimal.

Capsules: This shape can be thought of as a "rounded cylinder." It consists of the set of points that lie within some distance r of a line segment $\overline{ab}$ (see Fig. 25(c)).

k-DOPs: People like objects bounded by flat sides, because the mathematics involved is linear. (There is no need to solve algebraic equations.) Unfortunately, an axis-aligned bounding box may not be a very close approximation to an object. (Imagine a skinny diagonal line segment.) As mentioned above, computing the minimum general bounding box may also be quite complex. A k-DOP is a compromise between these two.

Given an integer parameter $k \geq 3$ (or generally $k \geq d+1$), we generate k directional vectors that are roughly equally spaced and span the entire space. For example, in two-dimensional space, we might consider a set of unit vectors at angles $2\pi i/k$, for $0 \leq i < k$. Let $\{u_1, \ldots, u_k\}$ be the resulting vectors. We then compute extreme point of the object along each of these directions. We then put an orthogonal hyperplane through this point. The intersection of these hyperplanes defines a convex polygon with k sides (generally, a convex polytope with k facets) that encloses the objects. This is called a *k-discrete oriented polytope*, or *k-DOP* (see Fig. 25(d) and (e)).

Detecting Collisions: By enclosing an object within a bounding enclosure, collision detection reduces to determining whether two such enclosures intersect each other. Note that if we support k different types of enclosure, we need to handle all possible pairs of combinations of collisions. Here are a few examples:

AABB-AABB: We can test whether two axis-aligned bounding boxes overlap by testing that all pairs of intervals overlap. For example, suppose that we have two boxes b and b', where the box b extends from the lower-left corner (x_1, y_1) to the upper right corner (x_2, y_2) and b' extends from the lower-left corner (x'_1, y'_1) to the upper right corner (x'_2, y'_2) (see Fig. 26(a)). These boxes overlap if and only if

$$[x_1, x_2] \cap [x'_1, x'_2] \neq \emptyset \qquad \text{and} \qquad [y_1, y_2] \cap [y'_1, y'_2] \neq \emptyset$$

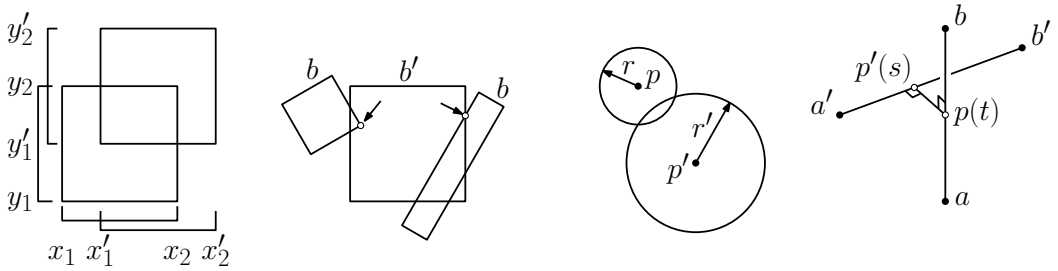


Fig. 26: Detecting collisions.

Box-Box: Determining whether two arbitrarily oriented boxes b and b' intersect is a nontrivial task. If they do intersect, one of the following must happen (see Fig. 26(b)):

- A vertex of b lies within b' or vice versa.

- An edge of b intersects a face of b' or vice versa.

One way to simplify these computations is to first compute a rotation that aligns one of the two boxes with the coordinate axes. The operations involved with determining point membership or edge-face intersection with an AABB is simpler than for general boxes. If both tests fail, we reverse the roles of the two boxes and try again.

Sphere-Sphere: We can determine whether two spheres intersect by computing the distances between their centers. Given two spheres, one with center p and radius r and the other with center p' and radius r', they intersect if and only if $\text{dist}(p, p') \leq r + r'$ (see Fig. 26(c)).

Capsule-Capsule: Just as the sphere-sphere intersection reduces to computing the distances between the center points, the capsule-capsule intersection reduces to computing the distance between two line segments. In particular, given a capsule defined as the set of points that are within distance r of $\overline{ab}$ and the capsule defined as the set of points that are within distance r' of $\overline{a'b'}$, we would first compute the distance between the line segments $\overline{ab}$ and $\overline{a'b'}$. Assuming we can do this, the capsules intersect if and only if the shortest distance between the line segments is at most $r + r'$.

Here is a short sketch of how to compute the shortest distance between two line segments. We know that any point on the infinite $\overline{ab}$ can be expressed as an affine combination $p(t) = (1-t)a+tb$, where $0 \leq t \leq 1$. Similarly, any point on the line $\overline{a'b'}$ can be expressed as $p'(s) = (1-s)a'+sb'$. At the closest point between the two (infinite) lines, the line segment $\overline{p(t)p'(s)}$ must be perpendicular to both lines (see Fig. 25(d)). (Proving this is a simple exercise in Euclidean geometry.) Enforcing this perpendicularity condition involves solving a linear system of two equations and two unknowns s and t. We can easily solve the resulting 2×2 linear system of equations. We clamp the values of s and t to the interval $[0, 1]$, since all other points lie off the line segment. Once we know s and t, we can get the coordinates of the points easily, and once we have the coordinates we can get the distance.

Hierarchies of bounding bodies: What if the above bounding volumes are not sufficiently accurate for your purposes? A natural generalization is that of constructing a hierarchy consisting of multiple levels of bounding bodies, where the bodies at a given level enclose a constant number of bodies at the next lower level.

If you consider the simplest case of axis-aligned bounding boxes, the resulting data structure is called an *R-tree*. In Fig. 27 we given an example, where the input boxes are shown in (a), the hierarchy (allowing between 2–3 boxes per group) is shown in (b) and the final tree is shown in (c).

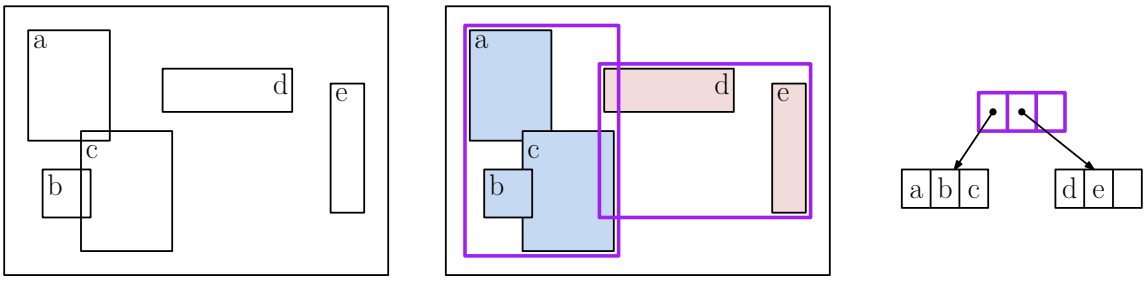


Fig. 27: A hierarchy of bounding boxes: (a) the input boxes, (b) the hierarchy of boxes, (c) the associated R-tree structure.

There are a number of interesting (and often quite complicated) technical issues in the design of R-trees. For example: What is the best way to cluster smaller boxes together to form larger boxes? How do you minimize the wasted space within each box? How do you minimize the overlap between

boxes at a given level of the tree? As optimization problems, these are all quite challenging. Usually, designers apply simple heuristic strategies to obtain good, albeit suboptimal, solutions.

This structure is widely used in the field of spatial databases, since the number of boxes contained within another box can be adjusted so that each node corresponds to a single disk block. (In this respect, it is analogous to the famous *B-tree* data structure for storing 1-dimensional data sets.)

Grids: One virtue of simple data structures is that are usually the easiest to implement, and (if you are fortunate in your choice of geometric model) they may work well. An example of a very simple data structure that often performs quite well is a simple rectangular grid. For simplicity, suppose that want to store a collection of objects. We will assume that the distribution of objects is fairly uniform. In particular, we will assume that there exists a positive real Δ such that almost all the objects are of diameter at most $c\Delta$ for some small constant c, and the number of objects that intersect any cube of side length Δ is also fairly small.

If these two conditions are met, then a square grid of side length Δ may be a good way to store your data. Here is how this works. First, for each of your objects, compute its axis-aligned bounding box. We can efficiently determine which cells of the grid are overlapped by this bounding box as follows. Let p and q denote the bounding box's lower-left and upper-right corners (see Fig. 28(a)). Compute the cells of the grid that contain these points (see Fig. 28(b)). Then store a pointer to the object in all the grid cells that lie in the rectangle defined by these two cells (see Fig. 28(c)). Note that this is not perfect, since the object may be associated with grid cells that it does not actually intersect. This increases the space of the data structure, but it does not compromise the data structure's correctness.

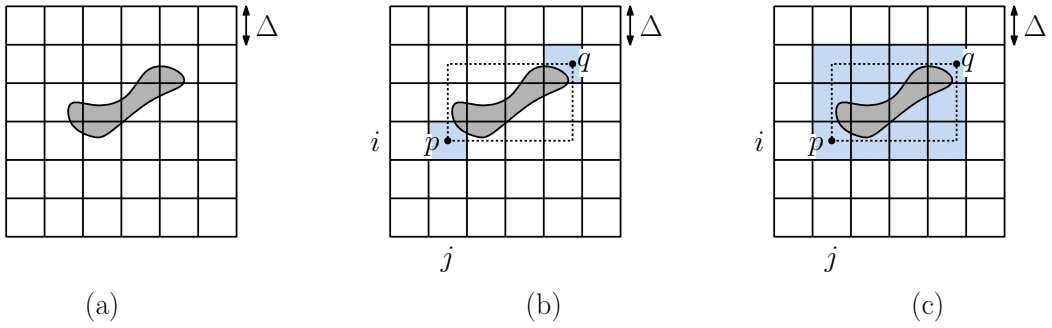


Fig. 28: Storing an object in a grid.

Computing the indices of the grid cell that contain a given point is a simple exercise in integer arithmetic. For example, if $p = (p_x, p_y)$, then let

$$j = \left\lfloor \frac{p_x}{\Delta} \right\rfloor \qquad \text{and} \qquad i = \left\lfloor \frac{p_y}{\Delta} \right\rfloor.$$

Then, the point p lies within the grid cell $G[i, j]$.

If the diameter of most of the objects is not significantly larger than Δ, then each object will only be associated with a constant number of grid cells. If the density of objects is not too high, then each grid square will only need to store a constant number of pointers. Thus, if the above assumptions are satisfied then the data structure will be relatively space efficient.

Storing a Grid: As we have seen, a grid consists of a collection of cells where each cell stores a set of pointers to the objects that overlap this cell (or at least might overlap this cell). How do we store these cells? Here are a few ideas.

***d*-dimensional array:** The simplest implementation is to allocate a d-dimensional array that is sufficiently large to handle all the cells of your grid. If the distribution of objects is relatively uniform,

then it is reasonable to believe that a sizable fraction of the cells will be nonempty. On the other hand, if the density is highly variable, there may be many empty cells. This approach will waste space.

Hash map: Each cell is identified by its indices, (i, j) for a 2-dimensional grid or (i, j, k) for the 3-dimensional grid. Treat these indices like a key into a hash map. Whenever a new object o is entered into some cell (i, j), we access the hash map to see whether this cell exists. If not, we generate a new cell object and add it to the hash map under the key (i, j). If so, we add a pointer to o into this hash map entry.

Linear allocation: Suppose that we decide to adopt an array allocation. A straightforward implementation of the d-dimensional array will result in a memory layout according to how your compiler chooses to allocate arrays, typically in what is called *row-major order* (see Fig. 29(a)). For example, if there are N columns, then the element (i, j) is mapped to index $i \cdot N + j$ in row-major order.

Why should you care? Well, the most common operation to perform on a grid is to access the cells that surround a given grid square. Memory access tends to be most efficient when accessed elements are close to one another in memory (since they are likely to reside within the same cache lines). The problem with row-major order is that entries in successive rows are likely to be far apart in physical memory.

A cute trick for avoiding this problem is to adopt a method of mapping cells to physical memory that is based on a *space filling curve*. There are many different space-filling curves. We show two examples in Figs. 29(b) and (c). The first is called the *Hilbert curve* and the second is known as the *Morton order* (also called the *Z-order*).

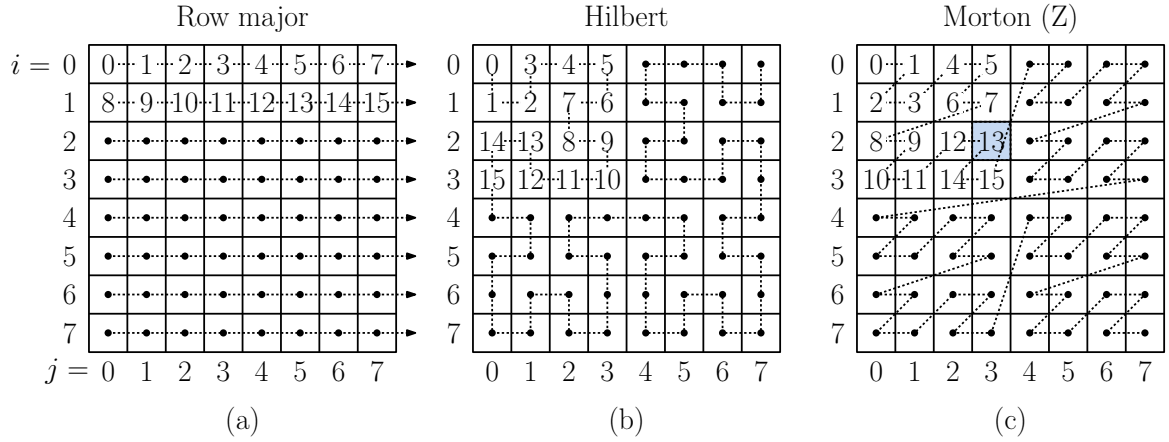


Fig. 29: Linear allocations to improve reference locality of neighboring cells: (a) row-major order, (b) the Hilbert curve, (c) the Morton order (or Z-order).

There is experimental evidence that shows that altering the physical allocation of cells can improve running times moderately. Unfortunately, the code that maps an index (i, j) to the corresponding address in physical memory becomes something of a brain teaser.

Computing the Morton Order: Between the Hilbert order and the Morton order, the Morton order is by far the more commonly use. One reason for this is that there are some nifty tricks for computing the this order. To make this easier to see, let us assume that we are working in two-dimensional space and that the grid of size $2^m \times 2^m$. The trick we will show applies to any dimension. If your grid is not of this size, you can embed it within the smallest grid that has this property.

Next, since the grid has 2^m rows and 2^m columns, we can view each row and column index as an m-element bit vector, call them $i = \langle i_1, \ldots, i_m \rangle_2$ and $j = \langle j_1, \ldots, j_m \rangle_2$, in order from most significant bit (i_1) to the least significant bit (i_m). Next, we take these bit vectors and interleave them as if we were shuffling a deck of cards:

$$k \;=\; \langle i_1, j_1, i_2, j_2, \ldots, i_m, j_m \rangle_2.$$

If you have not seen this trick before, it is rather remarkable that it works. As an example, consider the cell at index $(i, j) = (2, 3)$, which is labeled as 13 in Fig. 29(c). Expressing i and j as 3-element bit vectors we have $i = \langle 0, 1, 0 \rangle_2$ and $j = \langle 0, 1, 1 \rangle_2$. Next, we interleave these bits to obtain

$$k \;=\; \langle 0, 0, 1, 1, 0, 1 \rangle_2 \;=\; 13,$$

just as we expected.

This may seem like a lot of bit manipulation, particularly if m is large. It is possible, however, to speed this up. For example, rather than processing one bit at a time, we could break i and j up into 8-bit bytes, and then for each byte, we could access a 256-element look-up table to convert its bit representation to one where the bits have been "spread out." (For example, suppose that you have the 8-element bit vector $\langle b_0, b_1, \ldots, b_7 \rangle_2$. The table look-up would return the 16-element bit vector $\langle b_0, 0, b_1, 0, \ldots, b_7, 0 \rangle_2$.) You repeat this for each byte, applying a 16-bit shift in each case. Finally, you apply an addition right shift of the j bit vector by a single position and bitwise "or" the two spread-out bit vectors for i and j together to obtain the final shuffled bit vector. By interpreting this bit vector as an integer we obtain the desired *Morton code* for the pair (i, j).

Quadtrees: Grids are fine if the density of objects is fairly regular. If there is considerable variation in the density, a quadtree is a practical alternative. You have probably seen quadtrees in your data structures course, so I'll just summarize the main points, and point to a few useful tips.

First off, the term "quadtree" is officially reserved for 2-dimensional space and "octree" for three dimensional space. However, it is too hard to figure out what the name should be when you get to 13-dimensional space, so I will just use the term "d-dimensional quadtree" for all dimensions.

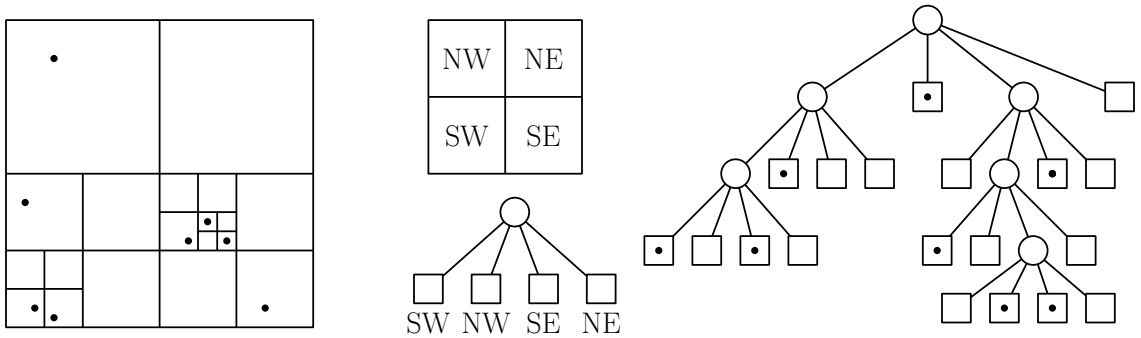


Fig. 30: A quadtree decomposition and the associated tree.

We begin by assuming that the domain of interest has been enclosed within a large bounding square (or generally a hypercube in d-dimensional space). Let's call this Q_0. Let us suppose that we have applied a uniform scaling factor so that Q_0 is mapped to the d-dimensional unit hypercube $[0, 1]^d$. A *quadtree box* is defined recursively as follows:

- Q_0 is a quadtree box

- If Q is any quadtree box, then the 2^d boxes that result by subdividing Q through its midpoint by axis aligned hyperplanes is also a quadtree box.

This definition naturally defines a hierarchical subdivision process, which subdivides Q_0 into a collection of quadtree boxes. This defines a tree, in which each node is associated with a quadtree box, and each box that is split is associated with the 2^d sub-boxes as its children (see Fig. 30). The root of the tree is associated with Q_0. Because Q_0 has a side length of 1, it follows directly that the quadtree boxes at level k of the tree have side length $1/2^k$.

Quadtree variants: (Optional material)

There are a couple of practical variants of quadtrees that are worth knowing about. Here are a few. (See Samet's book for much more information.)

Binary Quadtrees: In dimension 3 and higher, having to allocate 2^d children for every internal node can be quite wasteful. Unless the points are uniformly distributed, it is often the case that only a couple of these nodes contain points. An alternative is rely only on binary splits. First, split along the midpoint x-coordinate, then the midpoint y-coordinate, and so forth, cycling through the axes (see Fig. 31).

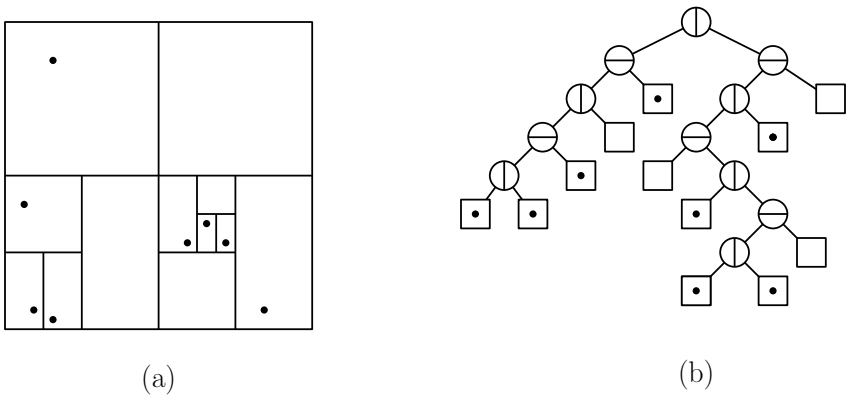


Fig. 31: A binary quadtree.

Linear Quadtree: A very clever and succinct method for storing quadtrees for point sets involves no tree at all! Recall the Morton order, described earlier in this lecture. A point (x, y) is mapped to a point in a 1-dimensional space by shuffling the bits of x and y together. This maps all the points of your set onto a space filling curve.

What does this curve have to do with quadtrees? It turns out that the curve visits the cells of the quadtree (either the standard, binary, or compressed versions) according to an in-order traversal of the tree (see Fig. 32).

How can you exploit this fact? It seems almost unbelievable that this would work, but you sort all the points of your set by the Morton order and store them in an array (or any 1-dimensional data structure). While this would seem to provide very little useful structure, it is remarkable that many of the things that can be computed efficiently using a quadtree can (with some additional modifications) be computed directly from this sorted list. Indeed, the sorted list can be viewed as a highly compressed encoding of the quadtree.

The advantage of this representation is that it requires zero additional storage, just the points themselves. Even though the access algorithms are a bit more complicated and run a bit more slowly, this is a very good representation to use when dealing with very large data sets.

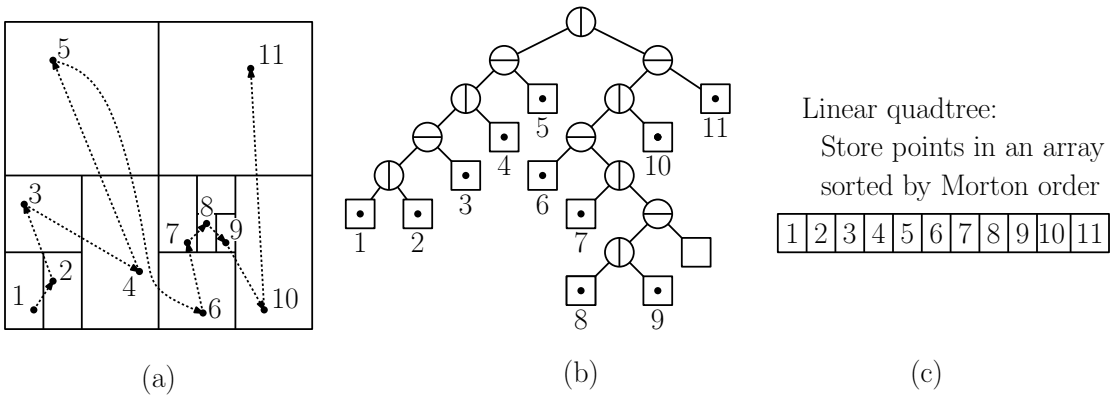


Fig. 32: A linear quadtree.

Kd-trees: While quadtrees are widely used, there are some applications where more flexibility is desired in how the object space is partitioned. There are a number of alternative index structures that are based on the hierarchically subdividing space into simple regions. Data structures based on such hierarchical subdivisions are often called *partition trees*. One of the most widely-used partition-tree structures is the kd-tree.

A *kd-tree*[3] is a partition tree based on orthogonal slicing. We start by assuming that all the points of our space are stored within some large bounding (axis-aligned) rectangle, which is associated with the root node of the tree. Every node of the tree is associated with a (hyper)-rectangular region, called its *cell*. Each internal node of the tree is associated with an axis-aligned *splitting plane*, which is used to split the cell in two. The points falling on one side are stored in one child and points on the other side are stored in the other. Each internal node t of the kd-tree is associated with the following quantities:

$t.cut\text{-}dim$	the cutting dimension (e.g., 0, 1, or 2 representing x, y, or z, respectively)
$t.cut\text{-}val$	the cutting value (a real number)

Of course, there generally may be additional information associated with each node (for example, the number of objects lying within the node's cell), depending on the exact application. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to $t.cut\text{-}val$ are stored in the left subtree, and the remaining points are stored in the right subtree (see Fig. 33). (If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side.) When a single point remains, we store it in a leaf node, whose only field $t.point$ is this point.

There are two key decisions in the implementation of the kd-tree.

How is the cutting dimension chosen? The simplest method is to cycle through the dimensions one by one. (This method is shown in Fig. 33.) Since the cutting dimension depends only on the level of a node in the tree, one advantage of this rule is that the cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

One disadvantage of this splitting rule is that, depending on the data distribution, this simple cyclic rule may produce very skinny (elongated) cells, and such cells may adversely affect query times. Another method is to select the cutting dimension to be the one along which the points have the greatest *spread*, defined to be the difference between the largest and smallest coordinates. Bentley call the resulting tree an *optimized kd-tree*.

[3]The terminology surrounding kd-trees has some history. The data structure was proposed originally by Jon Bentley. In his notation, these were called "k-d trees," short for "k-dimensional trees" since the generalize classical binary trees for 1-dimensional data. Thus, there are 2-d trees, 3-d trees, and so on. However, over time, the specific value of k was lost, and they are simply called kd-trees.

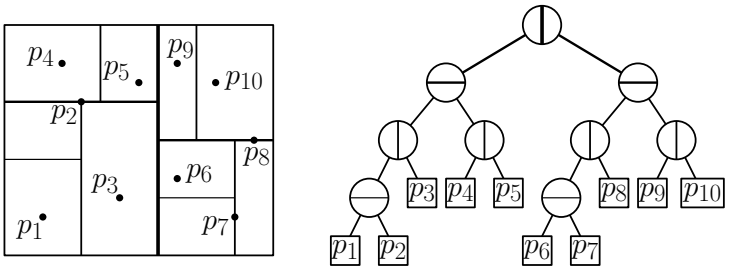


Fig. 33: A kd-tree and the associated spatial subdivision.

How is the cutting value chosen? To guarantee that the tree is balanced, that is, it has height $O(\log n)$, the best method is to let the cutting value be the *median coordinate value* along the cutting dimension. In our example, when there are an odd number of points, the median is associated with the left (or lower) subtree.

Note that a kd-tree is a special case of a more general class of hierarchical spatial subdivisions, called *binary space partition trees* (or *BSP trees*) in which the splitting lines (or hyperplanes in general) may be oriented in any direction, not just axis-aligned.

Lecture 9: Basics of Skeletal Animation and Kinematics

Game Animation: Most computer games involve *characters* that move around in a fluid and continuous manner. Unlike objects that move according to the basic laws of physics (e.g., balls, projectiles, vehicles, water, smoke), the animation of skeletal structures may be subject to many complex issues, such as physiological constraints (e.g., how to jump onto a platform), athletic technique (e.g., how a football quarterback throws a pass), stylistic choices (e.g., how a dancer moves), or simply the arbitrary conventions of everyday life (e.g., how to hold chopsticks). Producing natural looking animation is the topic of our next few lectures.

Skeletal Model and Tree Structure: The most common form of character animation used in high-end 3-dimensional games is through the use of skeletal animation. A character is modeled as *skin* surface stretched over a *skeletal framework* which consists of moving *joints* connected by segments called *bones* (see Fig. 34(a) and (b)). Note that "skin" refers to the model's surface, which typically includes not only skin but the clothing the model is wearing (see Fig. 34(c)). Animation is performed by modifying the relationships between pairs of adjacent joints, for example, by altering joint angles and deforming the skin accordingly. We will discuss this process extensively later.

A skeletal model is based on a hierarchical representation where peripheral elements (e.g., hands and feet) are linked as children of more central elements (e.g., legs, arms, torso, etc.). Clearly, a skeletal model can be represented internally as a multi-way rooted tree, in which each node represents a single joint. The *bones* of the tree are not explicitly represented, since (as we shall see) they do not play a significant role in the animation or rendering process. We will discuss later how the skin is represented. For now, let us consider how the joints are represented.

We assume that the tree is represented as any standard (rooted, unordered) multi-way tree. For example, for any node it should be possible to enumerate all the children of this node, to test whether the node is the root, and if it is not the root to determine its parent node. In this lecture, we will denote each joint by an integer index, say j, and we will let $p(j)$ denote j's parent. If we consider just the parent links, the result is an *inverted tree structure*, where all paths lead to the root (see Fig. 35(b)). We can make the assumption that the root is identified by a special index, say $j = 0$. In addition, each node will store some *internal information*, as will be discussed below.

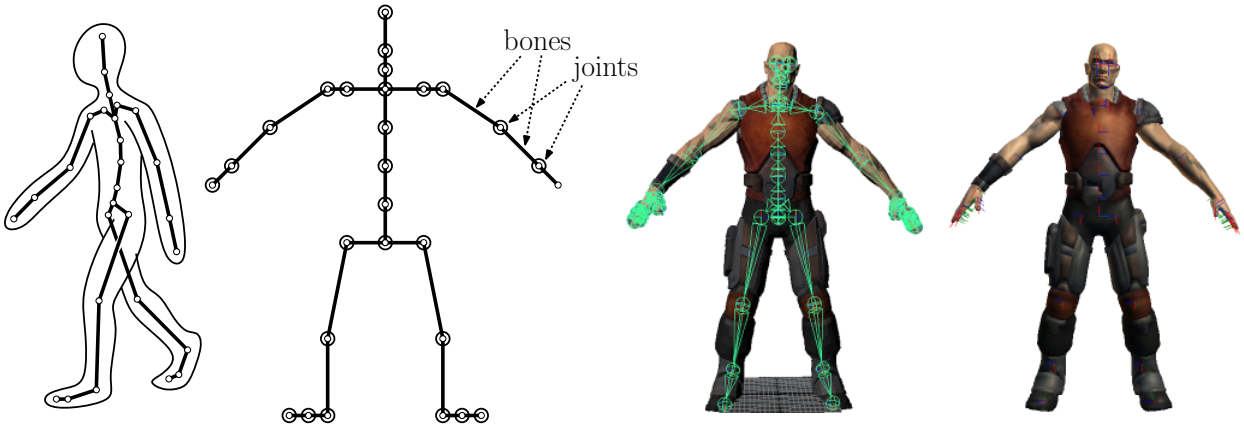


Fig. 34: (a) and (b) skeletal model and (c) the bind (or reference) pose.

Bind Pose: Before discussing animation on skeletal structures, it is useful to first say a bit about the notion of a *pose*. In humanoid and animal skeletons, joints move by means of rotations[4] (as opposed say to translation, which arises with some robots). Assigning angles to the various joints of a skeleton uniquely specifies the skeleton's exact geometric structure, called its *pose*.

When a designer defines the initial layout of the model's skin, the designer does so relative to a default pose, which is called the *reference pose* or the *bind pose*.[5] For human skeletons, the bind pose is typically one where the character is standing upright with arms extended straight out to the left and right (similar to Fig. 34(b) above).

Joint Internal Information: Each joint can be thought of as defining its own *joint coordinate frame* (see Fig. 35(a)). Recall that in affine geometry, a coordinate frame consists of a point (the origin of the frame) and three mutually orthogonal unit vectors (the x, y, and z axes of the frame). Given the skeleton's inverted tree structure (see Fig. 35(b)), rotating a joint can be achieved by applying a suitable rotation transformation to its associated coordinate frame. Each frame of the hierarchy is understood to be positioned *relative* to its parent's frame. In this way, when the shoulder joint is rotated, the descendants' joints (elbow, hand, fingers, etc.) also move as a result (see Fig. 35(c)).

Change-of-Coordinates Transformation: In order to determine the motion of the various bones that result from some joint rotation, we need to know the relationships between the various joints of the skeleton. There is a very general and elegant way of doing this through the application of affine geometry. Given any two coordinate frames in d-dimensional space, it is possible to convert a point (or free vector) represented in one coordinate frame to its representation in the other frame by multiplying the point (given as a $(d+1)$-dimensional vector in homogeneous coordinates) times an suitable $(d+1) \times (d+1)$ matrix. The resulting affine transformation is called a *change-of-coordinates transformation*.

Constructing such transformations is an exercise in linear algebra. For the sake of completeness, let us consider the process in a simple 2-dimensional example. Suppose we have two coordinate frames, F

[4]It is rather interesting to think about how this happens for your own joints. For example, your shoulder joint has two degrees of freedom, since it can point your upper arm in any direction it likes. Your elbow also has two degrees of freedom. One degree comes by flexing and extending your forearm. The other can be seen when you turn your wrist, as in turning a door knob. Your neck has (at least) three degrees of freedom, since, like your shoulder, you can point the top of your head in any direction, and, like your elbow, you can also turn it clockwise and counterclockwise.

[5]I suspect that the name "bind pose" arises because designers attach or "bind" the skin to the model relative to this initial pose.

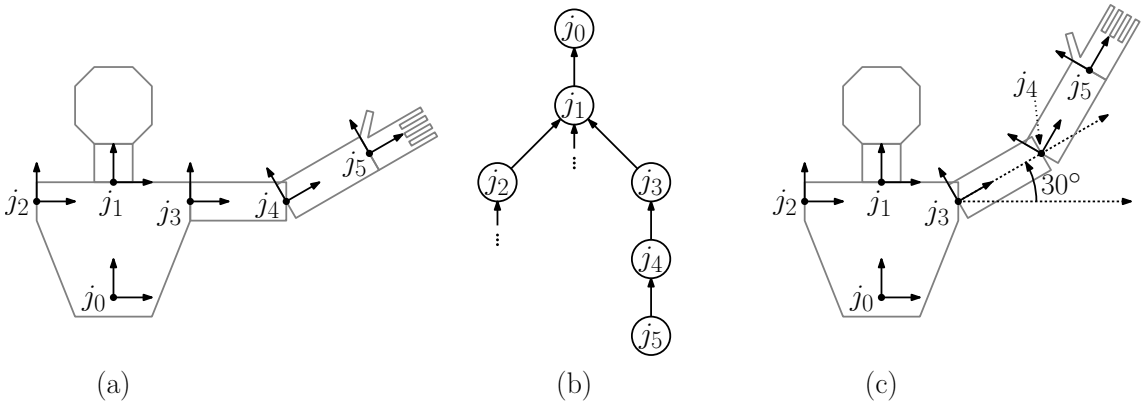


Fig. 35: (a) Skeletal model, (b) inverted tree structure, and (c) rotating a frame propagates to the descendants.

and G (see Fig. 36). Let $F.o$, $F.x$, and $F.y$ denote F's origin point, and its two basis vectors. Define $G.o$, $G.x$ and $G.y$ similarly.

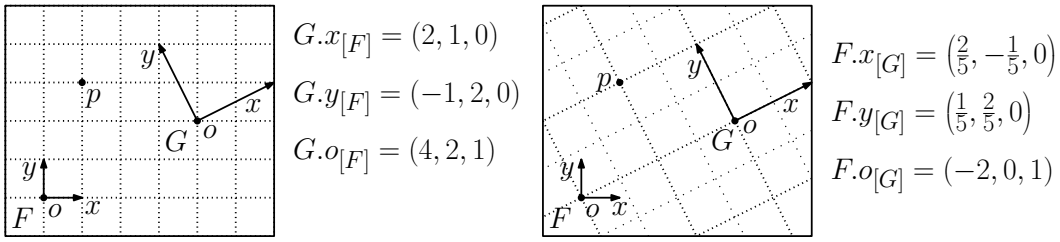


Fig. 36: Change-of-coordinates transformation.

Given any point in space, it can be represented either with respect to F's coordinate system or G's. For any point p, define $p_{[F]}$ to be p's homogeneous coordinates relative to frame F, and define $p_{[G]}$ similarly for frame G. We can do the same for any vector $\vec{v}$.

In order to define the change-of-coordinates transformation, we need to know first what G's basis elements are relative to F. In the above example, it is easy to verify that

$$G.x_{[F]} = (2, 1, 0), \quad G.y_{[F]} = (-1, 2, 0), \quad \text{and} \quad G.o_{[F]} = (4, 2, 1).$$

(Recall that we are using affine homogeneous coordinates, where the last component is 0 to denote a vector or 1 to denote a point.) Also, it is easy to verify that

$$F.x_{[G]} = \left(\frac{2}{5}, -\frac{1}{5}, 0\right), \quad F.y_{[G]} = \left(\frac{1}{5}, \frac{2}{5}, 0\right), \quad \text{and} \quad F.o_{[G]} = (-2, 0, 1).$$

To obtain the change-of-coordinates transformations from G to F, define $T_{F \leftarrow G}$ to be the transformation that maps a point given in G's coordinate system to its representation in F's coordinate system. This transformation can be represented as a matrix whose columns are $G.x_{[F]}$, $G.y_{[F]}$, and $G.o_{[F]}$:

$$T_{[F \leftarrow G]} = \begin{pmatrix} 2 & -1 & 4 \\ 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

Conversely, to convert the other direction, define $T_{G \leftarrow F}$ to be the transformation that maps a point given in F's coordinate system to its representation in G's coordinate system. This transformation can be represented as a matrix whose columns are $F.x_{[G]}$, $F.y_{[G]}$, and $F.o_{[G]}$:

$$T_{[G \leftarrow F]} = \begin{pmatrix} 2/5 & 1/5 & -2 \\ -1/5 & 2/5 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

(See any standard reference on linear algebra for a proof.)

Change-of-Coordinates Example: To test this, let's consider the point p and vector $\vec{v}$ in Fig. 37.

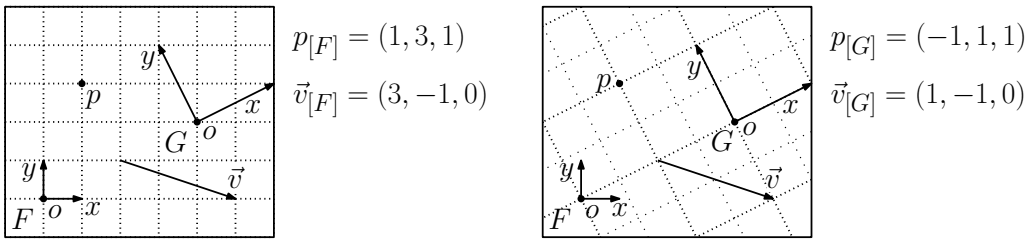


Fig. 37: Example of applying the change-of-coordinates transformation.

Clearly, $p_{[F]} = (1, 3, 1)$ and $p_{[G]} = (-1, 1, 1)$. Applying the above transformations, we obtain the expected results

$$T_{[F \leftarrow G]} \cdot p_{[G]} = \begin{pmatrix} 2 & -1 & 4 \\ 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} = p_{[F]},$$

and

$$T_{[G \leftarrow F]} \cdot p_{[F]} = \begin{pmatrix} 2/5 & 1/5 & -2 \\ -1/5 & 2/5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} = p_{[G]},$$

Next, consider $\vec{v}$. We have $\vec{v}_{[F]} = (3, -1, 0)$ and $\vec{v}_{[G]} = (1, -1, 0)$. Again, applying the above transformations, we have

$$T_{[F \leftarrow G]} \cdot \vec{v}_{[G]} = \begin{pmatrix} 2 & -1 & 4 \\ 1 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ -1 \\ 0 \end{pmatrix} = \vec{v}_{[F]},$$

and

$$T_{[G \leftarrow F]} \cdot \vec{v}_{[F]} = \begin{pmatrix} 2/5 & 1/5 & -2 \\ -1/5 & 2/5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ -1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = \vec{v}_{[G]}.$$

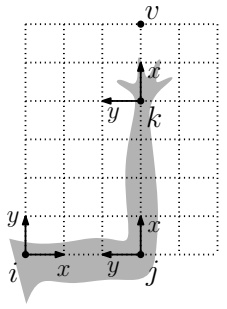
Abstraction Revisited: We have seen the use of homogeneous matrices before for the purpose of performing affine transformations (that is, for moving objects around in space). We are using the same mechanism here, but the meaning is quite different. Here the objects are not being moved, rather we are simply *translating* the names of points and vectors from one coordinate system to another. The geometric objects are themselves not moving.

You might wonder, "What's the difference in how you look at it?" Recall that in affine geometry we defined points and (free) vectors as different abstract objects, that employ the same representation

(homogeneous vectors). Here, we are distinguishing two different types of operations (affine transformations versus change-of-coordinates transformations), but using the same representation (homogeneous matrices) for both. Even though the same representation is being used, these should be conceptualized as two very different operations.

Joint Transformations: Returning to the problem of skeletal systems, let us assume that we are working in 3-dimensional space, and consider the skeleton in its bind pose. For any two joints j and k, define $T_{[k \leftarrow j]}$ to be the change-of-coordinates transformation that maps a point in joint j's coordinate system to its representation in k's coordinate system. (At this point we are not considering joint rotations.) That is, if v is a column vector in homogeneous coordinates representing of a point relative to j's coordinate system, then $v' = T_{[k \leftarrow j]} \cdot v$ is exactly the same point in space, but it is expressed in coordinates relative to k's coordinate frame. (In previous lectures I have used p for points and v for free vectors. Since we are using p for "parent", I'll refer to points by the letter v, but don't be confused.)

Given any non-root joint j, define the *local-pose transformation*, denoted $T_{[p(j) \leftarrow j]}$, to be an affine transformation that converts a point in j's coordinate frame to its representation in its parent's ($p(j)$) coordinate frame. Define the *inverse local-pose transformation*, denoted $T_{[j \leftarrow p(j)]}$, to be the inverse of this transformation. That is, it converts a point expressed relative to j's parent's frame back to j's frame. (Recall that these transformations do not change the position of a point. They simply translate the same point from its representation in one frame to another.)



(a)

$$v_{[k]} = (2, 0, 1)$$

$$v_{[j]} = (6, 0, 1)$$

$$v_{[i]} = (3, 6, 1)$$

(b)

Fig. 38: Three joints i, j, and k in a (rather nonstandard) bind pose. The point v is represented in homogeneous coordinates relative each frame.

Consider three joints i, j, and k, where $i = p(j)$ and $j = p(k)$ (see Fig. 38(a)). The local-pose transformation for k, $T_{[p(k) \leftarrow k]}$, can be expressed more succinctly as $T_{[j \leftarrow k]}$. Given a point $v_{[k]}$ expressed relative to k's frame, we can express it relative to j's frame as

$$v_{[j]} \;=\; T_{[j \leftarrow k]} \cdot v_{[k]}.$$

Similarly, a point $v_{[j]}$ expressed relative to j's frame can be expressed relative to i's frame as

$$v_{[i]} \;=\; T_{[i \leftarrow j]} \cdot v_{[j]}.$$

Combining these, we can express a point in k's frame relative to i's frame by taking the product of these two matrices

$$v_{[i]} \;=\; T_{[i \leftarrow j]} \cdot T_{[j \leftarrow k]} \cdot v_{[k]} \;=\; T_{[i \leftarrow k]} \cdot v_{[k]},$$

where $T_{[i \leftarrow k]} = T_{[i \leftarrow j]} \cdot T_{[j \leftarrow k]}$. Clearly, by multiplying appropriate chains of the local-pose transformations and their inverses, we can walk up and down the paths of the tree allowing us to convert a point relative to any one joint into its representation relative to any other joint.

An Example: To make this a bit more concrete, let us consider an example in 2-dimensional space. Consider the pose shown in Fig. 38(a). (This is not a normal bind pose, since the elbow should not be bent, but it makes for a more interesting case.) Let i denote the shoulder joint, j the elbow joint, and k the hand joint. Consider a point v that lies two units beyond the model's index finger. Its homogeneous coordinates relative to the hand frame are $(2, 0, 1)$. (Since the x-axis points up.) Its coordinates relative to the elbow frame are $(6, 0, 1)$, and its coordinates relative to the shoulder frame are $(3, 6, 1)$ (see Fig. 38(b)). That is,

$$v_{[k]} = \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} \quad v_{[j]} = \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix} \quad v_{[i]} = \begin{pmatrix} 3 \\ 6 \\ 1 \end{pmatrix}.$$

Because k's coordinate frame lies 4 units along the x-axis relative to j's coordinate frame, the local pose transformation $T_{[j \leftarrow k]}$ (which maps a point in k's coordinate frame to j's coordinate frame) clearly increases the x-coordinate by 4 units. Thus:

$$T_{[j \leftarrow k]} = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We can easily check this, since

$$T_{[j \leftarrow k]} \cdot v_{[k]} = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix} = v_{[j]},$$

just as we expected.

Next, consider how to map a point in j's coordinate frame to its parent's frame i. Observe that the y-coordinate of the transformed point (its vertical distance) is the x-coordinate of the original point. Thus, the middle row of the matrix is $(1, 0, 0)$. The x-coordinate of the new point (its distance to the right of the shoulder) is 3 minus the old y-coordinate. Thus, the first row of the matrix is $(0, -1, 3)$. Therefore, the transformation that achieves this change of coordinates is

$$T_{[i \leftarrow j]} = \begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Again, we can check this, since

$$T_{[i \leftarrow j]} \cdot v_{[j]} = \begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 1 \end{pmatrix} = v_{[i]},$$

as we expected.

Now, to obtain the transformation $T_{[k \leftarrow i]}$, we multiply these two matrices (translating from k to j, then j to i)

$$T_{[i \leftarrow k]} = T_{[i \leftarrow j]} \cdot T_{[j \leftarrow k]} = \begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 1 \end{pmatrix}$$

Finally, to check this we have

$$T_{[i \leftarrow k]} \cdot v_{[k]} = \begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \\ 1 \end{pmatrix} = v_{[i]}.$$

Again, this is just what we expect to happen.

Of course, applying this in 3-dimensional space will involve handling 4×4 matrices and the associated rotation and translation matrices. While this would be much harder to do by hand, it can be done in by a similar process that is purely mechanical (and hence, easy to program).

Forward Kinematics: Next, suppose that in addition to knowing the local-pose transformations and their inverses, we also know the rotation transformations associated with the individual joints of the system. *Kinematics* (also called *forward kinematics*) is the problem of determining where a point is transformed as a result of these rotations.

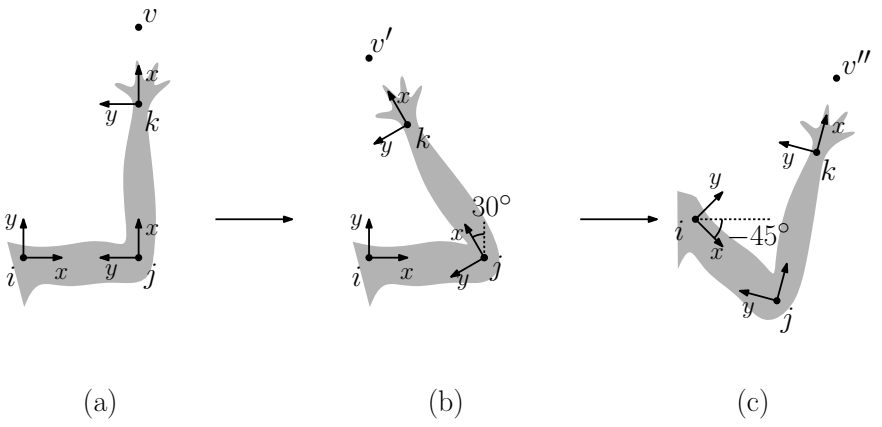


Fig. 39: Forward kinematics example.

We can apply our knowledge of rotation transformations and the local-pose transformations and their inverses to solve this problem. For example, recall the point v in our earlier example (see Fig. 39(a)) Suppose that the elbow is rotated by counterclockwise by 30° (see Fig. 39(b)), and then the shoulder is rotated clockwise by 45°, that is, counterclockwise by −45° (see Fig. 39(c)). The question that we want to consider, where is the point v mapped to as a result of these two rotations? Let v' be its position after the elbow rotation and let v'' be its position after both rotations.

Before getting to the answer, recall from our earlier lecture on affine geometry the rotation transformations in homogeneous coordinates:

$$\text{Rot}(30^\circ) \;=\; \begin{pmatrix} \cos 30^\circ & -\sin 30^\circ & 0 \\ \sin 30^\circ & \cos 30^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix} \;=\; \begin{pmatrix} \sqrt{3}/2 & -1/2 & 0 \\ 1/2 & \sqrt{3}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

and

$$\text{Rot}(-45^\circ) \;=\; \begin{pmatrix} \cos 45^\circ & -\sin 45^\circ & 0 \\ \sin 45^\circ & \cos 45^\circ & 0 \\ 0 & 0 & 1 \end{pmatrix} \;=\; \begin{pmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

We need to decide which frame to use as our reference frame. Let's use the shoulder joint i, since it is the most "global". (Generally, we would select the root of our skeleton tree.) We saw already how to compute $v_{[i]}$. Using this as a starting point, let's first consider the effect of the elbow rotation. Because the elbow rotation occurs about the elbow's coordinate frame, we first need to translate v into its representation with respect to j's frame (by multiplying by the inverse local-pose transformation $T_{[j \leftarrow i]}$). We then apply the 30° rotation about the elbow joint. Finally, we convert this representation back to the shoulder frame (by applying the local-pose transformation $T_{[i \leftarrow j]}$). Thus, we have

$$v'_{[i]} \;=\; T_{[i \leftarrow j]} \cdot \text{Rot}(30^\circ) \cdot T_{[j \leftarrow i]} \cdot v_{[i]}.$$

This yields a representation of v' relative to the shoulder frame. Since the second rotation is performed about the shoulder frame, we do not need to perform any change-of-coordinate transformation. We can just apply the rotation transformation directly. This yields:

$$v''_{[i]} \;=\; \text{Rot}(-45°) \cdot v'_{[i]}.$$

Putting both steps together, we have

$$v''_{[i]} \;=\; \text{Rot}(-45°) \cdot T_{[i \leftarrow j]} \cdot \text{Rot}(30°) \cdot T_{[j \leftarrow i]} \cdot v_{[i]}.$$

Top-down or Bottom-up? You might wonder why we did the elbow rotation first followed by the shoulder transformation. Does the order really matter? The issue is that our local-pose transformations have been built under the assumption that the model is in the bind pose, that is, none of the joints are rotated. If we were to have performed the shoulder rotation first, and then attempted to apply the inverse local-pose transformation $T_{[j \leftarrow i]}$ to convert the result from the shoulder's frame to the elbow frame, we would discover that this transformation is no longer correct. The reason is that the entire arm (and the elbow joint in particular) has moved into a new position, but $T_{[j \leftarrow i]}$ was defined based on its original position. To avoid this problem, the transformations should be applied in a *bottom-up manner*, first rotating the descendant nodes (e.g., wrist) and then working up to their ancestors (e.g., elbow and then shoulder).

Take-Away Lesson: I must acknowledge that implementing this by by hand would be a mess (especially in 3-space), but hopefully you get the idea. By using our local-pose transformations (and possibly their inverses), we can change to the coordinate frame where the rotation takes place, then apply the rotation, then translate back. While it would be messy to write down all the transformations, if we have precomputed the local pose transformations and their inverses, this can all be programmed in a straightforward manner by traversing the tree (in postorder) and performing simple matrix multiplications.

Lecture 10: Skeletal Animation and Skinning

Recap: Last time we introduced the principal elements of skeletal models and discussed forward kinematics. Recall that a skeletal model consists of a collection of joints, which have been joined into a rooted tree structure. Each joint of the skeleton is associated with a *coordinate frame* which specifies its position and orientation in space. Each joint can be rotated (subject to sum constraints). The assignment of rotation angles (or generally rotation transformations) to the individual joints defines the skeleton's *pose*, that is, its geometrical configuration in space. Joint rotations are defined relative to a default pose, called the *bind pose* (or *reference pose*).

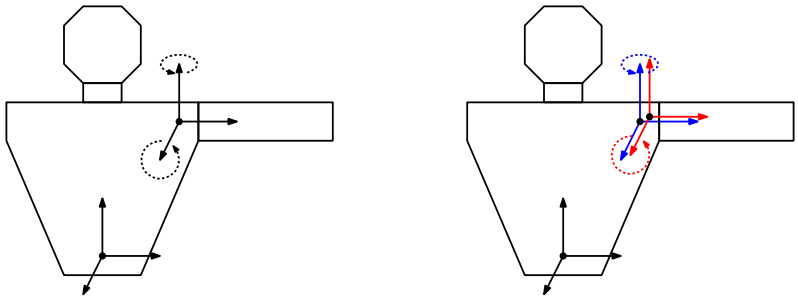
Last time, we showed how to determine the skeleton's configuration from a set of joint angles. This is called *forward kinematics*. (In contrast, *inverse kinematics* involves the question of determining how to set the joint angles to achieve some desired configuration, such as grasping a door knob.) Today we will discuss how animation clips are represented, how to cover these skeletons with "skin" in order to form a realistic model, and how to move the skin smoothly as part of the animation process.

Local and Global Pose Transformations: Recall from last time that, given a joint j (not the root), its parent joint is denoted $p(j)$. We assume that each joint j is associated with two transformations, the *local-pose transformation*, denoted $T_{[p(j) \leftarrow j]}$, which converts a point in j's coordinate system to its representation in its parent's coordinate system, and the *inverse local-pose transformation*, which reverses this process. (These transformations may be represented explicitly, say, as a 4×4 matrix in homogeneous coordinates, or implicitly by given a translation vector and a rotation, expressed, say as a quaternion.)

Recall that these transformations are defined relative to the bind pose. By chaining (that is, multiplying) these matrices together in an appropriate manner, for any two joints j and k, we can generally compute the transformation $T_{[k \leftarrow j]}$ that maps points in j's coordinate frame to their representation in k's coordinate frame (again, with respect to the bind pose.)

Let M (for "*Model*") denote the joint associated with the root of the model tree. We define the *global pose transformation*, denoted $T_{[M \leftarrow j]}$, to be the transformation that maps points expressed locally relative to joint j's coordinate frame to their representation relative to the model's global frame. Clearly, $T_{[M \leftarrow j]}$ can be computed as the product of the local-pose transformations from j up to the root of the tree.

Meta-Joints: One complicating issue involving skeletal animation arises from the fact that different joints have different numbers of degrees of freedom. A clever trick that can be used to store joints with multiple degrees of freedom (like a shoulder) is to break the into two or more separate joints, one for each degree of freedom. These *meta-joints* share the same point as their origin (that is, the translational offset between them is the zero vector). Each meta-joint is responsible for a single rotational degree of freedom. For example, for the shoulder one joint might handle rotation about the vertical axis (left-right) and another might handle rotation about the forward axis (up-down) (see Fig. 40). Between the two, the full spectrum of two-dimensional rotation can be covered. This allows us to assume that each joint has just a single degree of freedom.



(a) (b)

Fig. 40: Using two meta-joints (b) to simulate a single joint with two degrees of freedom (a).

Animating the Model: There are a number of ways to obtain joint angles for an animation. Here are a few:

Motion Capture: For the common motion of humans and animals, the easiest way to obtain animation data is to capture the motion from a subject. Markers are placed on a subject, who is then asked to perform certain actions (walking, running, jumping, etc.) By tracking the markers using multiple cameras or other technologies, it is possible to reconstruct the positions of the joints. From these, it is simple exercise in linear algebra to determine the joint angles that gave rise to these motions.

Motion capture has the advantage of producing natural motions. Of course, it might be difficult to apply for fictitious creatures, such as flying dragons.

Key-frame Generated: A design artist can use animation modeling software to specify the joint angles. This is usually done by a process called *key framing*, where the artists gives a detailed layout of the model at certain "key" instances in over the course of the animation, called *key frames*. (For example, when animating a football kicker, the artist might include the moment when the leg starts to swing forward, an intermediate point in the swing, and the point at which

the leg is at its maximum extension.) An automated system can then be used to smoothly interpolate the joint angles between consecutive key frames in order to obtain the final animation. (The term "frame" here should not be confused with the use of term "coordinate frame" associated with the joints.)

Goal Oriented/Inverse kinematics: In an ideal world, an animator could specify the desired behavior at a high level (e.g., "a character approaches a table and picks up a book"). Then the physics/AI systems would determine a natural-looking animation to achieve this. This is quite challenging. The reason is that the problem is under-specified, and it can be quite difficult to select among an infinite number of valid solutions. Also, determining the joint angles to achieve a particular goal reduces to a complex nonlinear optimization problem.

Representing Animation Clips: In order to specify an animation, we need to specify how the joint angles or generally the joint frames vary with time. This can result in a huge amount of data. Each joint that can be independently rotated defines a *degree of freedom* in the specification of the pose. For example, the human body has over 200 degrees of freedom! (It's amazing to think that our brain can control it all!) Of course, this counts lots of fine motion that would not normally be part of an animation, but even a crude modeling of just arms (not including fingers), legs (not including toes), torso, neck involves over 20 degrees of freedom.

As with any digital signal processing (such as image, audio, and video processing), the standard approach for efficiently representing animation data is to first *sample* the data at sufficiently small time intervals. Then, use some form of interpolation technique to produce a smooth *reconstruction* of the animation. The simplest manner to interpolate values is based on *linear interpolation*. It may be desireable to produce smoother results by applying more sophisticated interpolations, such as quadratic or cubic spline interpolations. When dealing with rotated vector quantities, it is common to use *spherical interpolation*.

In Fig. 41 we give a graphical presentation of a animation clip. Let us consider a fairly general set up, in which each pose transformation (either local or global, depending on what your system prefers) is represented by a 3-element translation vector (x, y, z) indicating the joint frame's position and a 4-element quaternion vector (s, t, u, v) to represent the frame's rotation. Each row of this representation is a sequence of scalar values, and is called a *channel*.

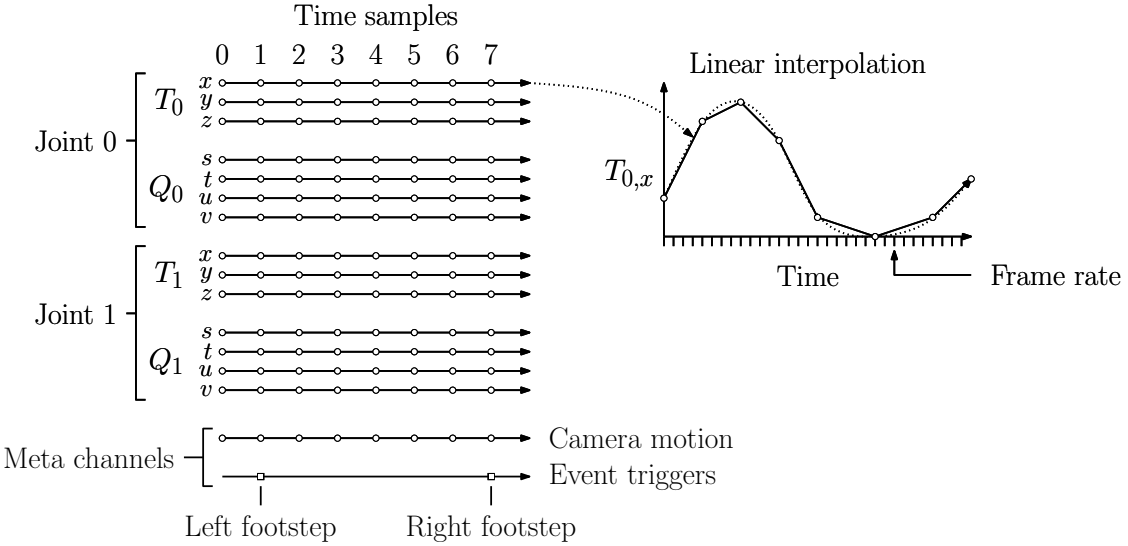


Fig. 41: An uncompressed animation stream.

It is often useful to add further information to the animation, which are not necessarily related to the rendering of the moving character. Examples include:

Event triggers: These are discrete signals sent to other parts of the game system. For example, you might want a certain sound playback to start with a particular event (e.g., footstep sound), a display event (e.g., starting a particle system that shows a cloud of dust rising from the footstep), or you may want to trigger a game event (e.g., a non-playing character ducks to avoid a punch).

Continuous information: You may want some process to adjust smoothly as a result of the animation. An example would be having the camera motion being coordinated with the animation. Another example would be parameters that continuously modify the texture coordinates or lighting properties of the object. Unlike event triggers, such actions should be smoothly interpolated.

This auxiliary information can be encoded in additional streams, called *meta-channels* (see Fig. 41). This information will be interpreted by the game engine.

Skinning and Vertex Binding: Now that we know how to specify the movement of the skeleton over time, let us consider how to animate the skin that will constitute the drawing of the character. The first question is how to represent this skin. The most convenient representation from a designer's perspective, and the one that we will use, is to position the skeleton in the reference pose and draw the skin around the resulting structure (see Fig. 42(a)).

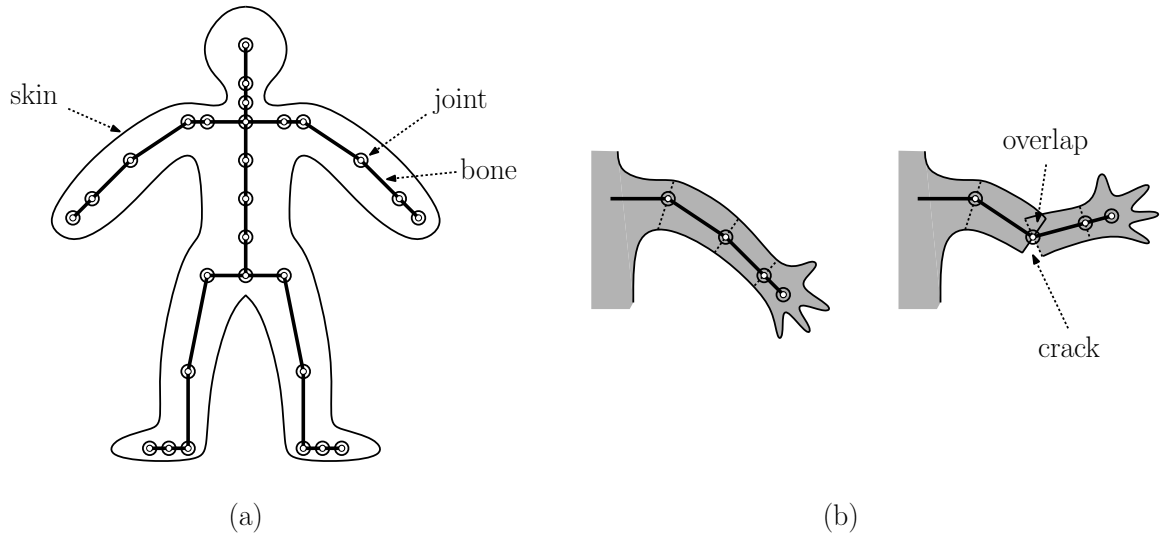


Fig. 42: (a) Binding skin to a skeletal model in the reference pose and (b) cracks and overlaps.

In order that the skin move smoothly along with the skeleton, we need to associate, or *bind*, vertices of the mesh to joints of the system, so that when the joints move, the skin moves as well. (This is the reason that the reference pose is called the bind pose.)

If we were to bind each vertex to a single joint, then we would observe *cracks* and *overlaps* appearing in our skin whenever neighboring vertices are bound to two different joints that are rotated apart from one another.

Dealing with this problem in a realistic manner will be too difficult. (The manner in which the tissues under your skin deform is a complex anatomical process. Including clothing on top of this makes for a tricky problem in physics as well.) Instead, our approach will be to find a heuristic solution that will be easy to compute and (hopefully) will produce fairly realistic results.

Multiple Joint Binding with Weights: The trick is to allow each vertex of the mesh to be bound to multiple joints. When this is done, each joint to which a vertex is bound is assigned a *weighting factor*, that specifies the degree to which this joint influences the movement of the vertex.

For example, the mesh vertices near your elbow will be bound to both the shoulder joint (your upper arm) and the elbow joint (your lower arm). As we move down the arm, the shoulder weight factor diminishes while the elbow weight factor increases. Consider for example, consider a vertex v that is located slightly above the elbow joint (see Fig. 43(a)). In the bind pose, let v_1 and v_2 denote its positions relative to the shoulder and elbow joint frames, respectively. Since this point is slightly above the elbow joint, we give a slightly higher weight with respect to the shoulder joint. Suppose that the weights are $w_1 = \frac{3}{4}$ and $w_2 = \frac{1}{4}$.

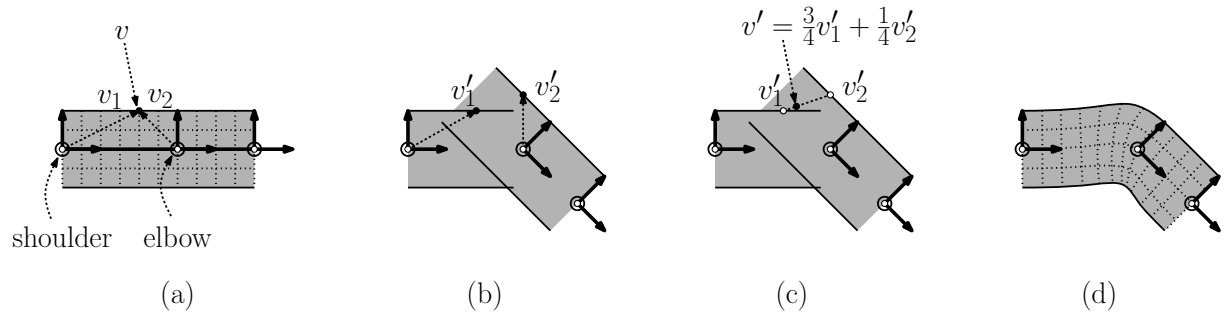


Fig. 43: Skinning with multiple joint bindings.

Now, suppose that we bend both of these joints. Let v_1' and v_2' denote the respective images of the points v_1 and v_2 after the rotation. (They will be in the same position relative to their respective joints, but their absolute positions in space have changed. See Fig. 43(b).) We use our weight factors to interpolate between these two positions, so the final position of the vertex is at the point $v' = \frac{3}{4}v_1' + \frac{1}{4}v_2'$ (see Fig. 43(c)). Because of the smooth variations in weights, the vertices of the mesh will form a smooth interpolation between the upper arm and the lower arm (see Fig. 43(d)). It may not be physically realistic, but it is a reasonable approximation, and is very easy to compute.

To make this more formal, we assume that each vertex of the mesh is associated with:

Joints: A list of *joint indices* to which this mesh vertex is bound

Weights: A corresponding list of *weights*, which determine the influence of each joint on this vertex. These weights are assumed to be nonnegative and sum to 1.

The number of joints to which a typical vertex is bound is typically small, e.g., from two to four. Good solid modelers provide tools to automatically assign weights to vertices, and designers can query and adjust these weights until they produce the desired look.

The above binding information can be incorporated into the mesh information already associated with a vertex: the (x, y, z)-coordinates of its location (with respect to the model coordinate system), the (x, y, z)-coordinates of its normal vector (for lighting and shading computation), and its (s, t) texture coordinates.

Moving the Joints: In order to derive the computations needed to move a vertex from its initial position to its final position, let's start by introducing some notation. First, recall that our animation system informs us at any time t the current angle for any joint. Abstractly, we can think of this joint angle as providing a *local rotation*, $R_j^{(t)}$, that specifies how joint j has rotated. For example, if the joint has undergone a rotation through an angle θ about some axis, then $R_j^{(t)}$ would be represented by a

rotation matrix by angle θ about this same axis. (The analysis that we perform below works under the assumption that $R_j^{(t)}$ is any affine transformation, not necessarily a rotation.)

Consider a vertex v of the mesh. Let $v^{(0)}$ denote v's position in the initial reference pose, and let $v^{(t)}$ denote its position at the current time. We assume that this information is provided to us from the solid modeler. We can express v in one of two coordinate systems. Let v_j denote its initial position with respect to j's coordinate frame and let v_M denote its initial position with respect to the model's frame. (Throughout this section, we will assume that v is associated with the single joint j, rather than blended between multiple joints.) Given the above local rotation transformation, we have $v_j^{(t)} = R_j^{(t)} v_j^{(0)}$. (Because we assume that v rotates with frame j, its representation with respect to j does not really change over time. Instead, think of $v_j^{(t)}$ as its representation relative to the joint's unrotated reference frame.)

Recall that $T_{[M \leftarrow j]}$ denotes the bind-pose transformation, which we introduced earlier. In general, let $T_{[M \leftarrow j]}^{(t)}$ denote the transformation that maps the vertex $v_j^{(0)}$ (which is in j's coordinate frame at time 0) to $v_M^{(t)}$ (which is in the model's frame at any time t).

Let's consider how to compute $T_{[M \leftarrow j]}^{(t)}$. As we did earlier, we'll build this up in a series of stages, by converting a point from its frame to its parent, then its grandparent, and so on until we reach the root of the tree. To map a vertex at time 0 from its frame to its parent's frame at time t we need to do two things. First, we apply the local joint rotation that takes us from time 0 to time t with respect to j's local frame, and then we transform this to j's parent's frame. That is, we need to first apply $R_j^{(t)}$ and then $T_{[p(j) \leftarrow j]}$. Let us define $T_{[p(j) \leftarrow j]}^{(t)}$ to be the product $T_{[p(j) \leftarrow j]} R_j^{(t)}$. We now have

$$v_{p(j)}^{(t)} \;=\; T_{[p(j) \leftarrow j]} v_j^{(t)} \;=\; T_{[p(j) \leftarrow j]} R_j^{(t)} v_j^{(0)} \;=\; T_{[p(j) \leftarrow j]}^{(t)} v_j^{(0)}.$$

To obtain the position of a vertex associated with j's coordinate frame, we need only compose these matrices in a chain working back up to the root of the tree. We apply a rotation, convert to the coordinate frame of the parent joint, apply the parent rotation, convert to the coordinates of the grandparent joint, and so on. Suppose that the path from j to the root is $j = j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_m = M$, then transformation we desire is

$$\begin{aligned} T_{[M \leftarrow j]}^{(t)} \;&=\; \prod_{i=1}^{m-1} T_{[j_{m-i+1} \leftarrow j_{m-i}]}^{(t)} \;=\; T_{[j_m \leftarrow j_{m-1}]}^{(t)} \ldots T_{[j_3 \leftarrow j_2]}^{(t)} T_{[j_2 \leftarrow j_1]}^{(t)} \\ &=\; T_{[j_m \leftarrow j_{m-1}]} R_{j_{m-1}}^{(t)} \ldots T_{[j_3 \leftarrow j_2]} R_{j_2}^{(t)} T_{[j_2 \leftarrow j_1]} R_{j_1}^{(t)}. \end{aligned}$$

We refer to this as the *current-pose transformation*, since it tells where joint j is at time t relative to the model's global coordinate system. Observe that with each animation time step, all the matrices $R_j^{(t)}$ change, and therefore we need to perform a full traversal of the skeletal tree to compute $T_{[M \leftarrow j]}^{(t)}$ for all joints j. Fortunately, a typical skeleton has perhaps tens of joints, and so this does not represent a significant computational burden (in contrast to operations that need to be performed on each of the individual vertices of a skeletal mesh).

Putting it all Together: Finally, let's consider how to apply blended skinning together with the dynamic pose transformations. This will tell us where every vertex of the mesh is mapped to in the currrent animation. We assume that for the current-pose transformation $T_{[M \leftarrow j]}^{(t)}$ has been computed for all the joints, and we assume that each vertex v is associated with a list of joints and associated weights. Let $J(v) = \{j_1, \ldots, j_k\}$ be the joints associated with vertex v, and let $W(v) = \{w_1, \ldots, w_k\}$ be the associated weights. Typically, k is a small number, ranging say from 1 to 4. For i running from 1 to k, our approach will be compute the coordinates of v relative to joint j_i, then apply the current-pose

transformation for this joint in order to obtain the coordinates of v relative to the (global) model frame. This gives us k distinct points, each behaving as if it were attached to a different joint. We then blend these points together, to obtain the desired result.

Recall the inverse bind-pose transformation $T_{[j \leftarrow M]}$, which maps a vertex v from its coordinates in the model frame to its coordinates relative to j's coordinate frame. This transformation is defined relative to the bind pose, and so is applied to vertices of the original model, prior to animation. Once we have the vertex in its representation at time 0 relative to joint j, we can then apply the current-pose transformation $T^{(t)}_{[M \leftarrow j]}$ to map it to its current position relative to the model frame. If v was bound to a single joint j, we would have

$$v_M^{(t)} \;=\; T^{(t)}_{[M \leftarrow j]} T_{[j \leftarrow M]} v_M^{(0)}.$$

Let us define $K_j^{(t)} = T^{(t)}_{[M \leftarrow j]} T_{[j \leftarrow M]}$. This is called the *skinning transformation* for joint j. Intuitively, it tells us where vertex v is mapped to under at time t of the animation assuming that it is fixed to joint j.

Now, we can generalize this to the case of blending among a collection of vertices. Recall that v has been bound to the joints of $J(v)$. Its blended position at time t is given by the weighted sum of the image of the skinning transformed vertices $K_{j_i}^{(t)} v_M^{(0)}$ for each joint j_i to which v is bound:

$$v_M^{(t)} \;=\; \sum_{j_i \in J(v)} w_i K_{j_i}^{(t)} v_M^{(0)}.$$

This then is the final answer that we seek. While it looks like a lot of matrix computation, remember each vertex is associated with a constant number of joints, and each joint is typically at constant depth in the skeletal tree. Once these matrices are computed, they may be stored and reused for all the vertices of the mesh.

A simple example of this is shown in Fig. 44. In Fig. 44(a) we show the reference pose. In Fig. 44(b), we show what might happen if every vertex is bound to a single joint. When the joint flexes, the vertices at the boundary between to bones crack apart from each other. In Fig. 44(c) we have made a very small change. The vertices lying on the seam between the two pieces have been bound to both joints j_1 and j_2, each with a weight of 1/2. Each of these vertices is effectively now placed at the midpoint of the two "cracked" copies. The result is not very smooth, but it could be made much smoother by adding weights to the neighboring vertices as well.

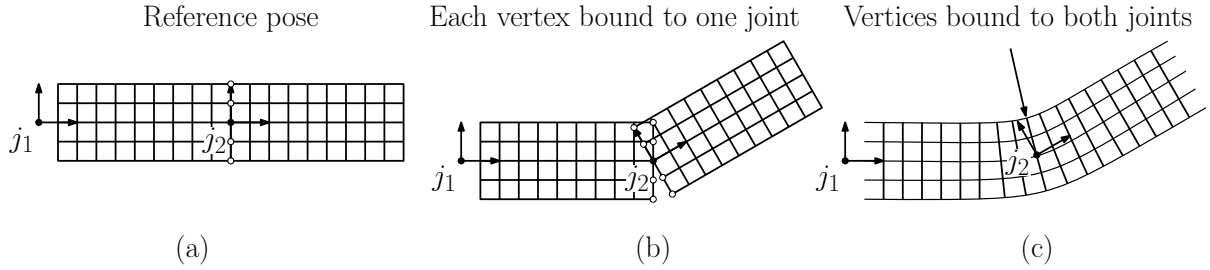


Fig. 44: A simple example of blended skinning.

It is worth making a few observations at this point about the storage/computational requirements of this approach.

Matrix palette: In order to blend *every* vertex of the model, we need only one matrix for each joint of the skeleton, namely the skinning matrices $K_j^{(t)}$. While a skeleton may have perhaps tens of

joints, it may have hundreds of vertices. Assuming that the joint are indexed by integers, the palette can be passed to the GPU as an array of matrices.

Vertex information: Each vertex is associated with a small list of joint indices and weights. In particular, we do not need to associate entire matrices with individual vertices.

From the perspective of GPU implementation, this representation is very efficient. In particular, we need only associate a small number of scalar values with each vertex (of which there are many), and we store a single vertex with each joint (or which there are relatively few). In spite of the apparent tree structure of the skeleton, everything here can be represented using just simple arrays. Modern GPUs provide support for storing matrix palettes and performing this type of blending.

Shortcomings of Blended Skinning: While the aforementioned technique is particularly well suited to efficient GPU implementation, it is not without its shortcomings. In particular, if joints are subjected to high rotations, either in flexing or in twisting, the effect can be to cause the skin to deform in particular unnatural looking ways (see Fig. 45).

(a) (b)

Fig. 45: Shortcomings of vertex blending in skinning: (a) Collapsing due to bending and (b) collapsing due to twisting.

Lecture 11: Motion Panning: Navigation Meshes

Motion For the next few lectures we will discuss one of the major issues in the design of AI and algorithms in games, namely planning the motion for non-player characters (NPCs). Motion is a remarkably complex topic, which can range from trivial (e.g., computing a straight-line path between two points in the plane) to very complex tasks, such as:

- Planning the coordinated motion of a group of agents who wish to move to a specified location amidst many obstacles
- Planning the motion of an articulated skeletal model subject to constraints, such as maintaining hand contact with a door handle or avoiding collisions while passing through through a narrow passageway
- Planning the motion of a character while navigating through a dense crowd of other moving people (who have their own destinations), or planning motion either to evade or to pursue the player
- Planning ad hoc motions, like that of a mountain climber jumping over boulders or climbing up the side of a cliff

Historically, much of the initial development of techniques in this area arose from other fields, such as robotics, autonomous vehicle navigation, and computational geometry. Game designers have some advantages in solving these problems, since the environment in which the NPCs move is under the control of the game designer. This means that a game designer can simplify motion planning by creating additional free space in the environment, thus making it easier to plan motion. (In contrast,

the designer of an autonomous vehicle cannot remodel the world to make roads wider.) Nonetheless, the techniques that we will present for doing motion planning are broadly applicable, even though they may not need to be applied in their full generality.

Overview: Given the diverse nature of motion planning problems it is not surprising that the suite of techniques is quite large. We will take the approach of describing a few general ideas, that can be applied (perhaps with modifications) across a broad range of problems. These involve the following elements:

Single-object motion:

From objects to points: Methods such as *configuration spaces* can be applied to reduce the problem of moving a complex object (or assembly of objects) with multiple degrees of freedom (DOFs) to the motion of a single point through a multi-dimensional space.

Discretization: Methods such as *waypoints*, *roadmaps*, and *navigation meshes* are used to reduce the problem of moving a point in continuous space to computing a path in discrete graph-like structure.

Shortest paths: This includes efficient algorithms for computing shortest paths, updating them as conditions change, and representing them for later access.

Multiple-object motion:

Flocking: There exist methods for planning basic flocking behavior (as with birds and herding animals) and applications to simulating crowd motion.

Purposeful crowd motion: Techniques such as *velocity obstacles* are used for navigating a single agent from an initial start point to a desired goal point through an environment of moving agents.

Guarding and Pursuit/Evasion: These include methods for solving motion-planning tasks where one agent is either hunting for or attempting to elude detection by the player or another agent.

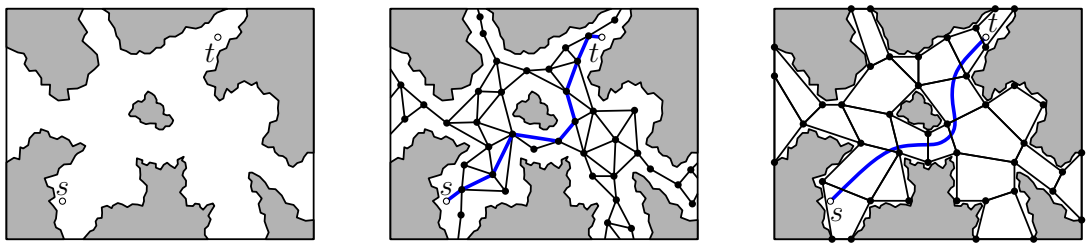
Like many of the topics we have covered this semester, we could easily devote an entire course to this one topic, but we will instead try to sample some of the key ideas. In this lecture, we will focus on one of the most widely used concepts from this area, called a *navigation mesh*. This is the principal support feature that the Unity Engine provides for character navigation.

Navigation Meshes: A *navigation mesh* (or *NavMesh*) is a data structure used to model free-space, particularly for an agent that is moving along a two-dimensional surface. (Such a surface is formally referred to as a *two-manifold*). A navigation mesh is a spatial subdivision (more specifically, a cell-complex) whose faces are convex polygons, usually triangles. Each face of the mesh behaves like a node in a graph, and two nodes are joined by an edge if the associated faces share a common edge. Because the faces are convex, any point from inside one face can be reached by a straight line from any other point inside the same face. As with a waypoint system, there is an underlying graph which can be used for computing paths, but by storing the cell complex, the paths computed are not constrained to follow the waypoints.

For example, in Fig. 46(a) we show a possible workspace. In (b) we show a possible waypoint system, and in (c) we show a possible navigation mesh. We show a possible path using each representation between a start point s and destination t.

Because they provide a more faithful representation of the underlying free-space geometry, navigation meshes have a number of advantages over waypoint-based methods:

- They are capable of generating shorter and more natural paths than traditional waypoint methods.



(a) (b) (c)

Fig. 46: (a) An environment, (b) a possible waypoint-based roadmap, and (c) a possible navigation map.

- Waypoint methods can generate an excessive number of points to compensate for their shortcomings, and so navigation meshes can be considerably more space efficient.
- They can be used to plan the movement of multiple (spatially separate) agents, such as a group of people walking abreast of each other. (Note that a waypoint system would need to plan their motion in a single-file line.)
- It is easier to incorporate changes to the environment (such as the insertion, removal, or modification of obstacles).
- A wide variety of pathfinding algorithms can be modified and optimized for using navigation meshes.

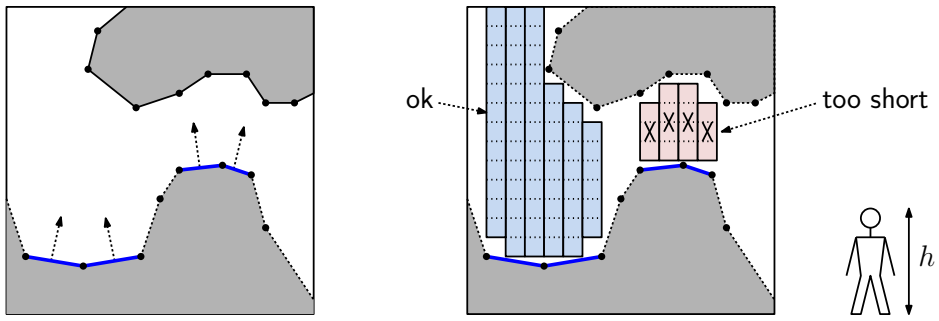
Of course, because are more complicated than waypoint-based methods, there are also disadvantages to the use of navigation meshes:

- They are not as easy to generate as waypoint-based methods.
- Because navigation meshes require a more complex representation of geometry of the scene, the associated pathfinding algorithms are more complex and may take longer to execute.
- They are difficult to generate by hand, and automated generation systems are relatively complex.

Automatic Generation of Navigation Meshes: If the environment is simple, the navigation mesh can be added by the artist who generated the level. Of course, we cannot do this for environments that imported from other sources. If the level is quite large, it is often possible to generate a navigation mesh fairly easily. (Consider for example the sidewalks and roads of an urban scene.) In less structured settings, it is often desirable to generate the navigation mesh automatically. How is this done?

There are many possible approaches to building navigation meshes. We will discuss (a simplified version of) a method due to Mikko Mononen. Let's begin with a few assumptions. First, we assume that the moving agents will be walking along a 2-dimensional surface. This surface need not be flat, and it may contain architectural elements such as ramps, tunnels, and stairways. We will assume that the input is expressed as a polygonal mesh of the world. We will also assume that our moving agent is a walking/running humanoid, and hence can be coarsely modeled as a vertical line segment or a thin cylinder with a vertical axis that translates along this surface.

Find the walkable surfaces: Since we assume that our agent is walking, a polygon is suitable for walking on if (1) the polygon is roughly parallel to the ground, and (2) there is sufficient headroom about this polygon for our agent to walk. Such a polygon is said to be *walkable*. We can identify the polygons that satisfy the first condition by computing the angle between the polygon's (outward pointing) normal vector and the vertical unit vector (see Fig. 47(a)). This angle can be computed through the use of the dot-product operator, as described in earlier lectures.

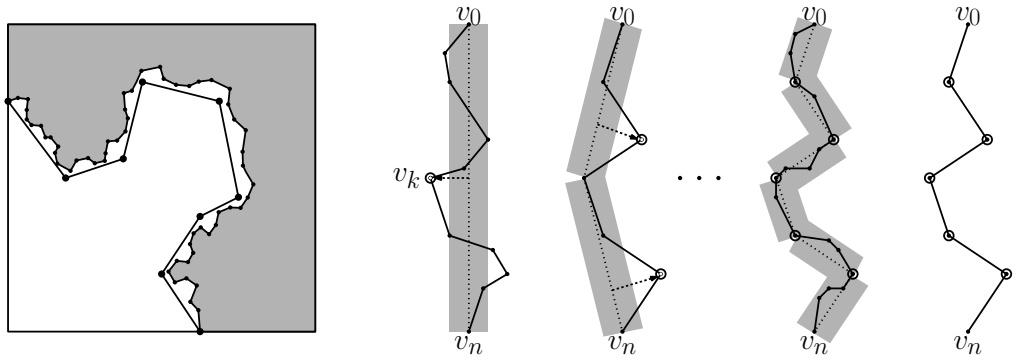


(a) (b)

Fig. 47: Walkable surface (side view): (a) Identifying "flat" polygons and (b) voxel method for determining sufficient headroom.

In order to test the the second property, let h denote the height of the agent. Mononen suggests the follows very fast and simple approach. First, voxelize the 3-dimensional space using a grid of sufficient resolution. (For example, the width of the grid should be proportional to the narrowest gap the agent can slip through.) For each polygon that passes condition (1), we determine how many voxels lie immediately above this triangle until hitting the next obstacle (see Fig. 47(b)). (Note that this includes *all* the obstacle polygons, not just the ones that are nearly level.)

Simplify the Polygon Boundaries: Consider the boundary of the walkable surface. (This the boundary between walkable polygons and polygons that are not walkable.) This boundary may generally consist of a very complex polygonal curve with many vertices. We next approximate this curve by a one have much fewer vertices (see Fig. 48(a)).



(a) (b)

Fig. 48: The Ramer-Douglas-Peucker Algorithm.

There is a standard algorithm for simplifying polygonal curves, called the Ramer-Douglas-Peucker[6] Algorithm. Here is how the algorithm works. First, let δ denote the maximum error that we will allow in our approximation. Suppose that the curve runs between two points v_0 and v_n. If the entire curve fits within a pair of parallel lines at distance δ on either side of the line segment $\overline{v_0 v_n}$,

[6]The algorithm was discovered independently by Urs Ramer in 1972 and by David Douglas and Thomas Peucker in 1973. Ramer published his result in the computer graphics community and Douglas and Peucker published theirs in the cartography community.

then we stop. Otherwise, we find the vertex v_k at maximum distance from this line segment. We add a new vertex at v_k, and then we recursively repeat the algorithm on the two sub-curves $\overline{v_0 v_k}$ and $\overline{v_k v_n}$ (see Fig. 48(b)).

Notice that the Ramer-Douglas-Peucker algorithm is not quite what we desired, because it generates a curve that lies on both sides of the original curve. Some modifications are necessary in order to produce a curve that has the property of lying entirely on one side of the original curve, but the principle is essentially the same.

Triangulating the Simplified Polygon: After simplification, we have a collection of polygons, each of which may contain some number of holes (see Fig. 49(a)). The final step is to generate a triangulation of this polygon. Ideally, we would like to have a triangulation in which the triangular elements are relatively "fat." Mononen suggests a very simple heuristic for achieving such a triangulation. (Again, I'll present a variant of his approach.)

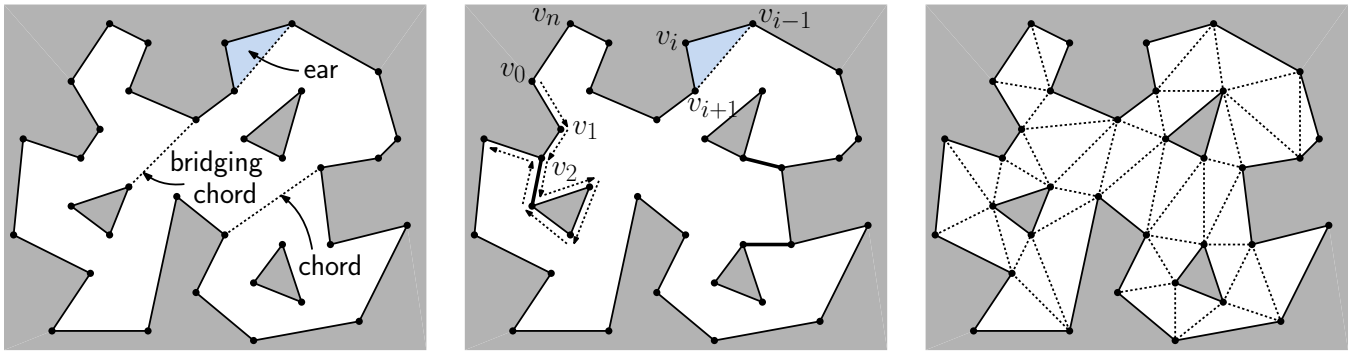


Fig. 49: Triangulating the simplified polygon.

Before presenting the algorithm, let's give a couple of definitions. A line segment that connects two vertices of the polygon and that lies entirely within the interior of the polygon is called a *chord* (see Fig. 49(a)). A chord that connects two holes together or that connects a hole with the outer boundary is called a *bridging chord*. A chord that connects two vertices that share common neighboring vertex cuts of a single triangle from the polygon. This triangle is called an *ear*.

Here is the algorithm:

Bridge the holes: First, connect each hole of the polygon either to another hole or two the boundary of the outer polygon using bridging chords. Repeatedly select the bridging chord of minimum length, until all the holes are connected to the outer boundary. (If there are h holes, this will involve exactly $h - 1$ bridging chords.)

By thinking of each bridging chord a consisting of two edges, one leading into the hole and one leading out, the resulting boundary now consists of one connected component, which we can treat as a polygon without any holes. Number the vertices $v_0, \ldots, v_n$ in counterclockwise order around this polygon (see Fig. 49(b)).

Remove Ears: If the polygon consists of only three vertices, then we are done. Otherwise, find three consecutive vertices v_{i-1}, v_i, v_{i+1} such that $\overline{v_{i-1} v_{i+1}}$ is a chord. The triangle $\triangle v_{i-1}, v_i, v_{i+1}$ is an ear. Among all the possible ears, select the one whose chord is of minimum length. Cut this ear off (ouch!) by adding the chord $\overline{v_{i-1} v_{i+1}}$. The remaining polygon has one fewer vertex. Repeat the process recursively on this polygon, until only three vertices remain. The union of all the removed ears is the final triangulation (see Fig. 49(c)).

By the way, this is but one way to triangulate a polygon with holes. There are many algorithms that are significantly more efficient than this one (from the perspective of worst-case running

time). The best such algorithms run in time $O(n \log n)$. If the polygon has no holes, it is possible to triangulate it in $O(n)$ time, but the algorithm is quite complicated, and the $O(n \log n)$ time algorithm is more widely used.

Computing Paths in Polygonal Domains: The final triangulation is the desired navigation mesh. The last detail that remains is how to compute shortest paths in the navigation mesh. In our next lecture we will present algorithms for computing shortest paths in geometric graphs. Assuming that we have such an algorithm, let us next consider how to generalize a graph-based shortest path to a mesh-based shortest path.

Computing a the exact shortest path on a mesh or within a polygonal domain can be solved efficiently in theory, but the algorithm is a bit complicated. We will propose a simpler approach, which produces a good approximate solution.

Discretize: First, distribute a small number of vertices along the each edge of the mesh (see Fig. 50(a) and (b)). Next, for each face of the mesh, form a complete graph by connecting these vertices together (see Fig. 50(a) and (c))).

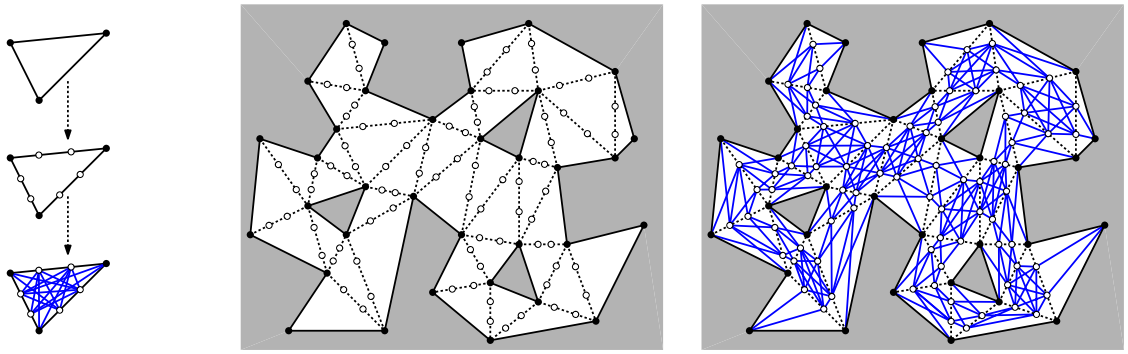


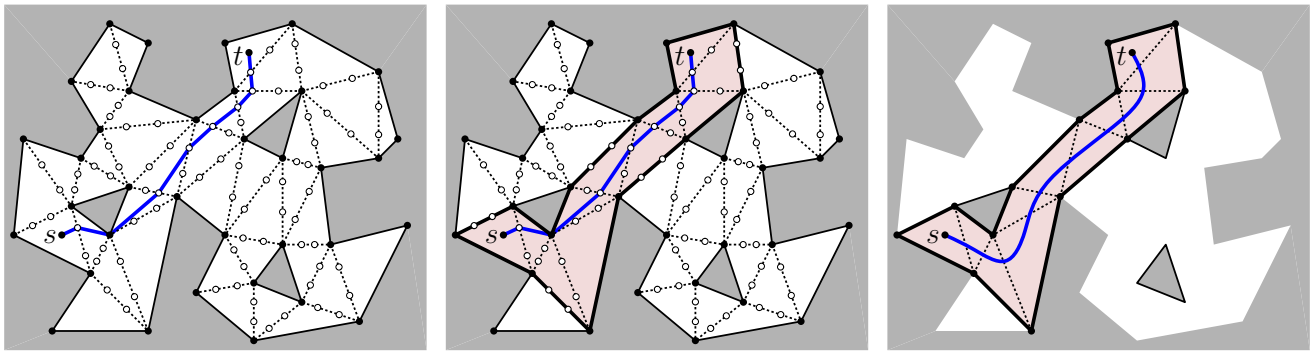
Fig. 50: Discretizing a polygonal domain for computing shortest paths.

Shorten/Smooth: Compute the shortest path in the resulting graph using any graph-based method (see Fig. 51(a)). Identify a simple polygon (without holes) by combining all the faces of the mesh that this path passes through (see the shaded polygon in Fig. 51(b)). Finally, apply any further optimizations (such as shortening or smoothing) to the path as desired, subject to the constraint that the path does not leave this shaded polygon (see Fig. 51(c)). Because this polygon has no holes, it is much easier to perform the desired optimizations.

Lecture 12: Representing Meshes and the Doubly-Connected Edge List

Meshes and Some Terminology: In the previous lecture, we discussed navigation meshes. This structure provided a way to represent the "walkable surface" of a domain as a triangulated mesh. In order to walk from one mesh element to the next, we need a representation that allows us to easily identify neighboring mesh elements. In this lecture, we will describe such a data structure, which is called the *doubly-connected edge list*, or *DCEL* for short.

When representing meshes, it is common to distinguish between two facets of the representations, *geometry* and *topology*. The geometric information involves the locations of objects, such as the coordinates of vertices or the equations of its faces. The topological information involves how the elements



(a) (b) (c)

Fig. 51: (a) The shortest path between s and t, (b) the polygon containing the path (shaded in red), and (c) the final smoothed path.

of the mesh are connected together. This also involves issues such as whether there are holes or cavities within the model.

In the field of topology, a surface patch is called *2-manifold* (see Fig. 52(a)). A defining property of 2-manifolds is that in any sufficiently small local neighborhood surrounding any interior point of the surface looks (up to stretching) like a small circular disk. (See Fig. 52(b) for examples of violations.) We our manifolds to have boundaries, and they may generally contain holes. Intuitively, you can think of a 2-manifold (with boundary) to be a very thin rubber sheet, to which someone may have cut out holes.

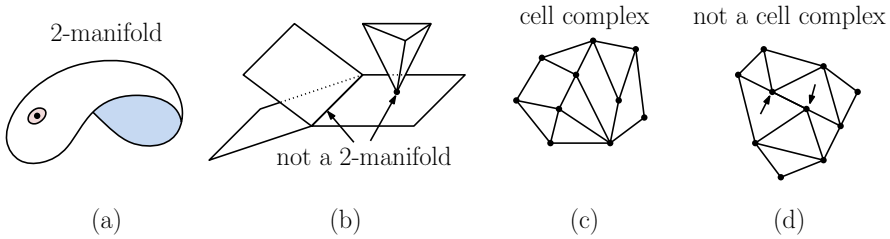


Fig. 52: 2-Manifolds and cell complexes.

In order to represent surfaces, it is common to break them up into small polygonal elements, which are typically triangles. (Triangles are nice, because are always convex and always planar. In general, a polygonal in 3-dimensional space that is built using four or more vertices might fail either of these properties.) When two triangles of the mesh are joined together, they are joined edge-to-edge (see Fig. 52(c)). This implies, in particular, that a vertex of one triangle will not appear in the interior of an edge or a face of another triangle (as in Fig. 52(d)). Such a decomposition is called a *cell complex* or (when triangles only are involved) a *simplicial complex*. The DCEL data structure is used for representing cell complexes on 2-manifold surfaces.

The DCEL: A cell complex subdivides a mesh into three types of elements, vertices (0-dimensional), edges (1-dimensional), and faces (2-dimensional). Thus, we can encode the topological information of a mesh as an undirected graph. For the purposes of unambiguously distinguishing left from right, it will be convenient to represent each undirected edge by two oppositely directed edges, called *half-edges*. An edge directed from u to v is said to have u as its *origin* and v as its *destination*.

For now, let us make the simplifying assumption that the faces of the mesh not have holes inside of

them. (This assumption can always be satisfied by introducing some number of *bridging edges* that join the outer boundary of the face to each of the holes. With this assumption, it may be assumed that the edges edges each face form a single cyclic list.

The DCEL consists of three principal elements, vertices, edges, and faces. For each, we store the following information:

Vertex: Each vertex stores its spatial coordinates, along with a reference to any single incident half-edge that has this vertex as its origin, v.incident.

Face: Each face f stores a reference to a single half-edge for which this face is the incident face, f.incident. (Such a half-edge will be directed counterclockwise about the face.)

Edge: Each half-edge (u, v) is naturally associated with two vertices, its origin u and its destination v, its twin half-edge (v, u), and its two incident faces, one to the left and one to the right. To distinguish left from right, let us assume that our mesh has an outward facing side and an inward side. (This works fine for most meshes that arise in solid modeling, since they enclose solid bodies. There are exceptions, however, such as a Mobius strip.) Consider the half-edge (u, v) and imagine that you are standing in the middle of the edge on the outer side of the mesh with u behind you and v in front. The face to your left is the half-edge's left face, and the other is the right face.

Each half-edge e the DCEL stores the following references (see Fig. 53):

- e.org: e's origin
- e.twin: e's oppositely directed twin half-edge
- e.left: the face on e's left side
- e.next: the next half-edge after e in counterclockwise order about e's left face
- e.prev: the previous half-edge to e in counterclockwise order about e's left face (that is, the next edge in clockwise order).

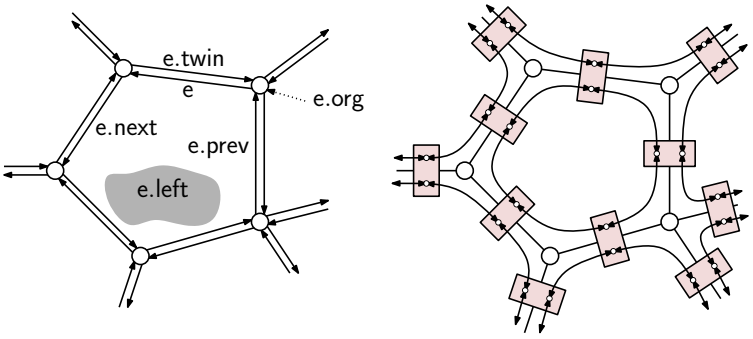


Fig. 53: Doubly-connected edge list.

You might observe that there are a number of potentially useful references that we did *not* store. This is done to save space, because they can all be computed easily from the above:

- e.dest: e's destination vertex (e.dest $\leftarrow$ e.twin.org)
- e.right: the face on e's right side (e.right $\leftarrow$ e.twin.left)
- e.onext: the next half-edge that shares e's origin that comes after e in counterclockwise order (e.onext $\leftarrow$ e.prev.twin)
- e.oprev: the previous half-edge that shares e's origin that comes before e in counterclockwise order (e.oprev $\leftarrow$ e.twin.next)

Fig. 53 shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternative way of viewing the data structure that gives a better sense of the connectivity structure is based on covering each edge with a two element block, one for e and the other for its twin. The next and prev references define a doubly-linked list around each of the faces of the mesh. The next references are directed counterclockwise around each face and the prev references are directed clockwise.

Mesh traversal with DCELs: Suppose that we have a mesh stored and a DCEL, and we want to enumerate the vertices that lie on some face f in counterclockwise order about the face. We could start at any incident edge e, output its origin e.org, and then go to the next edge in counterclockwise order about the face e.next. This is presented in the procedure faceVertices below.

Enumerate the vertices about a face

```
faceVerticesCCW(Face f) {
    Edge start = f.incident;
    Edge e = start;
    do {
        output e.org;
        e = e.next;
    } while (e != start);
}
```

As another example, suppose that we want to enumerate all the vertices that are neighbors of a given vertex v in clockwise order about this vertex. We could start at any incident edge e (which by definition has e as its origin), output its destination vertex, and then visit the next vertex about the origin in clockwise order.

Enumerate the neighbors of a vertex

```
vertexNeighborsCW(Vertex v) {
    Edge start = v.incident;
    Edge e = start;
    do {
        output e.dest; // formally: output e.twin.org
        e = e.oprev; // formally: e = e.twin.next
    } while (e != start);
}
```

Lecture 13: Motion Planning: Basic Concepts

Recap: Previously, we discussed navigation meshes as a technique for planning the motion of walking agents in games. In this and future lectures, we will delve deeper into the theory and practice of motion planning as it relates to game programming.

Configuration Spaces: To begin, let us consider the problem of planning the motion of a single agent among a collection of obstacles. Since the techniques that we will be discussing originated in the field of robotics, henceforth we will usually refer to a moving agent as a "*robot*". The environment in which the agent operates is called its *workspace*, which consists of a collection of geometric objects, called *obstacles*, which the robot must avoid. We will assume that the workspace is static, that is, the

obstacles do not move.[7] We also assume that a complete geometric description of the workspace is available to us.[8]

For our purposes, a *robot* will be modeled by two main elements. The first element is the robot's geometric model, say with respect to its reference pose (e.g., positioned at the origin). The second is its *configuration*, by which we mean a finite sequence of numeric parameters that fully specifies the position of the robot. Combined, these two elements fully define the robot's exact shape and position in space.

For example, suppose that the robot is a 2-dimensional polygon that can translate and rotate in the plane (shown as a triangle in Fig. 54(a)). Its geometric representation might be given as a sequence of vertices, relative to its reference position. Let us assume that this reference pose overlaps the origin. We refer to the location of the origin as the robot's *reference point*. The robot's configuration may be described by its translation, which we can take to be the (x, y) coordinates of its reference point after translation and an angle θ that represents the counterclockwise angle of rotation about its reference point (see Fig. 54(a)). Thus, the configuration is given by a triple (x, y, θ). We define the space of all valid configurations to be the robot's *configuration space*. For any point $p = (x, y, \theta)$ in this space, we define $\mathcal{R}(p)$ to be the corresponding *placement* of the robot in the workspace. The dimension of the configuration space is sometimes referred to as the robots number of *degrees of freedom* (DOF).

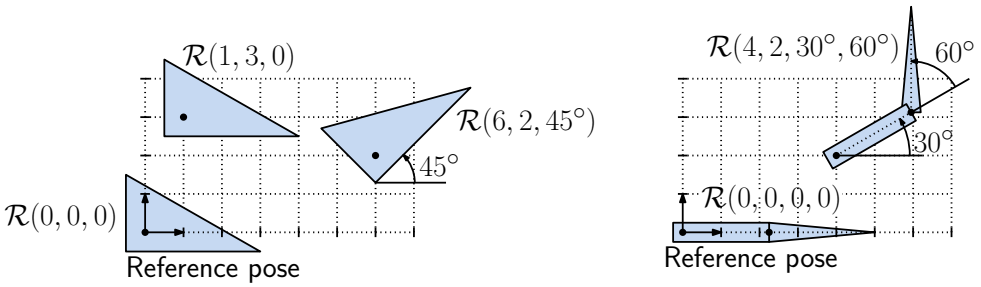


Fig. 54: Configurations of: (a) translating and rotating robot and (b) a translating and rotating robot with a revolute joints.

In 3-dimensional space, a similarly rigid object can be described by six parameters, the (x, y, z)-coordinates of the object's reference point, and the three Euler angles (θ, ϕ, ψ) that define its orientation in space.)[9]

A more complex example would be an *articulated arm* consisting of a set of links, connected to one another by a set of *revolute joints*. The configuration of such a robot would consist of a vector of joint angles (see Fig. 54(b)). The geometric description would probably consist of a geometric representation of the links. Given a sequence of joint angles, the exact shape of the robot could be derived by combining this configuration information with its geometric description.

Free Space: Because of limitations on the robot's physical structure and the obstacles, not every point in configuration space corresponds to a legal placement of the robot. Some configurations may be illegal

[7]The assumption of a static workspace is not really reasonable for most games, since agents move and structures may change. A common technique for dealing with dynamic environments is to separate the static objects from the dynamic ones, plan motion with respect to the static objects, and then adjust the plan incrementally to deal with the dynamic ones.

[8]The assumption of a known workspace is reasonable in computer games. Note that this is not the case in robotics, where the world surrounding the robot is either unknown or is known only approximately based on the robots limited sensor measurements.

[9]A quaternion might be a more reasonable representation of the robot's angular orientation in space. You might protest that the use of a quaternion will involve four parameters rather than three. But remember that the quaterions used for representing rotations are *unit quaternions*, meaning that once three of the parameters are given, the fourth one is fixed.

because:

- The joint angle is outside the joint's operating range. (E.g., you can bend your knee backwards, but not forwards . . . ouch!)
- The placement associated with this configuration intersects some obstacle (see Fig. 55(a)).

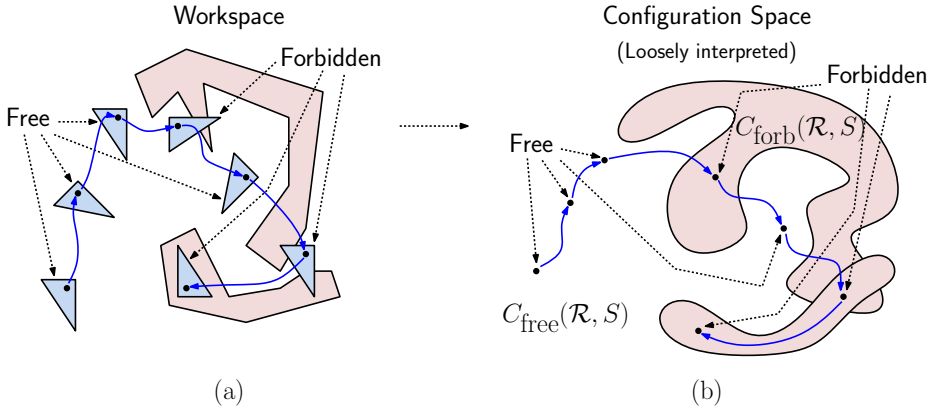


Fig. 55: Workspace showing free and forbidden configurations and a possible configuration space.

Such illegal configurations are called a *forbidden configurations*. Given a robot $\mathcal{R}$ and workspace S, the set of all forbidden configurations is denoted $C_{\text{forb}}(\mathcal{R}, S)$, and all other placements are called *free configurations*, and the set of these configurations is denoted $C_{\text{free}}(\mathcal{R}, S)$, or *free space*. These two sets partition configuration space into two distinct regions (see Fig. 55(b)).

C-Obstacles and Paths in Configuration Space: *Motion planning* is the following problem: Given a workspace S, a robot $\mathcal{R}$, and initial and final configurations $s, t \in C_{\text{free}}(\mathcal{R}, S)$, determine whether it is possible to move the robot from one configuration by a path $\mathcal{R}(s) \rightarrow \mathcal{R}(t)$consisting entirely of free configurations (see Fig. 56(a)).

Based on the definition of configuration space, it is easy to see that the motion planning problem reduces to the problem of determining whether there is a path from s to t in configuration space (as opposed to the robot's workspace) that lies entirely within the robot's free configuration subspace (see Fig. 56(b)). Thus, we have reduced the task of planning the motion of a robot in its workspace to the problem of finding a path for a single point through free configuration space.

Configuration Obstacles and Minkowski Sums: Since high-dimensional configuration spaces are difficult to visualize, let's consider the simple case of translating a convex polygonal robot in the plane amidst a collection of polygonal obstacles. In this cased both the workspace and configuration space are two-dimensional. We claim that, for each obstacle in the workspace, there is a corresponding *configuration obstacle* (or *C-obstacle*) that corresponds to it in the sense that if $\mathcal{R}(p)$ does not intersect the obstacle in the workspace, then p does not intersect the corresponding C-obstacle.

For simplicity, let us assume that the reference point for our robot $\mathcal{R}$ is at the origin. Let $\mathcal{R}(p)$ denote the *translate* of the robot so that its reference point lies at point p. Given a polygonal obstacle P, the corresponding C-obstacle is formally defined to the set of placements of $\mathcal{R}$ that intersect P, that is

$$\mathcal{C}(P) = \{p \ : \ \mathcal{R}(p) \cap P \neq \emptyset\}.$$

One way to visualize $\mathcal{C}(P)$ is to imagine "scraping" $\mathcal{R}$ along the boundary of P and seeing the region traced out by $\mathcal{R}$'s reference point (see Fig. 57(a)).

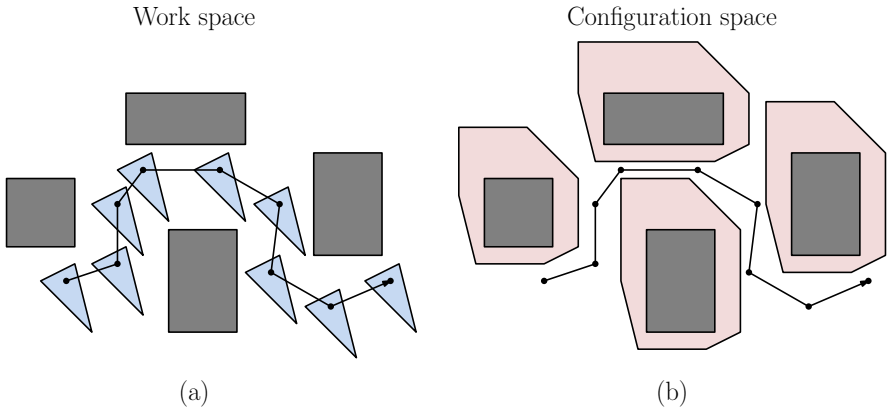


Fig. 56: Motion planning: (a) workspace with obstacles and (b) configuration space and C-obstacles.

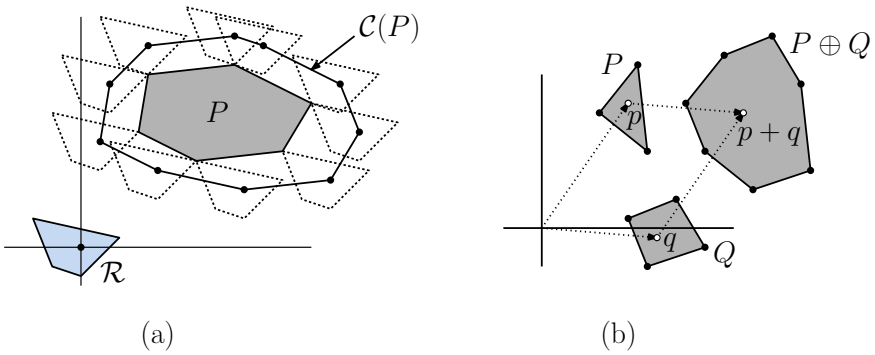


Fig. 57: Minkowski sum of two polygons.

Given $\mathcal{R}$ and P, how do we compute the configuration obstacle $\mathcal{C}(P)$? To do this, we first introduce the notion of a *Minkowski sum*. Let us think of points in the plane as vectors. Given any two sets P and Q in the plane, define their *Minkowski sum* to be the set of all pairwise sums of points taken from each set (see Fig. 57(b)), that is,

$$P \oplus Q \;=\; \{p + q \;:\; p \in P,\; q \in Q\}.$$

Also, define $-S = \{-p \;:\; p \in S\}$. (In in the plane $-S$ is just the 360° rotation of S about the origin, but this does not hold in higher dimensions.) We introduce the shorthand notation $\mathcal{R} \oplus p$ to denote $\mathcal{R} \oplus \{p\}$. Observe that the *translate* of $\mathcal{R}$ by vector p is $\mathcal{R}(p) = \mathcal{R} \oplus p$. The relevance of Minkowski sums to C-obstacles is given in the following claim.

Claim: Given a translating robot $\mathcal{R}$ and an obstacle P, $\mathcal{C}(P) = P \oplus (-\mathcal{R})$ (see Fig. 58).

Proof: Observe that $q \in \mathcal{C}(P)$ iff $\mathcal{R}(q)$ intersects P, which is true iff there exist $r \in \mathcal{R}$ and $p \in P$ such that $p = r + q$ (see Fig. 58(a)), which is true iff there exist $-r \in -\mathcal{R}$ and $p \in P$ such that $q = p + (-r)$ (see Fig. 58(b)), which is equivalent to saying that $q \in P \oplus (-\mathcal{R})$. Therefore, $q \in \mathcal{C}(P)$ iff $q \in P \oplus (-\mathcal{R})$, which means that $\mathcal{C}(P) = P \oplus (-\mathcal{R})$, as desired.

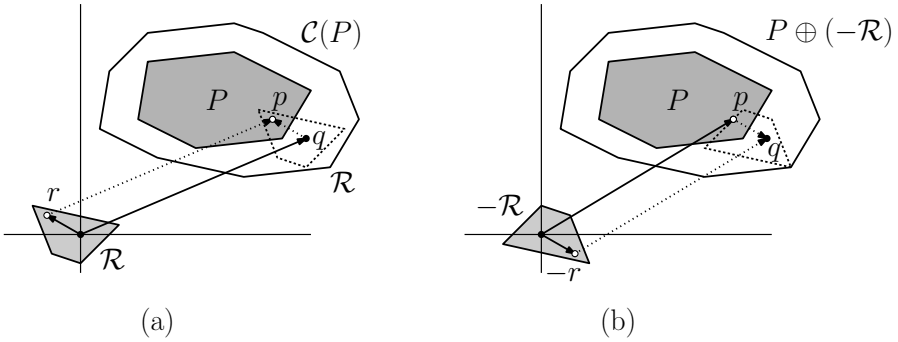


Fig. 58: Configuration obstacles and Minkowski sums.

It is an easy matter to compute $-\mathcal{R}$ in linear time (by simply negating all of its vertices) the problem of computing the C-obstacle $\mathcal{C}(P)$ reduces to the problem of computing a Minkowski sum of two convex polygons. We'll show next that this can be done in $O(m + n)$ time, where m is the number of vertices in $\mathcal{R}$ and n is the number of vertices in P.

Note that the above proof made no use of the convexity of $\mathcal{R}$ or P. It works for any shapes and in any dimension. However, computation of the Minkowski sums is most efficient for convex polygons. We will not present the algorithm formally here, but here is an intuitive explanation. First, compute the vectors associated with the edges of each polygon and merge them into a single list, sorted by angular order. Then link them together end-to-end (see Fig. 59). (It is not immediately obvious that this works, but it can be proved to be correct.)

C-Obstacles for Rotating Robots: When rotation is involved, this scraping process must consider not only translation, but all rotations that cause the robot's boundary to touch the obstacle's boundary. (One way to visualize this is to fix the value of θ, rotate the robot by this angle, and then compute the translational C-obstacle with the robot rotated at this angle. Then, stack the resulting C-obstacles on top of one another, as θ varies through one complete revolution. The resulting "twisted column" is the C-obstacle in 3-dimensional space.) Note that because the configuration space encodes not only translation, but the joint angles as well. Thus, a path in configuration space generally characterizes

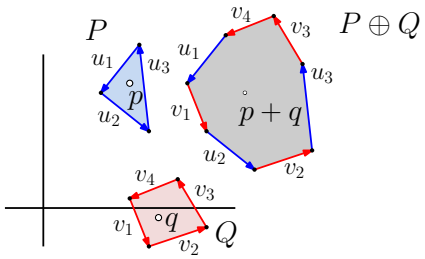


Fig. 59: Computing the Minkowski sum of two convex polygons.

both the translation and the individual joint rotations. (This is insanely hard to illustrate, so I hope you can visualize this on your own!)

When dealing with polyhedral robots and polyhedral obstacles models under translation, the C-obstacles are all polyhedra as well. However, when revolute joints are involved, the boundaries of the C-obstacles are curved surfaces, which require more effort to process than simply polyhedral models. Complex configuration spaces are not typically used in games, due to the complexity of processing them. Game designers often resort to more ad hoc tricks to avoid this complexity, and the expense of accuracy.

Lecture 14: Motion Panning: Finding Paths

Finding Paths: Last time we discussed how the problem of planning the motion of a robot amidst a set of obstacles could be reduced to the task of computing a path between two points in the *configuration space* associated with the robot. Unfortunately, computing such a path for even a single point in space is a nontrivial problem. Typically, we are interested not merely in the existence of a path, but rather finding a "good" path, where goodness connotes different things in different contexts (e.g., shortness, low-energy, natural looking, etc.) In this lecture, we will present a number of different approaches for computing such paths.

Potential-Field Navigation: The analogy to understand this approach is to imagine a smoothly varying terrain with hills and valleys. Suppose you place a marble on top of one of the hills of the terrain and let it go. It will naturally slide down until reaching the lowest point.

How can we base a path finding algorithm on this idea? Suppose that your obstacles reside in 2-dimensional space (see Fig. 60(a)). The idea is to model this as a terrain in 3-dimensional space. The start point s will lie on top of a hill, the goal point t as lying in the bottom of a deep basin, and all the obstacles are modeled as vertical cylinders whose 2-dimensional projections are the obstacles, but which have steep vertical sides (see Fig. 60(b)).

Now, when you start the marble rolling at point s, the force of gravity will naturally draw it down towards the destination t. Because the obstacles form high vertical "plateaus", the marble naturally roll around and avoid them. With a bit of luck, the marble will eventually roll around on this terrain and find its way from s to t. By projecting the resulting path down into the plane, we obtain the desired path (see Fig. 60(c)).

More formally, the process is modeled as follows. First, we set up an attractive field to the target point so that by minimizing this potential we approach the target. For example, this might just be the squared distance from the current point p to the target, $\text{dist}(p,t)^2$. Second, we set up a repulsive field around obstacles (and generally any forbidden regions of the configuration space). For example, let O denote the set of obstacles and let p be the current point. For any $o \in O$ define the potential field at p to be the inverse of the squared distance from o to p, $\text{dist}(p,o)^2$. Then, we define the overall potential

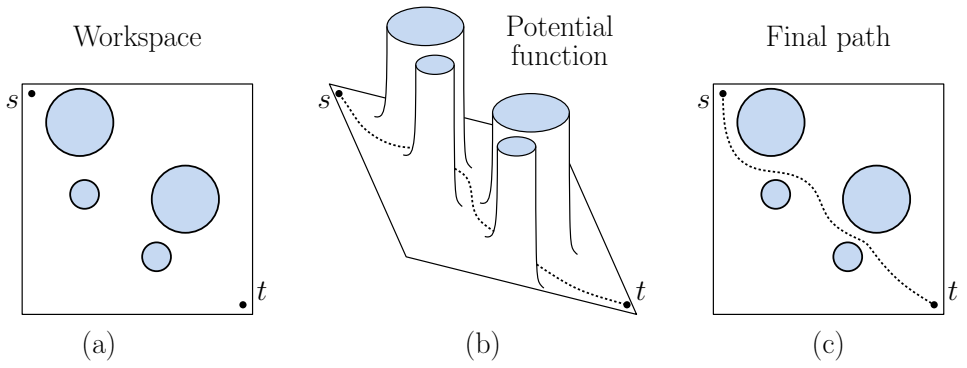


Fig. 60: Computing paths by potential-field navigation.

to be the sum of all these potentials. Finally, we could include weight factors w_t and w_o to control the relative strength of attraction versus repulsion (see Fig. 60(b))

$$\Psi(p) \;=\; w_t \cdot \text{dist}(p,t)^2 + w_o \cdot \sum_{o \in O} \frac{1}{\text{dist}(p,o)^2}.$$

The high potential walls around the obstacles keeps the ball from rolling into them. To induce the ball to roll from s to t, we put s at the peak of a very tall, broad mountain (think Mt. Fuji) and we put t and the bottom of a very deep, broad bowl. The final potential field Ψ is the sum of these various functions.

Path Finding via Gradient Descent: Given our potential field, we can apply a physics simulator to let our robotic marble flow "downhill" from s to t (and hope it eventually arrives!)

How do we compute this path? A natural approach is to compute a path of *steepest descent*. Given a point $p = (x, y)$, let $\Psi(x, y)$ denote the value of the potential field at any point direction (x, y), then the direction of steepest ascent it given by the *gradient vector*, which can be computed from the partial derivatives of Ψ. More formally, the gradient is

$$\nabla \Psi \;=\; \left(\frac{\partial \Psi}{\partial x}, \frac{\partial \Psi}{\partial y} \right)$$

(see Fig. 61). You might wonder why the partial derivative is used here. Observe, for example that if the function grows very rapidly with x, but is almost flat with respect to y, then the gradient will have a very high x-component and the y-component will be very close to zero. It takes a bit of calculus to show that among all directions, the gradient provides the direction of most rapid change. By the way, one reason for using squared distances in the above potential function, rather than standard Euclidean distance. It is much easier to compute derivatives of polynomial functions than functions involving square roots.

Given any starting point p, we can compute the next point along the direction of steepest descent as

$$p' \;\leftarrow\; p - \delta \cdot \nabla \Psi(p),$$

for a suitably small *step size* δ. By repeatedly recomputing the gradient and taking another step, we will eventually walk to a *local minimum* of the potential function.

There is some art in how the step size is determined. If the step size is too big, we may shoot past the minimum, and if the step size is too small, we may require many iterations before converging.

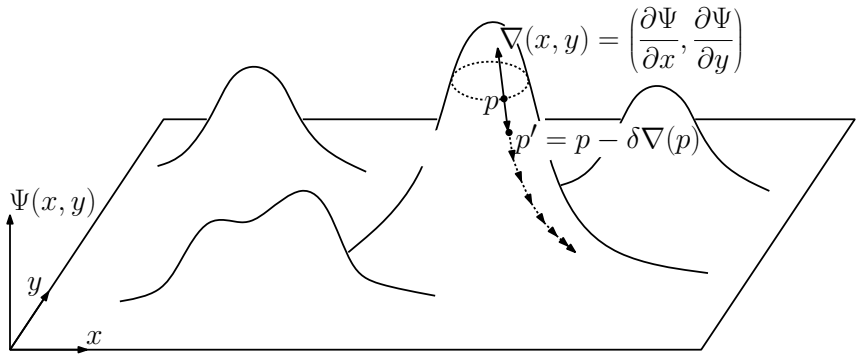


Fig. 61: Finding the path via steepest descent.

Advantages: The potential-field method is very easy to implement. Because the movement point naturally follows a smooth energy-minimizing path, when it converges, it tends to result in smooth, natural-looking motion. However, it is best used for simple motions, where the desired path doesn't involve many twists and turns.

Disadvantages: In addition to the difficulty of determining a good step size, the most significant disadvantage of the potential-based method for path planning is that it can get trapped in *local minima*. If t is not at the bottom of the local minimum then the algorithm simply gets stuck.

Discretizing Configuration Space: Because continuous spaces are difficult to search, it is common to find paths by a two-step process:

Discretize: Reduce the problem to one of searching a discrete structure (either a graph or a subdivision of space).

Search: Apply a path-finding algorithm (such as Breadth-First Search, Dijkstra's algorithm, or A*) to compute the path in the discrete structures. (We will discuss these algorithms in future lectures.)

In the remainder of this lecture, we will discuss a number of approaches for computing the aforementioned discrete structure.

Waypoints and Road Maps: Perhaps the simplest approach for generating a navigation graph, is to scatter a large number of points throughout free space, sometimes called *waypoints*, and then connect nearby waypoints to one another if the segment between them does not intersect any obstacles. (Since this is generally happening in configuration space, the points are in configuration free space and the segments should not intersect any C-obstacles.)

The edges of this graph can be labeled with the distance[10] between the associated points. The resulting graph is called a *road map*. Given the location of the start point s and the destination point t, we join these with edges to nearby waypoints. Finally, we can invoke a shortest path algorithm to compute the final path. If the graph is connected, then this is guaranteed to yield a valid path in configuration space, which is then translated back to a motion plan in the robot's workspace.

Selecting Waypoints: There are a number of methods for computing waypoints. Here are a few:

Placed by the game designer: The game designer has a notion of where it is natural for the game agents to move, and so places waypoints along routes that he/she deems to be important. For example, this would include points near the entrances and exits of rooms in an indoor environment

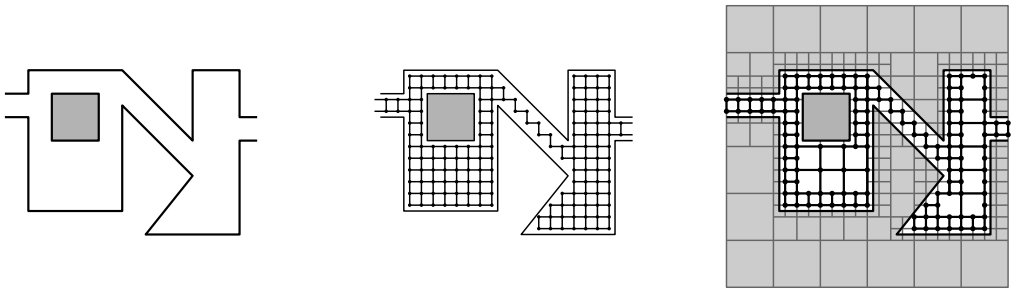
[10]In the case of translational motion, distance is an easily defined notion. When the configuration space include rotations, we need to define distances in terms of both rotations and translations.

or along the streets or crosswalks in an urban setting. This gives the designer a lot of control of the motion of the game agents, and a lot of flexibility to add more points where motion is highly constrained and fewer where motion is unconstrained.

In a large environment, however, there may be too many waypoints for a single designer to place. Thus, we would like something more automated.

Grid: The simplest way to cover a large area is to overlay a square grid of sufficiently high density and generate waypoints along the vertices of this grid that lie in free space (see Fig. 62(b)). Each grid point has up to four possible neighbors (assuming 2-dimensional space—in d-dimensional space each has up to $2d$ neighbors).

This has the advantage of simplicity, but there are some notable drawbacks. First, it can result in the generation of a very large number of grid points. The grid resolution has to be set small enough that the narrowest corridor has sufficiently many points, but then wide open areas will have many more points than are needed (see Fig. 62(b)). A second drawback is the path segments generated are all parallel to the coordinate axes. Hence, such a path will zig-zag excessively if the path travels diagonally. This issued can be ameliorated by a postprocessing pass that smooths out the path, but unless care is taken, the smoothing process might introduce shortcuts that pass through obstacles, which is not acceptable.



(a) (b) (c)

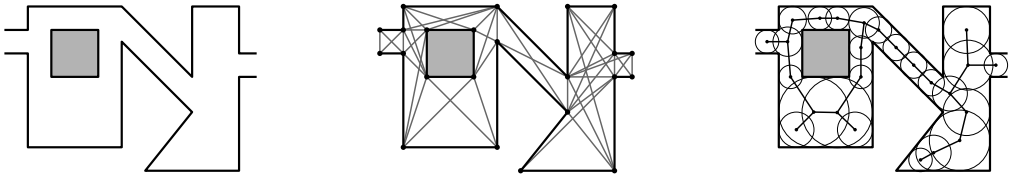
Fig. 62: A road map for a set of obstacles (a) based on waypoints generated on: (b) a grid and (c) a quadtree.

Adaptive Grid (Quadtree): One way to deal with the grid's lack of adaptivity is to apply a hierarchical point placement system that adapts the density of point placement to the distance to the closest obstacle. A natural generalization of the grid approach is to place waypoints on the vertices of a quadtree decomposition. Recall that a quadtree decomposes space into square *cells* (or generally hypercubes in higher dimensional space). A quadtree cell is said to be *stabbed* if the boundary of some obstacle cuts through it. Assuming that we can detect when a quadtree cell is stabbed, we repeatedly refine any stabbed cells until the cells are deemed to be sufficiently small. The waypoints are then placed at the vertices of the quadtree that lie in free space (see Fig. 62(c)).

While this adds adaptivity, it still does not resolve the issue of path segments that are parallel to the coordinate axes.

Boundary Vertices and the Visibility Graph: In order to deal with the problem of path segments that are aligned with the coordinate axes, we would like a method of generating waypoints that is independent of the coordinate system, and relies solely on the geometry of free space. Let us assume for now that we are working in 2-dimensional space, and the configuration space is bounded by line segments. If one is interested in shortest paths (assuming the Euclidean distance) it is easy to prove that such a path will only make turns at the vertices of the obstacle vertices. We say that two vertices of the boundary of free space are *visible* if the line segment between them

does not intersect any obstacles. The *visibility graph* is a graph whose vertices are the vertices of the boundary of free space and whose edges are the visible pairs (see Fig. 62(b)). It follows from the above remarks that the shortest path between any two points is a path in the visibility graph.



(a) (b) (c)

Fig. 63: A road map for a set of obstacles (a) based on waypoints generated on: (b) the visibility graph of the obstacle vertices, and (c) the medial axis.

The visibility graph is an intrinsic structure, meaning that it depends only on the object's geometry, not on the placement of the coordinate system. While it has a number of nice theoretical properties, it also has a number of drawbacks that make it unsuitable as a general purpose solution. First, the number of edges in the visibility graph can be as high as $O(n^2)$, where n is the number of vertices. If n is very large, this quadratic size may be unacceptable. This problem can be ameliorated by pruning the graph, say by keeping only the shorter of two edges that share a common vertex and have a very similar slope.

A second problem of the visibility graph is that the paths it generates, while of minimal length, have the undesirable property that they point scrapes right only the boundary. This doesn't generate very natural looking motion. This issue also can be ameliorated by first constructing an artificial boundary that is slightly offset from the actual boundary, and then constructing the visibility graph of the offset boundary.

A third problem with the visibility graph is that it guarantees shortest paths only in 2-dimensional space. In 3-dimensional space, it is not longer the case that the shortest path between points bends at vertices. It may bend in the interior of an boundary edge. The locations of these interior bending points cannot be predicted in advance (since they generally depend on the locations of the starting and ending points).

Medial-Axis Waypoints: The shortcomings of the visibility graph suggest a very different approach to path finding. People who walk down a corridor do not usually "scrape" along the boundary. Rather they usually seek a path near the center of the corridor, that is, they seek a path of *maximum clearance* from the obstacles. How can we compute such a path?

We say that a circular disk D lying entirely in free space is *maximal* if there is no obstacle-free disk of larger radius that contains D. The union of the centers of all maximal disks naturally defines a set of points that runs along the center of the free-space domain. It is a fundamental object in geometry, called the *medial axis* (see Fig. 63(c)).

By sampling waypoints on or near the medial axis, the robot will naturally move along the centers of corridors. (Of course, you can add to this a bit of random variation.) This method of placing waypoints is best for 2-dimensional domains (since it is messier to compute the medial axis of higher dimensional configuration spaces).

Adaptive Randomized Placement and PRMs: Another adaptive approach to placing waypoints that avoids dependencies on the coordinate axes is to select the waypoint placement randomly. Here is the idea. On the ith iteration, we generate a random point p within the domain of interest. We test to see whether p lies within free space. If not we discard it and go to the next iteration.

Otherwise, we next attempt to connect p to other nearby previously sampled points. This can be done in various ways. The simplest way to do this is to compute the k nearest neighbors of p. (Using distance alone as a criterion might produce neighbors that lie exclusively to one side of p, which is not good. More sophisticate methods can be used to ensure, if possible, that p's neighbors cover all the directions about p.) Next, we consider a line segment joining p to each of these neighbors. If the line segment lies entirely within free space, we add this segment to our road map. We repeat this process until some stopping condition is met, for example, until some fixed number of successful samples are generated or until the entire structure consists of one connected component. Because of the randomized nature in which points are generated, the resulting structure is called a *probabilistic road map* (or PRM).

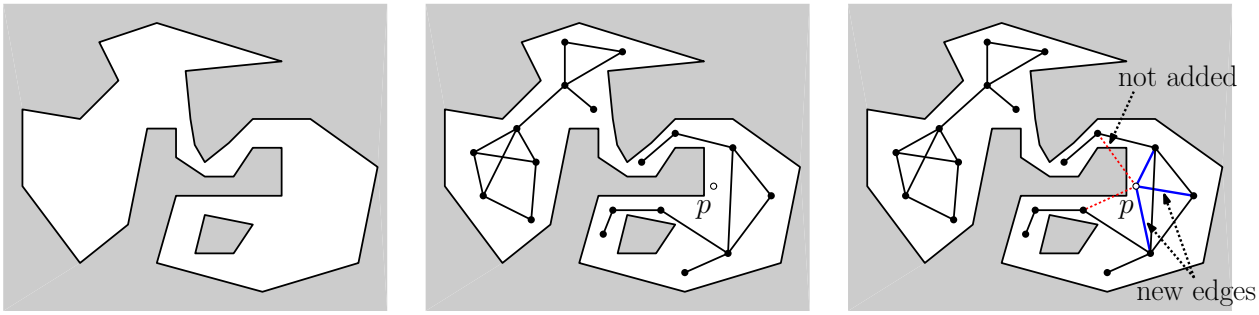


Fig. 64: Generating a probabilistic road map (PRM) for a set of obstacles. (b) The roadmap after iteration $i-1$ and (c) the result of adding the ith point. Edges that intersect obstacles (red) are not added.

PRMs are very popular in the field of robotics. They can be applied in arbitrary dimensions. It can be proved that, if there is a path between two configurations then with high probability the PRM will eventually discover it. Of course if the path travels through a narrow passageway (as seen in the middle part of the obstacle set of Fig. 64) it may take a lot of samples to discover this connection.

Because points are randomly generated throughout the domain, it suffers from some of the same issues as uniform grids. Wide open areas of free space will receive an excessive number of waypoints while narrow corridors may receive too few. Unlike uniform grids, it is possible to detect that a newly added waypoint is redundant and so it can be ignored. PRMs suffer the same problem as all the other waypoint methods we have discussed so far. The paths do not generally travel along natural paths, but rather they zig-zag from one waypoint to the next.

Rapidly-expanded Random Trees (RRTs): One of the issues with PRMs is that the structure that is generated is not necessarily connected. Another popular adaptive approach to generating a roadmap through randomization is through that process that guarantees that the structure that is generated is connected, and in fact it is a spanning tree over the set of sample points. Spanning trees are nice for navigation because (due to the fact that they are connected and acyclic) there is a unique path between any two points. While this may not be great for computing shortest paths, it is useful for determining the existence of any valid motion.

As with PRMs, the process begins by random sampling points from the domain. In this case, we will keep every sampled point, even if it does not lie within free space. Let us assume that we have already computed a spanning tree for the existing set of sample points (consider just the line segment p_0p_1 in Fig. 65(a)), and we are considering the addition of a new sample point p. We compute the closest point q on the current spanning tree to p. Note that q does not need to be a sampled point. It is allowed to lie within the interior of an edge of the spanning tree. If so, we add the point q as a new vertex to the spanning tree. We then consider the line segment

from p to q. If this line segment lies entirely within free space, we add it to the tree (see points p_2 and q_2 in Fig. 65(a) and p_3 and q_3 in Fig. 65(b)). If not (see q_4 to p_4 in Fig. 65(c)), we trim the segment back to obtain the longest subsegment that lies within free space with one endpoint at q. Let qp' denote this segment (see $q_4p'_4$ in Fig. 65(d)). We add this segment to the tree. The result is called a *rapidly-expanding random tree* or (RRT).

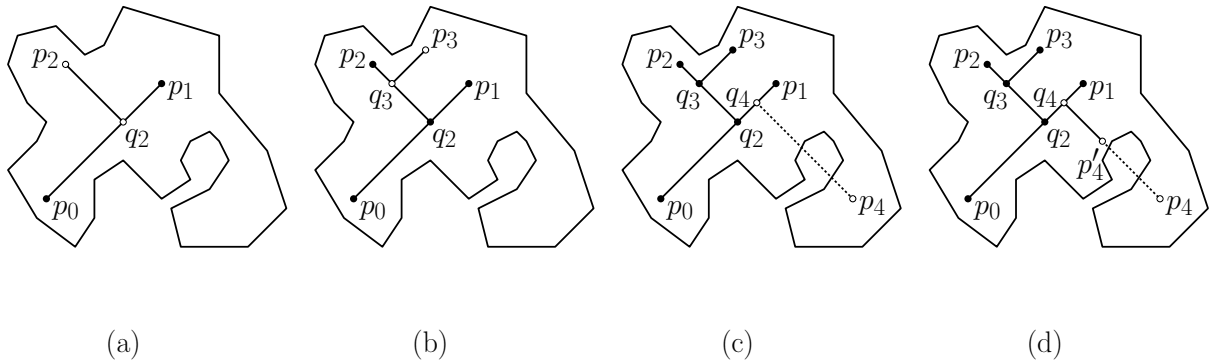


Fig. 65: Generating a roadmap through the use of rapidly-expanding random trees (RRTs).

Next to PRMs, RRTs are perhaps the most widely used method for computing connectivity structures in configuration spaces. Notice that both PRMs and RRTs have the advantage that they can be applied in configuration spaces of arbitrary dimensions.

Lecture 15: Motion Planning: Multiple Agent Motion

Recap: In the previous lectures, we have discussed techniques for computing the motion of a single agent. Today we will discuss techniques for planning and coordinating the motion of multiple agents. In particular, we will discuss three aspects of this problem:

Particle systems: Modeling (unintelligent) motion for a collection of objects

Flocking behavior: Where all agents are moving as a unit

Crowd behavior: Where a number of agents, each with their own desired destination, are to move without colliding with each other.

Today, we will discuss particle systems and flocking behavior.

Particle Systems: A *particle system* is a technique that uses a large number of small graphical artifacts, called *particles*, to create a large-scale, typically "fuzzy," visual effect. Examples of applications of particle systems include many amorphous natural phenomena such as fire, smoke, water, clouds, and explosions. In these examples particles are born and die dynamically, but there are also variants of particle systems where particles are persistent. These are used to produce static phenomena such as galaxies or "stringy" phenomena such as hair, grass, and other plants.

A fundamental principle that underlies particle systems is that our brains tend to cluster an aggregation of similar objects into the impression of a single impression. Thus, the many individual water drops that cascade down a flowing fountain are perceived as a flowing stream of water (see Fig. 66). (Note that particle systems do not really belong in this lecture on artificial intelligence, since their behavior is based solely on physical laws, and not on any model of intelligent behavior. Nonetheless, we have decided to discuss them here because they do represent a common technique for generating interesting patterns of movement in games and other applications of interactive computer graphics.)

Fig. 66: A particle system simulating the flow of a liquid.

Particle systems are almost as old as computer games themselves. The very earliest 2-dimensional games, such as *Spacewar!* and *Asteroids* simulated explosions through the use of a particle system. The power of the technique, along the term "particle system," came about in one of the special effects in the second Star Trek movie, where a particle system was used to simulate a fire spreading across the surface of a planet (the *genesis effect*).

How Particle Systems Work: One of the appeals of particle systems is that they are extremely simple to implement, and they are very flexible. Particles are created, live and move according to simple rules, and then die. The process that generates new particles is called an *emitter*. For each frame that is rendered, the following sequence of steps is performed:

Emission: New particles are generated and added to the system. There may be multiple sources of emission (and particles themselves can serve as emitters).

Attributes: Each new particle is assigned its (initial) individual properties that affect how the particle moves over time and is rendered. These may change over time and include the following:

Geometric attributes: initial position and velocity

Graphics attributes: shape, size, color, transparency

Dynamic attributes: lifetime, influence due to forces such as gravity and friction

Death: Any particles that have exceeded their prescribed lifetime are removed from the system.

Movement: Particles are moved and transformed according to their dynamic attributes, such as gravity, wind, and the density of nearby particles.

Rendering: An image of the surviving particles is rendered. Particles are typically rendered as a small blob.

In order to create a natural look, the process of emitting particles and the assignment of their attributes is handled in the probabilistic manner, where properties such as the location and velocity of particles being determined by a random number generator. Many game engines (including Unity) provide flexible systems for generating particle systems, offering a multitude of options that can be set by the designer.

Particle system can be programmed to execute any set of instructions at each step, but usually the dynamic properties of particles are very simple, and do not react in any complex manner to the presence of other particles in the system. Because the approach is procedural, it can incorporate any computational model that describes the appearance or dynamics of the object. For example, the motions and transformations of particles could be tied to the solution of a system of partial differential equations. In this manner, particles can be used to simulate physical fluid phenomena such as water, smoke, and clouds.

Updating Particles in Time: There are two common ways to update particles over time.

Closed-form function: Every particle is represented by a *parametric function* of time (and coefficients that are given by the particle's initial motion attributes. For example, given the particle's initial position p_0, its initial velocity vector $\vec{v}_0$, and some fixed field force $\vec{g}$ (representing, for example, the affect of gravity as a vector that points downwards) the position of the particle at time t can be expressed in closed form as:

$$p(t) \;=\; p_0 + \vec{v}_0 t + \frac{1}{2}\vec{g}t^2.$$

(If you have ever taken a course in physics, this formula will be familiar to you. If not, don't worry how it was derived. It is a simple consequence of Newton's basic laws of motion.) On the positive side, this approach requires no storage of the evolving state of the particle, just its initial state and the elapsed time. On the negative side, this approach is very limited, and does not allow particles to respond to each other or their environment.

Discrete physical integration: In this type of system, each particle stores in current *physical state*. This state consists of the particle's *current position*, which is given by a point p, its *current velocity*, which is given by a vector $\vec{v}$, and its *current acceleration*, which is given by a vector $\vec{a}$. Acceleration can be thought of as the accumulated effect of all the forces acting on the particle.[11] (For example, the force of gravity decreases the vertical component of the velocity. On the other hand, the force of air resistance or friction tends to decrease the particle's absolute velocity, without altering its direction.) We can then update the state over a small time interval, Δt. Think of Δt as the elapsed time between consecutive frames, or a fixed update time, such as 0.1 seconds. By the basic laws of kinematics, (1) the total vector sum $\vec{F}$ of forces alters acceleration, (2) acceleration changes the object's velocity over Δt, and (3) velocity changes the object's position in space over Δt:

$$(1)\ \vec{a} \;\leftarrow\; \frac{\vec{F}}{m}, \qquad (2)\ \vec{v}' \;\leftarrow\; \vec{v} + \vec{a}\cdot\Delta t \qquad \text{and} \qquad (3)\ p' \;\leftarrow\; p + \vec{v}\cdot\Delta t,$$

where p' and $\vec{v}'$ denote the particle's new position and velocity, respectively. (Note that, except for time, these are 3-dimensional vector quantities.)

Doing this in Unity: The Unity physics engine will take care of these operations automatically, whenever you attach to it a Rigidbody component (and assuming that isKinematic is not enabled). When isKinematic is disabled (the object is under control of the physics engine) then the body can be moved by applying forces.

```
Rigidbody rb = GetComponent<Rigidbody>();
rb.AddForce(Vector3.up * 10f);    // apply an upward force
```

The function AddForce has a second optional argument that controls the manner in which the force is applied. These include

- Force: Add a continuous force to the rigidbody, using its mass
- Acceleration: Add a continuous acceleration to the rigidbody, ignoring its mass
- Impulse: Add an instant force impulse to the rigidbody, using its mass
- VelocityChange: Add an instant velocity change to the rigidbody, ignoring its mass

[11] When dealing with particles, it is common to ignore the object's mass. In general, the mass m of an object is its resistance to change its velocity as a consequence of a force. The acceleration $\vec{a}$ due to a force $\vec{F}$ is given by $\vec{a} = \vec{F}/m$, which is derived from the well-known formula $\vec{F} = m\vec{a}$. Note that acceleration is a vector quantity, where the direction is given by the direction of the force that is acting on the body.

There is also a function AddTorque that adds a rotational force. The argument is a vector quantity, where the direction gives the axis of rotation and the magnitude gives the strength of the force (according to the right-hand rule, I believe).

When isKinematic is enabled (not under the control of the physics engine) the object can be moved either by changing transform.position or rigibody.position directly. The Unity manual says that the latter is more efficient. The same applies to transform.rotation and rigidbody.rotation. Altering the position and rotation will cause the object to "teleport" instantly to the new position. Unity manual also explains that there is a smoother way to apply motions to kinematic rigid bodies, through the functions movePosition and moveRotation

```
void Update () {
  Vector3 velocity = ... // current linear velocity
  Vector3 angVeloc = ... // current (Euler) angular velocity
  float DeltaT = Time.deltaTime; // elapsed time
  rb.MovePosition(transform.position + velocity * DeltaT);
  Quaternion deltaRot = Quaternion.Euler(angVeloc * DeltaT);
  rb.MoveRotation(rb.rotation * deltaRot);
}
```

Flocking Behavior: Next, let us consider the motion of a slightly smarter variety, namely *flocking*. We refer to flocking behavior in the generic sense as any motion arising when a group of agents adopt a decentralized motion strategy designed to hold the group together. Such behavior is exemplified by the motion of groups animals, such as birds, fish, insects, and other types of herding animals (see Fig. 67).

Fig. 67: An example of complex emergent behavior in flocking.

In contrast to full crowd simulation, where each agent may have its own agenda, in flocking it is assumed that the agents are *homogeneous*, that is, they are all applying essentially the same motion update algorithm. The only thing that distinguishes one agent from the next is their position relative to the other agents in the system. It is quite remarkable that the complex formations formed by flocking birds or schooling behavior in fish can arise in a system in which each creature is following (presumably) a very simple algorithm. The apparently spontaneous generation of complex behavior from the simple actions of a large collection of dynamic entities is called *emergent behavior*. While the techniques that implement flocking behavior do not involve very sophisticated models of intelligence, variants of this method can be applied to simple forms of crowd motion in games, such as a crowd of people milling around in a large area or pedestrians strolling up and down a sidewalk.

Boids: One of the earliest models and perhaps the best-known model for flocking behavior was given by C.

W. Reynolds from 1986 with his work on "boids." (The term is an intentional misspelling of "bird.") In this system, each agent (or *boid*) determines its motion based on a combination of four simple rules:

Separation: Each boid wishes to avoid collisions with other nearby boids. To achieve this, each boid generates a *repulsive potential field* whose radius of influence extends to its immediate neighborhood. Whenever another boid gets too close, the force from this field will tend to push them apart.

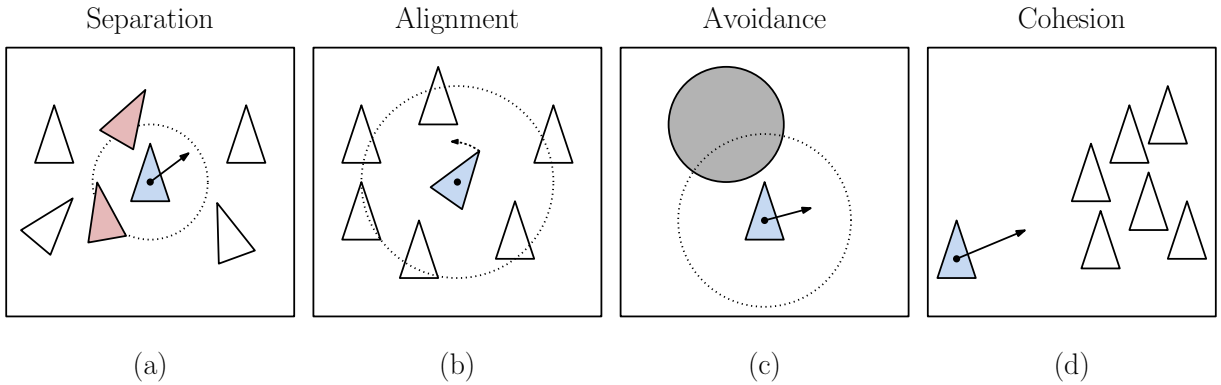


Fig. 68: Forces that control flocking behavior.

Alignment: Each boid's direction of flight is aligned with nearby boids. Thus, local clusters of boids will tend to point in the same direction and hence will tend to fly in the same direction.

Avoidance: Each boid will avoid colliding with fixed obstacles in the scene. At a simplest level, we might imagine that each fixed obstacle generates a repulsive potential field. As a boid approaches the object, this repulsive field will tend to cause the boid to deflect its flight path, thus avoiding a collision. Avoidance can also be applied to predators, which may attack the flock. (It has been theorized that the darting behavior of fish in a school away from a shark has evolved through natural selection, since the sudden chaotic motion of many fish can confuse the predator.)

Cohesion: Effects such as avoidance can cause the flock to break up into smaller subflocks. To simulate the flocks tendency to regroup, there is a force that tends to draw each boid towards the center of mass of the flock. (In accurate simulations of flocking motion, a boid cannot know exactly where the center of mass is. In general the center of attraction will be some point that the boid perceives being the center of the flock.)

Boid Implementation: Next let us consider how to implement such a system. We apply the same discrete integration approach as we did used for particle systems. In particular, we assume that each boid is associated with a state vector $(p, \vec{v})$ consisting of its current position p and current velocity v. (We assume that the boid is facing in the same direction as it is flying, but, if not, a vector describing the boid's angular orientation can also be added to the state.) We think of the above rules as imposing forces, which together act to define the boid's current acceleration. Given this acceleration vector a caused by the boid forces, we apply the update rules described earlier for particle systems:

$$\vec{v}' \leftarrow \vec{v} + \vec{a} \cdot \Delta t \qquad \text{and} \qquad p' \leftarrow p + \vec{v} \cdot \Delta t,$$

in order to obtain the new position p' and new velocity vector $\vec{v}'$.

How are these forces computed? First observe that, for the sake of efficiency, you do not want the behavior of each boid to depend on the individual behaviors of the $n - 1$ boids, since this would be much to costly, taking $O(n^2)$ time for each iteration. Instead, the system maintains the boids stored

in a *spatial data structure*, such as a grid or quadtree, which allows each boid to efficiently determine the boids that are near to it. Rules such as separation, avoidance, alignment can be based on a small number of nearest neighbor boids. Cohesion requires knowledge of the center of the flock, but this can be computed once at the start of each cycle in $O(n)$ time.

Since these are not really natural forces, but rather heuristics that are used to guiding motion, it is not essential to apply kinematics rigorously in order to determine future motion. Rather, each rule naturally induces a directional influence. (For example, the force of avoidance is directed away from the anticipated point of impact while the force for alignment is in the average direction that nearby boids are facing.) Also, each rule can be associated with a strength whose magnitude depends, either directly or inversely, on the distance from the point of interest. (For example, avoidance forces are very strong near the obstacle but drop off rapidly. In contrast, the cohesive force tends to increase slowly as the distance from the center of flock increases.) Thus, given a unit vector $\vec{u}$ pointing in the induced direction and the strength s given as a scalar, we can compute the updated acceleration vector as $\vec{a} = c \cdot s \cdot \vec{u}$, where c is a "fudge factor" that can be adjusted by the user that is used to model other unspecified physical quantities such as the boid's mass.

One issue that arises with this or any dynamical system is how to avoid undesirable (meaning unnatural looking) motion, such as collisions, oscillations, and unrealistically high accelerations. Here are two approaches:

Prioritize and truncate: Assume that there is a fixed maximum magnitude for the acceleration vector (based on how fast a boid can change its velocity based on what sort of animal is being modeled). Sort the rules in priority order. (For example, predator/obstacle avoidance is typically very high, flock cohesion is low.) The initial acceleration vector is the zero vector (meaning that the boid will simply maintain its current velocity). As each rule is evaluated, compute the associated acceleration vector and add it to the current acceleration vector. If the length of the acceleration vector ever exceeds the maximum allowed acceleration, then stop and return the current vector.

Weight and clamp: Assign weights to the various rule-induced accelerations. (Again, avoidance is usually high and cohesion is usually low.) Take the weighted sum of these accelerations. If the length of the resulting acceleration vector exceeds the maximum allowed acceleration, then scale it down

The first method has the virtue that, subject to the constraint on the maximum acceleration, it processes the most urgent rules first. The second has the virtue that every rule has some influence on the final outcome. Of course, since this is just a heuristic approach, the developer typically decides what sort of approach yields the most realistic results.

Lecture 16: Motion Planning: Computing Shortest Paths

Recap: In the previous lecture, we demonstrated how, through the use of waypoints and roadmaps, geometric path finding problems can be reduced to path finding in graphs. Today, we will discuss a number of efficient methods for computing shortest paths in graphs and applications to other types of path-finding problems.

Computing Shortest Paths: The problem of computing shortest paths in graphs is very well studied. Recall that a *directed graph* (or *digraph*) $G = (V, E)$ is a finite set of *nodes* (or *vertices*) V and a set of ordered pairs of nodes, called *edges* E (see Fig. 69(a)). If (u, v) is an edge, we say that v is *adjacent* to u (or alternately, that v is a *neighbor* of u). In most geometric settings, graphs are *undirected*, since if you get from u to v, you can get from v to u. It is often convenient to use a directed graph

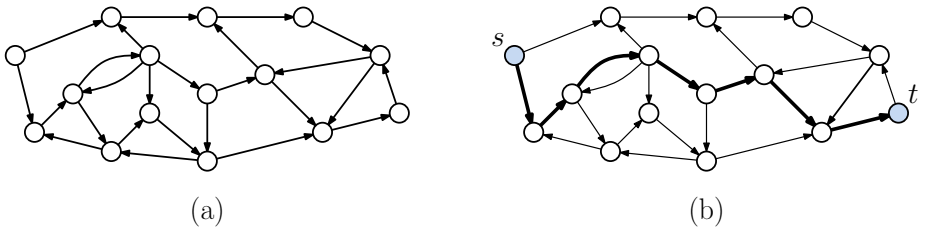


Fig. 69: A directed graph.

representation, however, since it allows you to model the fact that travel in one direction (say up hill) may be more expensive than travel in the reversed direction.

In the context of shortest paths, we assume that each edge (u, v) is associated with a numeric *weight*, $w(u, v)$. A *path* in G is any sequence of nodes $\langle u_0, \ldots, u_k \rangle$ such that (u_{i-1}, u_i) is an edge of G. The *cost* of a path is the sum of the weights of these edges $\sum_{i=1}^{k} w(u_{i-1}, u_i)$. The *shortest path problem* is, given a directed graph with weighted edges, and given a *start node* s and *destination node* t, compute a path of minimum cost from s to t (see Fig. 69(b)). Let us denote the shortest path cost from s to t in G by $\delta(s, t)$.

In earlier courses, you have no doubt seen examples of algorithms for computing shortest paths. Here are some of the better known algorithms.

Breadth-First Search (BFS): This algorithm is among the fastest algorithms for computing shortest paths, but it works under the restrictive assumption that all the edges have equal weight (which, without loss of generality, we may assume to be 1). The search starts at s, and then visits all the nodes that are connected to s by a single edge. It labels all of these nodes as being at distance 1 from s. It then visits each of these nodes one by one and visits all of their neighbors, provided that they have not already been visited. It labels each of these as being at distance 2 from s. Once all the nodes at distance 1 have been visited, it then processes all the nodes at distance 2, and so on. The nodes that are waiting to be visited are placed in a *first-in, first-out queue*. If G has n nodes and m edges, then BFS runs in time $O(n + m)$.

Dijkstra's Algorithm: Because BFS operates under the assumption that the edges weights are all equal, it cannot be applied to general weighted digraphs. Dijkstra's algorithm is such an algorithm. It makes the (not unreasonable) assumption that all the edge weights are nonnegative.[12] We will discuss Dijkstra's algorithm below, but intuitively, it operates in a greedy manner by propagating distance estimates starting from the source node to the other nodes of the graph, through an incremental process called *relaxation*. A straightforward implementation of Dijkstra's algorithm runs in $O(m \log n)$ time (and in theory even faster algorithms exist, but they are fairly complicated).

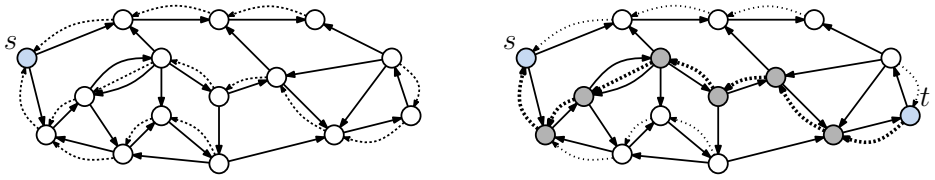
Bellman-Ford Algorithm: Since Dijkstra's algorithm fails if the graph has negative edge weights, there may be a need for a more general algorithm. The Bellman-Ford algorithm generalizes Dijkstra's algorithm by being able to handle graphs with negative edge weights, assuming there are no negative-cost cycles, that is, there is no cycle such that the sum of edge weights along the cycle is strictly smaller than zero. It runs in time $O(nm)$. (Note that the assumption that there are no negative-cost cycles is very reasonable. If such a cycle exists, the path cost could be made arbitrarily small by looping through this cycle an arbitrary number of times. Therefore, no shortest path exists.)

[12]Negative edge weights do not typically arise in geometric contexts, and so we will not worry about them. They can arise in other applications. For example, in financial applications, an edge may model a transaction where money can be made or lost. In such contexts, weights may be positive or negative. When computing shortest paths, however, it is essential that the graph have no cycles whose total cost is negative, for otherwise the shortest path is undefined.

Floyd-Warshall Algorithm: All the algorithms mentioned above actually can be used to solve a more general problem, namely the *single-source shortest path problem*. The reason is that if you run each algorithm until every node in the graph has been visited, it computes the shortest path from s to every other node. It is often useful in games to compute the shortest paths between *all pairs* of nodes, and store them in a table for efficient access. While it would be possible to do this by invoking Dijkstra's algorithm for every possible source node, an even simpler algorithm is the Floyd-Warshall algorithm. It runs in time $O(n^3)$.

Other Issues: There are a number of other issues that arise in the context of computing shortest paths.

Storing Paths: How are shortest paths represented efficiently? The simplest way is through the use of a *predecessor pointer*. In particular, each node (other than the source) stores a pointer to the node that lies immediately before it on the shortest path from s. For example, if the sequence $\langle s, u_1, \ldots, u_k \rangle$ is a shortest path, then $\text{pred}(u_k) = u_{k-1}$, $\text{pred}(u_{k-1}) = u_{k-2}$, and so on (see Fig. 70(a)). By following the predecessor pointer back to s, we can construct the shortest path, but in reverse (see Fig. 70(b)). Since this involves only a constant amount of information per node, this representation is quite efficient.



(a) (b)

Fig. 70: Storing/reconstructing the shortest path.

By the way, in the context of the all-pairs problem (Floyd-Warshall, for example) for each pair of nodes u and v, we maintain a two-dimensional array $P[u, v]$, which stores either null (meaning that the shortest path from u to v is just the edge (u, v) itself), or a pointer to *any* node along the shortest path from u to v. For example, if $P[u, v] = x$, then to chart the path from u to v, we (recursively) compute the path from u to x, and the path from x to v, and then we concatenate these two paths.

Single Destination: In some contexts, it is desirable to compute an *escape route*, that is, the shortest path from every node to some common destination. This can easily be achieved by reversing all the edges of the graph, and then running a shortest path algorithm. (This has the nice feature that the predecessor links provide the escape route.)

Closest Facility: Suppose that you have a set of locations, called *facilities*, $\{f_1, \ldots, f_k\}$. For example, these might represent safe zones, where an agent can go to in the event of danger. When an alarm is sounded, every agent needs to move to its closest facility. We can view this as a generalization of the single destination problem, but now there are multiple destinations, and we want to compute a path to the closest one.

How would we solve this? Well, you could apply any algorithm for the single-destination problem repeatedly for each of your facilities. If the number of facilities is large, this can take some time. A more clever strategy is to reduce the problem to a *single instance* of an equivalent single destination problem. In particular, create a new node, called the *super destination*. Connect all your facilities to the super destination by edges of cost zero (see Fig. 71(a)). Then apply the single destination algorithm to this instance. It is easy to see that the resulting predecessor links will point in the direction of the closest facility (see Fig. 71(b)). Note that this only requires *one* invocation of a shortest path algorithm, not k.

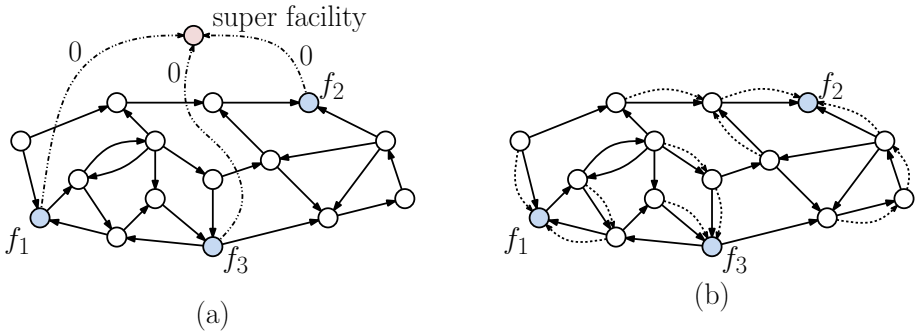


Fig. 71: Using shortest paths to compute the closest facility.

Of course, this idea can be applied to the case of multiple source points, where the goal is to find the shortest path from any of these sources.

Informed Search: BFS and Dijkstra have the property that nodes are processed in increasing order of distance from the source. This implies that if we are interested in computing just the shortest path from s to t, we can terminate either algorithm as soon as t has been visited. Of course, in the worst case, t might be the last node to be visited. Often, shortest paths are computed to destinations that are relatively near the source. In such cases, it is useful to terminate the search as soon as possible. If we are solving a single-source, single-destination problem, then it is in our interest to visit as few nodes as possible. Can we do better than BFS and Dijkstra? The answer is yes, and the approach is to use an algorithm based on informed search.

To understand why we might expect to do better, imagine that you are writing a program to compute shortest paths on campus. Suppose that a request comes to compute the shortest path from the Computer Science Building to the Art Building. The shortest path to the Art Building is 700 meters long. If you were to run an algorithm like Dijkstra, it would visit every node of your campus road map that lies within distance 700 meters of Computer Science before visiting the Art Building (see Fig. 72(a)). But, you know that the Art Building lies roughly to the west of Computer Science. Why waste time visiting a location that is 695 meters to east, since it is very unlikely to help you get to the Art Building. Dijkstra's algorithm is said to be an *uninformed* algorithm, but it makes use of no external information, such as the fact that the shortest path to a building to the west is more likely to travel towards the west, than the east. So, how can we exploit this information?

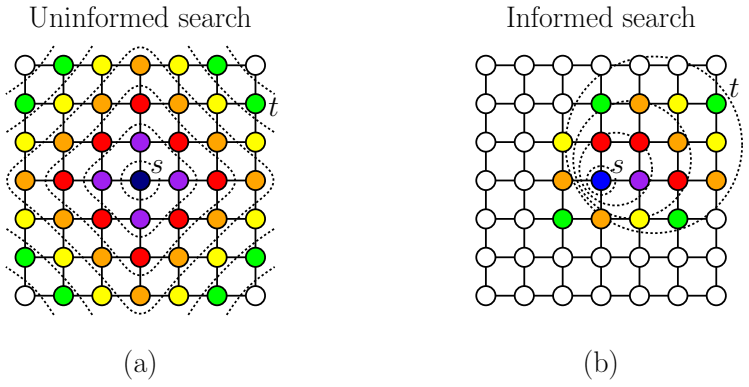


Fig. 72: Search algorithms where colors indicate the order in which nodes are visited by the algorithms: (a) uninformed search (such as Dijkstra) and (b) informed search (such as A*).

Information of the type described above is sometimes called a *heuristic*. It can be thought of as an "educated guess." An *informed search algorithm* is one that employs heuristic information to speed up the search. An example in our case would be using geometric information to direct the search to the destination (see Fig. 72(b)). If your heuristics are good, then they can be of significant benefit. Ideally, of course, even if your heuristics are bad, you still want a correct answer (although it might take longer to compute it).

Informed and Uninformed Search Algorithms: To develop this idea further, lets begin by presenting a simple implementation of Dijkstra's algorithm. (See the code block below.) Vertices are in one of three possible states: *undiscovered* (not yet seen), *discovered* (seen but not yet processed), and *finished* (processed). When a node u has been processed, its associated d-value should equal the actual cost of the shortest path from s to u, that is $d[u] = \delta(s, u)$. Thus, when t has been reached, $d[t]$ is the desired cost. (For simplicity, we ignore storing predecessor links, but that is an easy addition.)

Dijkstra's Algorithm

```
Dijkstra(G, s, t) {
  foreach (node u) {                      // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered            // distance to source is 0
  while (true) {                          // go until finding t
    let u be the discovered node that minimizes d[u]
    if (u == t) return d[t]               // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))   // update d[v]
        mark v discovered
      }
      mark u finished                     // we're done with u
  }
}
```

Best-First Search: What sort of heuristic information could we make use of to better inform the choice of which vertex u to process next? We want to visit the vertex that we think will most likely lead us to t quickly. Assuming that we know the spatial coordinates of all the nodes of our graph, one idea for a heuristic is the Euclidean distance from the node u to the destination t. Given two nodes u and v, let $\text{dist}(u, v)$ denote the Euclidean (straight-line) distance between u and v. Euclidean distance disregards obstacles, but intuitively, if a node is closer to the destination in Euclidean distance, it is likely to be closer in graph distance. Define the *heuristic function* $h(u) = \text{dist}(u, t)$. Greedily selecting the node that minimizes the heuristic function is called *best-first search*. Do *not* confuse this with breadth-first search, even though they share the same three-letter acronym. (See the code block below, as an example.)

Unfortunately, when obstacles are present it is easy to come up with examples where best-first search can return an incorrect answer. By using the Euclidean distance, it can be deceived into wandering into dead-ends, which it must eventually backtrack out of. (Note that once the algorithm visits a vertex, its d-value is fixed and never changes.)

A* Search: Since best-first search does not work, is there some way to use heuristic information to produce a correct search algorithm? The answer is yes, but the trick is to be more clever in how we use the heuristic function. Rather than just using the heuristic function $h(u) = \text{dist}(u, t)$ alone to select the next node to process, let us use both $d[u]$ and $h(u)$. In particular, $d[u]$ represents an estimate on the

Best-First Search

```
BestFirst(G, s, t) {
  foreach (node u) {                        // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered              // distance to source is 0
  repeat forever {                          // go until finding t
    let u be the discovered node that minimizes dist(u,t)
    if (u == t) return d[t]                 // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))     // update d[v]
        mark v discovered
      }
      mark u finished                       // we're done with u
  }
}
```

cost of getting from s to u, and $h(u)$ represents an estimate on the cost of getting from u to t. So, how about if we take their sum? Define

$$f(u) \;=\; d[u] + h(u) \;=\; d[u] + \text{dist}(u,t).$$

We will select nodes to be processed based on the value of $f(u)$. This leads to our third algorithm, called *A^*-search*. (See the code block below.)

A-Star Search

```
A-Star(G, s, t) {
  foreach (node u) {                        // initialize
    d[u] = +infinity;  mark u undiscovered
  }
  d[s] = 0;  mark s discovered              // distance to source is 0
  repeat forever {                          // go until finding t
    let u be the discovered node that minimizes d[u] + dist(u,t)
    if (u == t) return d[t]                 // arrived at the destination
    else {
      for (each unfinished node v adjacent to u) {
        d[v] = min(d[v], d[u] + w(u,v))     // update d[v]
        mark v discovered
      }
      mark u finished                       // we're done with u
  }
}
```

While this might appear to be little more than a "tweak" of best-first search, this small change is exactly what we desire. In general, there are two properties that the heuristic function $h(u)$ must satisfy in order for the above algorithm to work.

Admissibility: The function $h(u)$ *never overestimates* the graph distance from u to t, that is $h(u) \leq \delta(u,t)$. It is easy to see that this is true, since $\delta(u,t)$ must take into account obstacles, and so can never be smaller than the straight-line distance $h(u) = \text{dist}(u,t)$. A heuristic function is said to be *admissible* if this is the case.

Consistency: A second property that is desirable (for the sake of efficiency) states that, for any two nodes u' and u'', we have $h(u') \leq \delta(u', u'') + h(u'')$. Intuitively, this says that the heuristic cost of getting from u' to the destination cannot be larger than the graph cost from u' to u'' followed by the heuristic cost from u'' to the destination. Such a heuristic is said to be *consistent* (or *monotonic*). This can be viewed as a generalization of the *triangle inequality* from geometry (which states that the sum of two sides of a triangle cannot be smaller than the other side). Consistency follows for our heuristic from the triangle inequality:

$$\begin{aligned} h(u') &= \text{dist}(u', t) \leq \text{dist}(u', u'') + \text{dist}(u'', t) \qquad \text{(by the triangle inequality)} \\ &\leq \delta(u', u'') + \text{dist}(u'', t) = \delta(u', u'') + h(u''). \end{aligned}$$

It turns out that admissibility alone is sufficient to show that A* search is correct, but like a graph with negative edge weights, the search algorithm is not necessarily efficient, because we might declare a node to be "finished," but later we will discover a path of lower cost to this vertex, and will have to move it back to the "discovered" status. (This is similar to what happens in the Bellman-Ford algorithm). However, if both properties are satisfied, A* runs in essentially the same time as Dijkstra's algorithm in the worst case, and may actually run faster. The key to the efficiency of the search algorithm is that along any shortest path from s to t, the f-values are nondecreasing. To see why, consider two nodes u' and u'' along the shortest path, where u' appears before u''. Then we have

$$\begin{aligned} f(u'') &= d[u''] + h(u'') = d[u'] + \delta(u', u'') + h(u'') \\ &\geq d[u'] + h(u') = f(u'). \end{aligned}$$

Although we will not prove this formally, this is exactly the condition used in proving the correctness of Dijkstra's algorithm, and so it follows as a corollary that A* is also correct. It is interesting to note, by the way, that Dijkstra's algorithm is just a special case of A*, where $h(u) = 0$. Clearly, this is an admissible heuristic (just not a very interesting one).

Examples: Let us consider the execution of each of these algorithms on a common example. The input graph is shown in Fig. 73. For Best-First and A* we need to define the heuristic $h(u)$. To save us from dealing with square roots, we will use a different notion of geometric distance. Define the L_1 (or Manhattan) distance between two points to be the sum of the absolute values of the difference of the x and y coordinates. For example, in the figure the L_1 distance between nodes f and t is $\text{dist}_1(f, t) = 3 + 6 = 9$. For both best-first and A* define the heuristic value for each node u to be L_1 distance from u to t. For example, $h(f) = 9$. (Because the edge weights have been chosen to match the L_1 length of the edge, it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

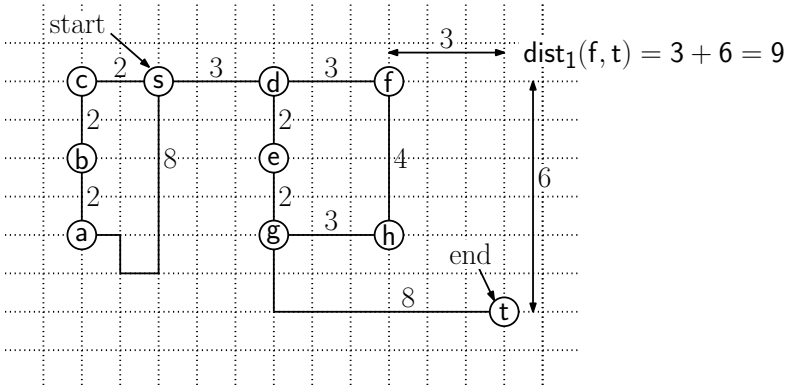


Fig. 73: The graph G used in the sample runs.

Dijkstra's Algorithm: The table below provides a trace of Dijkstra's algorithm. For each discovered node we show the value $d[u]$. At each stage (each row of the table), the node with the lowest d-value is selected next for processing. Once processed, a node's d-value never changes (as indicated by the down arrow). Note that at Stage 6 we could have processed either a or f, since they have the same d-values.

Dijkstra's Algorithm – Each entry is $d[u]$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
Init	$\underline{0}$	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	**0**	8	–	$\underline{2}$	3	–	–	–	–	–
2: c	$\downarrow$	8	4	**2**	$\underline{3}$	–	–	–	–	–
3: d		8	$\underline{4}$	$\downarrow$	**3**	5	6	–	–	–
4: b		6	**4**		$\downarrow$	$\underline{5}$	6	–	–	–
5: e		$\underline{6}$	$\downarrow$			**5**	6	7	–	–
6: a		**6**				$\downarrow$	$\underline{6}$	7	–	–
7: f		$\downarrow$					**6**	$\underline{7}$	10	–
8: g							$\downarrow$	**7**	$\underline{10}$	15
9: h								$\downarrow$	**10**	$\underline{15}$
10: t									$\downarrow$	**15**
Final	**0**	**6**	**4**	**2**	**3**	**5**	**6**	**7**	**10**	**15**

An intuitive way to think about how Dijkstra's algorithm operates is to imagine fluid flooding out from the source node s along the edges simultaneously. The first node that is hit by this fluid is processed, and begins propagating the fluid along its edges as well. This idea is loosely illustrated in Fig. 74, where the different colors indicate stages of the flooding algorithm For example, first red fluid is flooded out of s. The first vertex to be hit is c (at distance 2). The next phase of flooding is indicated by dark blue. It floods out along c's edges and continues to flood along s's edges. The first vertex to be hit by the blue flood is vertex d, which is processed at a distance of 3 units. While Dijkstra's algorithm does not explicitly track these flows, it processes nodes in the order in which the flooding fluid reaches the nodes.

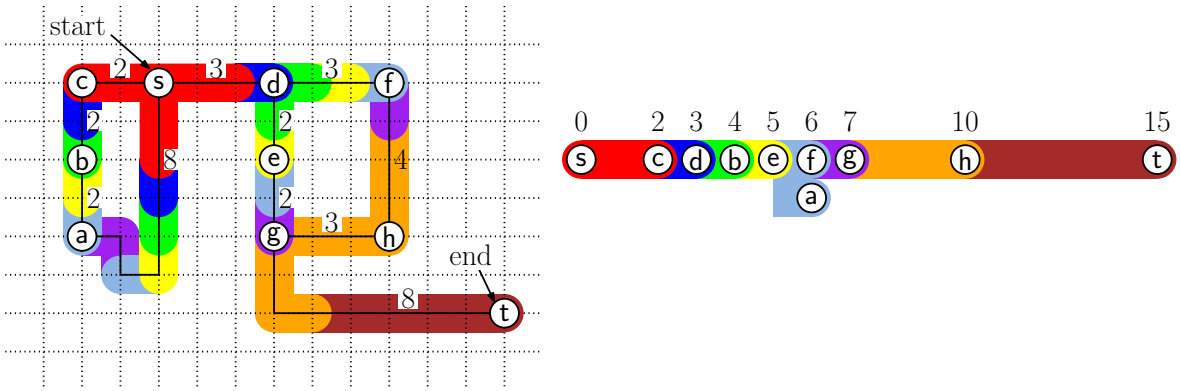


Fig. 74: Illustrating the propagation of distance information in Dijkstra's algorithm.

Best-First Search: The table below shows the trace of best-first search. For each discovered node we show the value $d[u] : h(u)$. At each stage, the discovered node with the smallest h-value (underlined) is chosen to be the next to be processed. Once processed, a node's d-value never changes (as indicated by the down arrow).

Best-First Search – Each entry is $d[u] : h(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	15	13	15	17	12	10	9	8	5	0
Init	0:15	∞:13	∞:15	∞:17	∞:12	∞:10	∞:9	∞:8	∞:5	∞:0
1: s	**0**	8:13	–	2:17	3:12	–	–	–	–	–
2: d	↓	8:13	–	2:17	**3**	5:10	6:9	–	–	–
3: f		8:13	–	2:17	↓	5:10	**6**	–	10:5	–
4: h		8:13	–	2:17		5:10	↓	13:8	**10**	–
5: g		8:13	–	2:17		5:10		**13**	↓	21:0
6: t		8:13	–	2:17		5:10		↓		**21**
Final	**0**	**8**	∞	**2**	**3**	**5**	**6**	**13**	**10**	**21**??

Not that Best-first determines that $d[t] = 21$, which is *incorrect* (it should be 15).

A* search: The table below shows the trace of the A* algorithm. For each discovered node we show the value $d[u] : h(u)$. At each stage, the discovered node with the smallest value of $d[u] + h[u]$ (underlined) is chosen to be the next to be processed. Once processed, a node's d value never changes (as indicated by the down arrow). Note that at Stages 3, 4, and 5 we have a choice of nodes to process next since there are multiple nodes with the same $d[u] + h(u)$ values.

A* Search – Each entry is $d[u] : f(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	15	13	15	17	12	10	9	8	5	0
Init	0:15	∞:13	∞:15	∞:17	∞:12	∞:10	∞:9	∞:8	∞:5	∞:0
1: s	**0**	8:13	–	2:17	3:12	–	–	–	–	–
2: d	↓	8:13	–	2:17	**3**	5:10	6:9	–	–	–
3: e		8:13	–	2:17	↓	**5**	6:9	7:8	–	–
4: f		8:13	–	2:17		↓	**6**	7:8	–	15:0
5: t		8:13	–	2:17			↓	7:8	–	**15**
Final	0	8	∞	2	3	5	6	7	∞	15

Note that the algorithm computes the correct result, but it terminates after just five stages, not ten as was the case for Dijkstra's algorithm.

ASTAR search can also be thought of as simulating fluid propagation, but the analogy is not as clear as in Dijkstra's algorithm. The order in which nodes are processed depends now on two things. First, the distance that the fluid needs to travel from s to reach the node (as given by the value $d[u]$ for node u) and the heuristic value $h[u]$. In Fig. 75, we illustrate the fluid flows as we did for Dijkstra's algorithm, and we use a colored line to indicate the length of the $h[u]$ value. In this case, because we are using the sum of horizontal and vertical distances as the heuristic values, these heuristic paths consist of a single horizontal and a single vertical edge.

A* with inadmissible heuristic: The table below shows the trace of the A* algorithm using the heuristic $h'(u) = 10 \cdot \text{dist}_1(u, t)$, which is not admissible for this input. For each discovered node we show the value $d[u] : h'(u)$. At each stage, the discovered node with the smallest value of $d[u] + h'(u)$ (underlined) is chosen to be the next to be processed. Once processed, a node's d value never changes (as indicated by the down arrow).

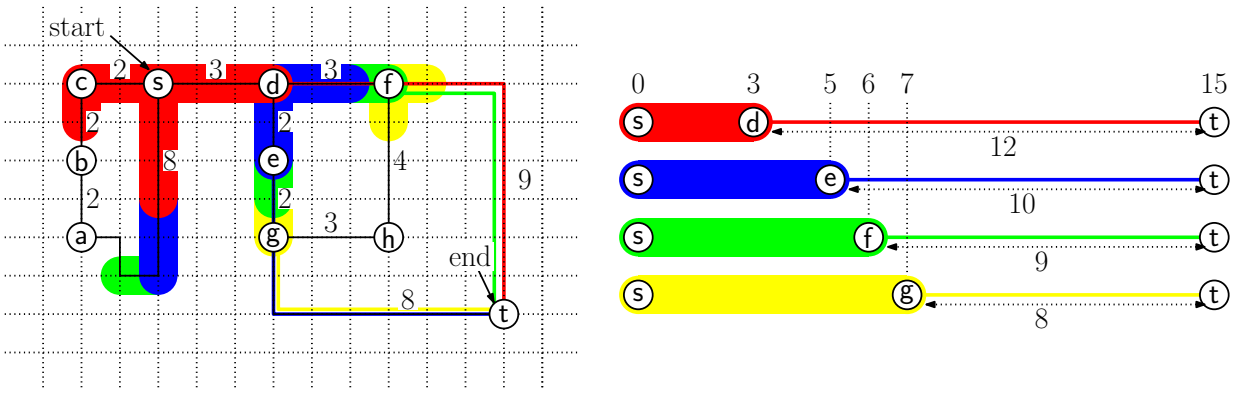


Fig. 75: Illustrating the propagation of distance information in A* search.

A* Search – Each entry is $d[u] : f'(u)$										
Stage	$d[s]$	$d[a]$	$d[b]$	$d[c]$	$d[d]$	$d[e]$	$d[f]$	$d[g]$	$d[h]$	$d[t]$
$h(u)$	150	130	150	170	120	100	90	80	50	0
Init	0:150	∞:130	∞:150	∞:170	∞:120	∞:100	∞:90	∞:80	∞:50	∞:0
1: s	**0**	8:130	–	2:170	3:120	–	–	–	–	–
2: d	↓	8:130	–	2:170	**3**	5:100	6:90	–	–	–
3: f		8:130	–	2:170	↓	5:100	**6**	–	10:50	–
4: h		8:130	–	2:170		5:100	↓	13:80	**10**	–
5: g		8:130	–	2:170		5:100		**13**	↓	21:0
6: t		8:130	–	2:170		5:100		↓		**21**
Final	**0**	**8**	∞	**2**	**3**	**5**	**6**	**13**	**10**	21??

Observe that this heuristic boosts the h-values so high that they dominate when computing the f-values. As a result, the algorithm effectively determines which nodes to process based on the h-values alone, and so the algorithm behaves in essentially the same manner as best-first search, and computes the same incorrect result.

Lecture 17: Artificial Intelligence for Games: Decision Making

Decision Making: Designing general-purpose AI systems that are capable of modeling interesting behaviors is a challenging task. On the one hand, we would like our AI system to be general enough to provide a game designer the ability to specify the subtle nuances that make a character interesting. On the other hand we would like the system to be easy to use and powerful enough that relatively simple behaviors can be designed with ease. It would be nice to have a library of different behaviors and different mechanisms for combining these behaviors in interesting ways.

Today we will discuss a number of different methods, ranging from fairly limited to more complex. In particular, we will focus on three different approaches, ranging from simple to more sophisticated.

- Decision trees (and variants)
- Finite state machines (and variants)
- Behavior trees

At an extreme level of abstraction, you might wonder what the big deal is. After all, a decision making process quite simply is a (possibly randomized) algorithm that maps a set of input conditions into

a set of actions. Thus, it could be implemented using any programming language. Indeed, scripting languages are often used for encoding decision-making systems.

The various techniques that we are going to present could all be implemented via direct implementation in any programming or scripting language. The problem with expressing complex behaviors through programs is that, due to their extremely general nature, programs can be difficult to understand and difficult to reason about. While the aforementioned techniques are limited in their capabilities, they do provide game designers a clean, visually-based structure in which to describe and reason about the behaviors they represent.

Decision Trees: As a starting point, let's consider perhaps the simplest structuring device for implementing a decision-making procedure, the *decision trees*. Decision trees are fast, easy to implement, and simple to understand. A decision tree is a rooted (typically) binary tree, where each node, or *decision point*, is labeled with a test, and the children of this node correspond to the possible outcomes of this test. An example of a decision tree for an combatant NPC is shown in Fig. 76.

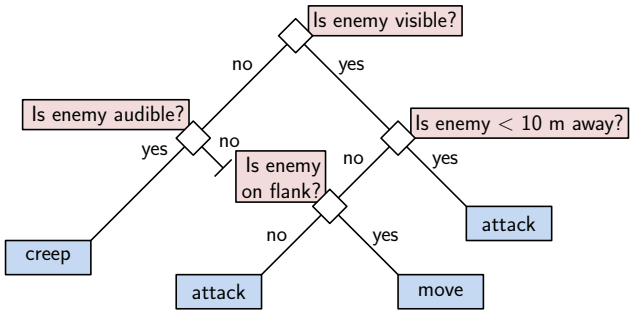


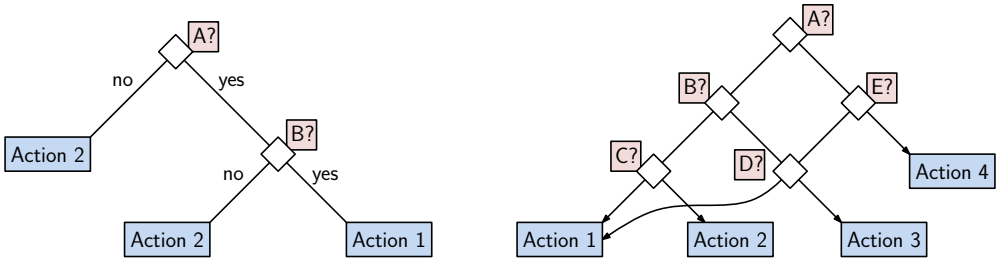
Fig. 76: A decision-tree for a combatant agent.

In the example the decisions were all binary, but this does not need to be the case. For example, as with switch statements in Java, it would be possible to have a node with multiple children, where each child corresponds to one of the possible value types.

Variations on a Theme: While our example showed just simple boolean conditions, you might wonder whether it is possible to express more complex conditions using decision trees. The short answer is yes, but it takes a little work. In fact, any decision making algorithm based on a finite sequence of discrete conditions can be expressed in this manner. For example, suppose you have two boolean tests A and B, and you want Action 1 to be performed if both A *and* B are satisfied, and otherwise you want Action 2 to be performed. This could be encoded using the decision tree shown in Fig. 76(a). (Note that encoding a boolean *or* condition is equally easy. Try it.)

Observe that in order to achieve the more complex boolean-and condition, we needed to make two copies of the "Action 2" node. Replicating leaf nodes is not a major issue, since each such node would presumably just contain a pointer to a function that implements this action. On the other hand, if we wanted to share entire subtree structures, replicating these subtrees (especially if it is done recursively) can quickly add up to a lot of space. Furthermore, copying is an error-prone process, since any amendment to one subtree would require making the same change in all the others (assuming you want all of them to implement the same decision-making procedure.) One way to avoid the issue of copying subtrees is allow subtrees to be shared (see Fig. 77(b)). In spite of the name "decision trees," the resulting "decision acyclic directed graphs" are just as easy to search.

Another variation on decision trees is to introduce *randomization* to the decision-tree. For example, we might have a decision point that says "flip a coin" (or more generally, generate a random integer over some finite range with a given probability distribution). Based on the result of this random choice, we



(a) (b)

Fig. 77: (a) Complex boolean conditions and (b) subtree sharing.

could then branch among various actions. This would allow us to add more variation to the behavior of our agents.

Implementing Decision Trees: Decision trees can be implemented in a number of ways. If the tree is small (and it is a tree, as opposed to a directed-acyclic graph) you can translate the tree into an appropriate layered if-then-else statement in your favorite programming/scripting language. More generally, you can express the tree as a graph-based data structure, where internal nodes hold pointers to predicate function and leaf nodes hold pointers to action functions.

Finite State Machines: Decision trees are really too simple to be used for most interesting decision-making processing. The next step up in complexity is to add a notion of *state* to the character, and then make decisions a function of both the current conditions and the character's current state.

For example, a character may behave more aggressively when it is healthy and less aggressively when it is injured. As another example, a designer may wish to have a character transition between various states (patrolling, chasing, fighting, retreating) in sequence or when triggered by game events. In each state, the characters behavior may be quite different.

A *finite state machine* (FSM) can be modeled as a directed graph, where each node of the graph corresponds to a state, and each directed edge corresponds to a event, that triggers a change of state and optionally some associated action. The associated actions may include things like starting an animation, playing a sound, or modifying the current game state.

As an example, consider the programming of an warrior bot NPC in a first-person shooter. Suppose that as the designer you decide to implement the following type of behavior:

- If you don't see an enemy, stand guard
- While on guard, if you see a small enemy, fight it, but if it is too big, then flee
- If you are fighting, if you discover you are losing the fight, then flee
- While fleeing, if you have escaped to a point where there is no threat, then return to the guarding state

We can encode this behavior in the form of the FSM showed in Fig. 78(a).

FSMs are a popular method of defining behavior in games. They are easy to implement, easy to design (if they are not too big), and they are easy to reason about. For example, based on the visual layout of the FSM, it is easy to see the conditions under which certain state transitions can occur and whether there are missing transitions (e.g., getting stuck in some state forever).

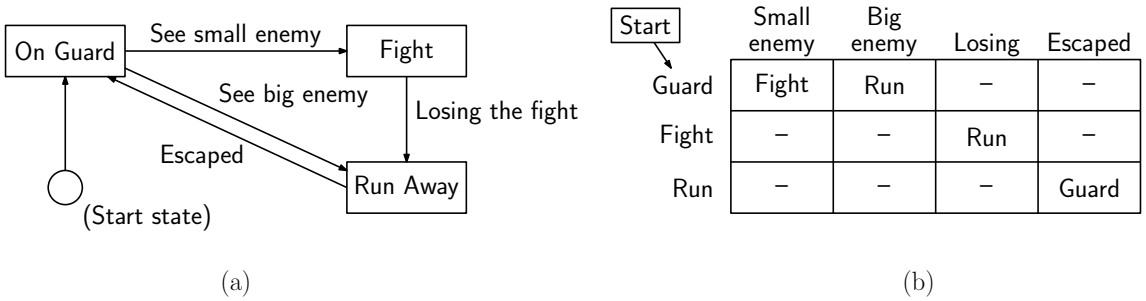


Fig. 78: A simple state machine for a warrior bot.

Implementing State Machines: How are FSMs implemented? A natural implementation is to use a two-dimensional array, where the row index is an encoding of the current state and the column index is an encoding of the possible events that may trigger a transition. Each entry of the array is labeled with the state to which the transition takes place (see Fig. 78(b)). The array will also contain further information, such as what actions and animations to trigger as part of the action.

As can be seen from the example shown in the figure, many state-event pairs result in no action or transition. If this is the case, then the array-based implementation can be space inefficient. A more efficient alternative would be to use the nonempty state-event pairs as keys into a hash table. Assuming a good hash-table implementation, the hash table's size would generally be proportional to the number nonempty entries.

Note that the FSM we have showed is *deterministic*, meaning that there is only a single transition that can be applied at any time. More variation can be introduced by allowing multiple transitions per event, and then using randomization to select among them (again, possibly with weights so that some transitions are more likely than others).

Hierarchical State Machines: One of the principal shortcoming with FSMs is that the number of states can *explode* as the designer dreams up more complex behavior, thus requiring more states, more events, and hence the need to consider a potentially quadratic number of mappings from all possible states to all possible events.

For example, suppose that you wanted to model multiple conditions simultaneously. A character might be *healthy/injured*, *wandering/chasing/attacking*, *aggressive/defensive/neutral*. If any combination of these qualities is possible, then we would require $2 \cdot 3 \cdot 3 = 18$ distinct states. This would also result in a number of repeated transitions. (For example, all nine of the states in which the character is "healthy" would need to provide transitions to the corresponding "injured" states if something bad happens to us. Requiring this much redundancy can lead to errors, since a designer may update some of the transitions, but not the others.)

One way to avoid the explosion of states and events is to design the FSM in a hierarchical manner. First, there are a number of high-level states, corresponding to very broad contexts of the character's behavior. Then within each high-level state, we could have many sub-states, which would be used for modeling more refined behaviors within this state. The resulting system is called a *hierarchical finite state machine* (HFSM).

For example, suppose we add an additional property to our warrior bot, namely that he/she gets hungry from time to time. When this event takes place, the bot runs to his/her favorite restaurant to eat. When the bot is full, he/she returns to the same state. Since this event can occur from within any state, we would need to add these transitions from all the existing states (see Fig. 79).

Of course, this would get to be very tedious if we had a very large FSM. The solution, is to encapsulate most of the guarding behaviors within one FSM, and then treat this as a single *super-state* in a

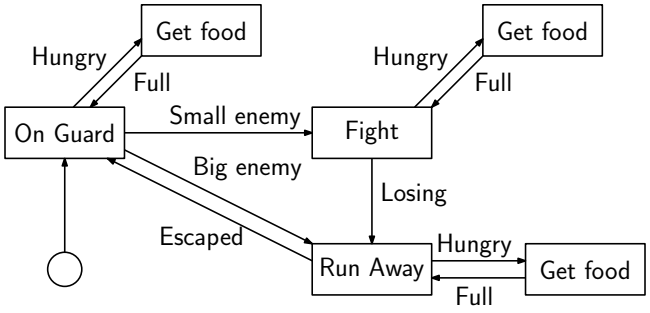


Fig. 79: State machine for a hungry warrior bot.

hierarchical FSM (see Fig. 80). The hungry/full transitions would cause us to save the current state (e.g., by pushing it onto a stack), performing the transition. On returning to the state, we would pop the stack and then resume our behavior as before.

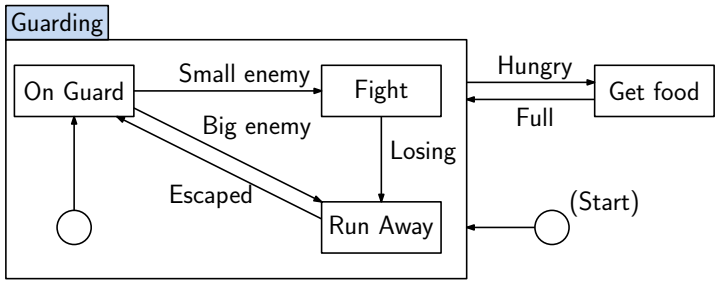


Fig. 80: Hierarchical state machine for a hungry warrior bot.

Note that we create a start state within the super-state. This is to handle the first time that we enter the state. After this, however, we always return to the same state that we left from.

This could be implemented, for example, by storing the state on a stack, where the highest-level state descriptor is pushed first, then successively more local states.

The process of looking up state transitions would proceed hierarchically as well. First, we would check whether the lowest level sub-state has any transition for handling the given event. If not, we could check its parent state in the stack, and so on, until we find a level of the FSM hierarchy where this event is to be handled.

The advantage of this hierarchical approach is that it naturally adds modularity to the design process. Because the number of local sub-states is likely to be fairly small, it simplifies the design of the FSM. In particular, we can store even a huge number of states because each sub-state level need only focus on the relatively few events that can cause transitions at this level.

FSM-Based Adventure Games: Very early text-based adventure games were based on finite-state automata. The earliest example was *Colossal Cave Adventure* by Will Crowther in the 1970's. It was implemented in Fortran and ran on a PDP-10 computer. The game-play involves a series of short descriptions, after which the player could enter simple commands. The objective was to navigate through the environment to find the treasure. Here is a short example:

```
You are standing at the end of a road before a small brick building. Around you is
a forest. A small stream flows out of the building and down a gully.
```

```
> enter building

You are inside a building, a well house for a large spring. There are some keys on
the ground here. There is a shiny brass lamp nearby. There is tasty food here.
There is a bottle of water here.

> take keys

Taken

> go south

You are in a valley in the forest beside a stream tumbling along a rocky bed.
```

It is not hard to see how this could be implemented using an FSM. The player's current position is modeled as the FSM state (with auxiliary information for the player's inventory), and commands were mapped to state transitions, and each state is associated with a short description.

Behavior Trees: While FSMs are general, they are not that easy to design. We would like a system that is more general the FSMs, more structured than programs, and lighter weight than general-purpose planners. Behavior trees were developed by Geoff Dromey in the mid-2000s in the field of software engineering, which provides a modular way to define software in terms of actions and preconditions. They were first used in Halo 2 and were adopted by a number of other games such as Spore.

Let us consider the modeling of a *guard dog* in an FPS game. The guard dog's range of behaviors can be defined hierarchically. At the topmost level, the dog has behaviors for major tasks, such as *patrolling*, *investigating*, and *attacking* (see Fig. 81(a)). Each of these high-level behaviors could then be broken down further into lower-level behaviors. For example, the patrol task may include a subtask for *moving*. The investigate task might include a subtask for *looking around*, and the attack task may include a subtask for *bite* (ouch!).

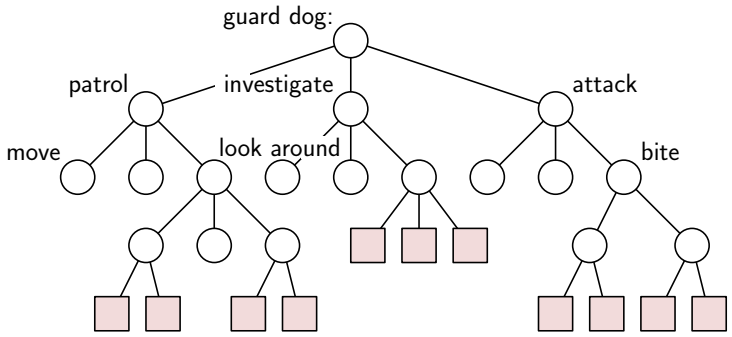


Fig. 81: Sample hierarchical structure of a guard-dog's behavior.

The leaves of the tree are where the AI system interacts with the game state. Leaves provide a way to gather information from the system through *conditions*, and a way to affect the progress of the game through *actions*. In the case of our guard dog, conditions might involve issues such as the dog's state (is the dog hungry or injured) or geometric queries (is there another dog nearby, and is there a line of sight to this dog?). Conditions are *read-only*. Actions make changes to the world state. This might involve performing an animation, playing a sound, picking up an object, or biting someone (which would presumably alter this other object's state). Conditions can be thought of as *filters* that indicate which actions are to be performed.

A *task* is a piece of code that models a latent computation. A task consists of a collection *conditions* that determine when the task is enabled and *actions*, which encode the execution of the task. Tasks can end either in *success* or *failure*.

Composing Tasks: *Composite tasks* provide a mechanism for composing a number of different tasks. There are two basic types of composite tasks, which form natural complements.

Sequences: A sequence task performs a series of tasks sequentially, one after the other (see Fig. 82(a)). As each child in the sequence succeeds, we proceed to the next one. Whenever a child task fails, we terminate the sequence and bail out (see Fig. 82(b)). If all succeed, the sequence returns success.

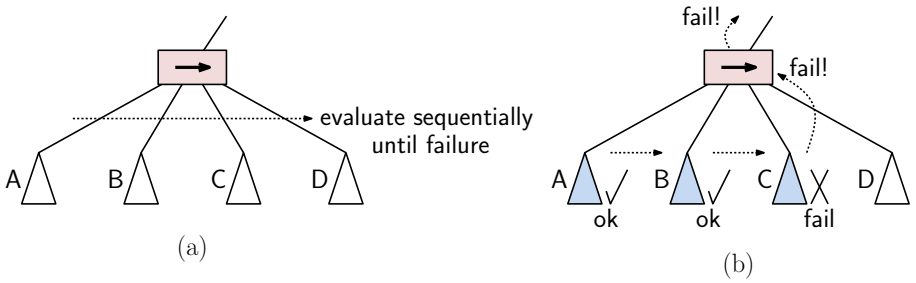


Fig. 82: Sequence: (a) structure and (b) semantics.

Selector: A selector task performs at most one of a collection of child tasks. A selector starts by selecting the first of its child tasks and attempts to execute it. If the child succeeds, then the selector terminates successfully. If the child fails, then it attempts to execute the next child, and so on, until one succeeds (see Fig. 83(b)). If none succeed, then the selector returns failure.

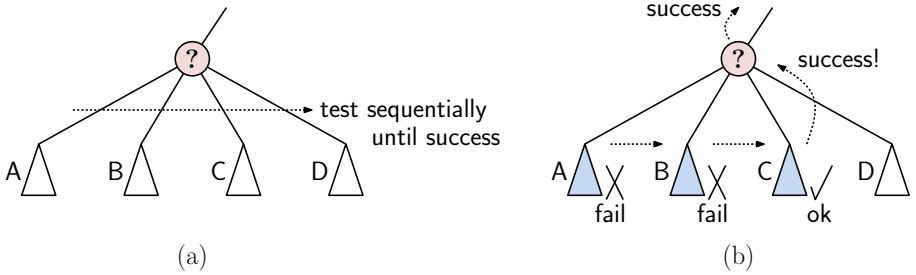


Fig. 83: Selector: (a) structure and (b) semantics.

An example of a behavior tree is presented in Fig. 84 for an enemy trying to enter a room. If the door is open, the enemy moves directly into the room (left child of the root). Otherwise, the enemy approaches the door and tries the knob. If it is unlocked, it opens the door. If locked, it breaks the door down. After this, it enters the room.

Sequences and selectors provide some of the missing elements of FSMs, but they provide the natural structural interface offered by hierarchical finite state machines. Sequences and selectors can be combined to achieve sophisticated combinations of behaviors. For example, a behavior might involve a sequence of tasks, each of which is based on making a selection from a list of possible subtasks. Thus, they provide building blocks for constructing more complex behaviors.

From a software-engineering perspective, behavior trees give a programmer a more structured context in which to design behaviors. The behavior-tree structure forces the developer to think about the

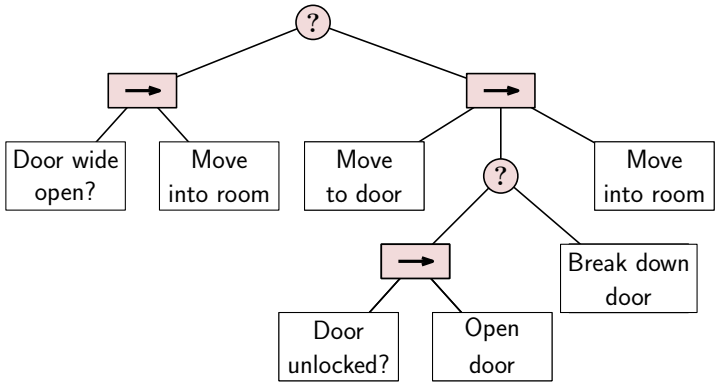


Fig. 84: Example of a behavior tree for an enemy agent trying to enter a room.

handling of success and failure, rather than doing so in an ad hoc manner, as would be the case when expressing behaviors using a scripting language. Note that the nodes of the tree, conditions and tasks, are simply links to bits of code that execute the desired test or perform the desired action. The behavior tree provides the structure within which to organize these modules.

Lecture 18: Procedural Generation: 1D Perlin Noise

Procedural Generation: Complex AAA games hire armies of designers to create the immense content that make up the game's virtual world. If you are designing a game without such extensive resources, an attractive alternative for certain natural phenomena (such as terrains, trees, and atmospheric effects) is through the use of *procedural generation*. With the aid of a random number generator, a high quality procedural generation system can produce remarkably realistic models. Examples of such systems include *terragen* (see Fig. 85(a)) and *speedtree* (see Fig. 85(b)).

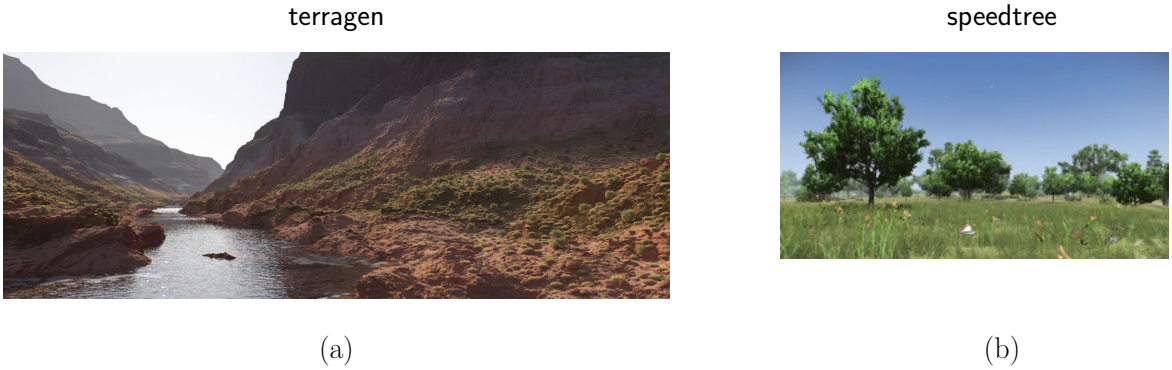


Fig. 85: (a) A terrain generated by *terragen* and (b) a scene with trees generated by *speedtree*.

Procedural model generation is a useful tool in developing open-world games. For example, the game *No Man's Sky* uses procedural generation to generate a universe of (so it is estimated) $1.8 \cdot 10^{19}$ different planets, all with distinct ecosystems, including terrains, flora, fauna, and climates (see Fig. 86). The structure of each planet is not stored on a server. Instead, each is generated deterministically by a 64-bit seed.

Before discussing methods for generating such interesting structures, we need to begin with a back-

Fig. 86: No Man's Sky.

ground, which is interesting in its own right. The question is how to construct random noise that has nice structural properties. In the 1980's, Ken Perlin came up with a powerful and general method for doing this (for which he won an Academy Award!). The technique is now widely referred to as Perlin Noise.

Perlin Noise: Natural phenomena derive their richness from random variations. In computer science, pseudo-random number generators are used to produce number sequences that appear to be random. These sequences are designed to behave in a totally random manner, so that it is virtually impossible to predict the next value based on the sequence of preceding values. Nature, however, does not work this way. While there are variations, for example, in the elevations of a mountain or the curves in a river, there is also a great deal of structure present as well.

One of the key elements to the variations we see in natural phenomena is that the magnitude of random variations depends on the scale (or size) at which we perceive these phenomena. Consider, for example, the textures shown in Fig. 87. By varying the frequency of the noise we can obtain significantly different textures.

Fig. 87: Perlin noise used to generate a variety of displacement textures.

The tendency to see repeating patterns arising at different scales is called *self similarity* and it is fundamental to many phenomena in science and nature. Such structures are studied in mathematics under the name of *fractals*. Perlin noise can be viewed as a type of random noise that is self similar at different scales, and hence it is one way of modeling random fractal objects.

Noise Functions: Let us begin by considering how to take the output of a pseudo-random number generator and convert it into a smooth (but random looking) function. To start, let us consider a sequence of random numbers in the interval $[0, 1]$ produced by a random number generator (see Fig. 88(a)). Let $Y = \langle y_0, \ldots, y_n \rangle$ denote the sequence of random values, and let us plot them at the uniformly places points $X = \langle 0, \ldots, n \rangle$.

Next, let us map these points to a continuous function, we could apply linear interpolation between pairs of points (also called *piecewise linear interpolation*. As we have seen earlier this semester, in order

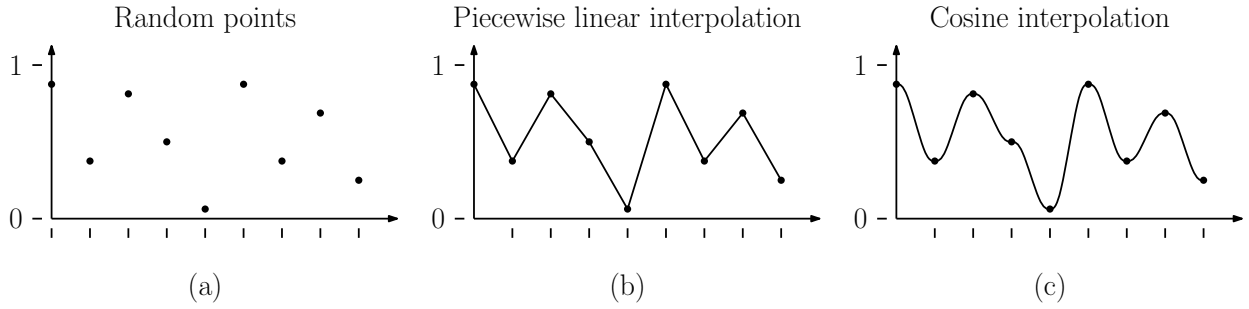


Fig. 88: (a) Random points, (b) connected by linear interpolation, and (c) connected by cosine interpolation.

to interpolate linearly between two values y_i and y_{i+1}, we define a parameter α that varies between 0 and 1, the interpolated value is

$$\text{lerp}(y_i, y_{i+1}, \alpha) \;=\; (1-\alpha)y_i + \alpha y_{i+1}.$$

To make this work in a piecewise setting we need to set α to the fractional part of the x-value that lies between i and $i+1$. In particular, if we define $x \bmod 1 = x - \lfloor x \rfloor$ to be the fractional part of x, we can define the linear interpolation function to be

$$f_\ell(x) \;=\; \text{lerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is the function shown in Fig. 88(b).

While linear interpolation is easy to define, it will not be sufficient smooth for our purposes. There are a number of ways in which to define smoother interpolating functions. (This is a topic that is usually covered in computer graphics courses.) A quick-and-dirty way to define such an interpolation is to replace the linear blending functions $(1-\alpha)$ and α in linear interpolation with smoother functions that have similar properties. In particular, observe that α varies from 0 to 1, the function $1-\alpha$ varies from 1 down to 0 while α goes the other way, and for any value of α these two functions sum to 1 (see Fig. 89(a)). Observe that the functions $(\cos(\pi\alpha)+1)/2$ and $(1-\cos(\pi\alpha))/2$ behave in exactly this same way (see Fig. 89(b)). Thus, we can use them as a basis for an interpolation method.

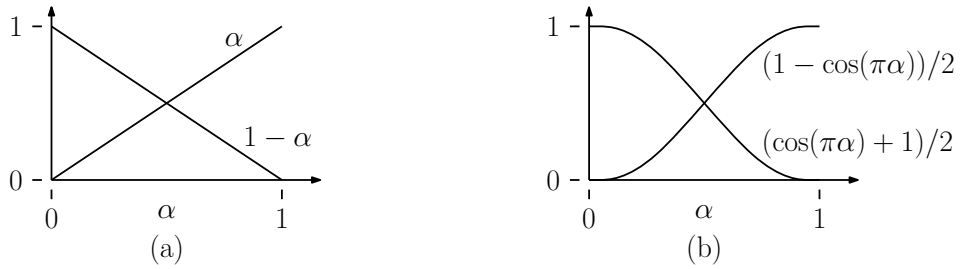


Fig. 89: The blending functions used for (a) linear interpolation and (b) cosine interpolation.

Define $g(\alpha) = (1-\cos(\pi\alpha))/2$. The cosine interpolation between two points y_i and y_{i+1} is defined:

$$\text{cerp}(y_i, y_{i+1}, \alpha) \;=\; (1-g(\alpha))y_i + g(\alpha)y_{i+1},$$

and we can extend this to a sequence of points as

$$f_c(x) \;=\; \text{cerp}(y_i, y_{i+1}, \alpha), \qquad \text{where } i = \lfloor x \rfloor \text{ and } \alpha = x \bmod 1.$$

The result is shown in Fig. 88(c). While cosine interpolation does not generally produce very good looking results when interpolating general data sets. (Notice for example the rather artificial looking flat spot as we pass through the fourth point of the sequence.) Interpolation methods such as cubic interpolation and Hermite interpolate are preferred. It is worth remembering, however, that we are interpolating random noise, so the lack of "goodness" here is not much of an issue.

Layering Noise: Our noise function is continuous, but there is no self-similarity to its structure. To achieve this, we will need to combine the noise function in various ways. Our approach will be similar to the approach used in the harmonic analysis of functions.

Recall that when we have a periodic function, like $\sin t$ or $\cos t$, we define (see Fig. 90)

Wavelength: The distance between successive wave crests

Frequency: The number of crests per unit distance, that is, the reciprocal of the wavelength

Amplitude: The height of the crests

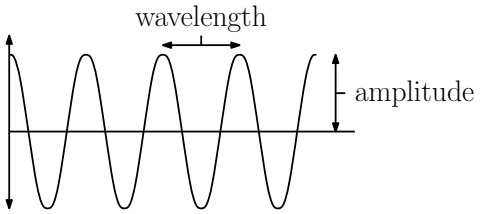


Fig. 90: Properties of periodic functions.

If we want to decrease the wavelength (equivalently increase the frequency) we can scale up the argument. For example $\sin t$ has a wavelength of 2π, $\sin(2t)$ has a wavelength of π, and $\sin(4t)$ has a wavelength of $\pi/2$. (By increasing the value of the argument we are increasing the function's frequency, which decreases the wavelength.) To decrease the function's amplitude, we apply a scale factor that is smaller than 1 to the value of the function. Thus, for any positive reals ω and α, the function $\alpha \cdot \sin(\omega t)$ has a wavelength of $2\pi/\omega$ and an amplitude of α.

Now, let's consider doing this to our noise function. Let $f(x)$ be the noise function as defined in the previous section. Let us assume that $0 \le x \le n$ and that the function repeats so that $f(0) = f(n)$ and let us assume further that the derivatives match at $x = 0$ and $x = n$. We can convert f into a periodic function for all $t \in \mathbb{R}$, which we call noise(t), by defining

$$\text{noise}(t) \;=\; f(t \bmod n).$$

(Again we are using the mod function in the context of real numbers. Formally, we define $x \bmod n = x - n \cdot \lfloor x/n \rfloor$.) For example, the top graph of Fig. 91 shows three wavelengths of noise(t).

In order to achieve self-similarity, we will sum together this noise function, but using different frequencies and with different amplitudes. First, we will consider the noise function with exponentially increasing frequencies: noise(t), noise$(2t)$, noise$(4t)$, ..., noise$(2^i t)$ (see Fig. 92). Note that we have not changed the underlying function, we have merely modified its frequency. In the jargon of Perlin noise, these are called *octaves*, because like musical octaves, the frequency doubles.[13] Because frequencies double with each octave, you do not need very many octaves, because there is nothing to be gained by considering wavelengths that are larger than the entire screen nor smaller than a single pixel. Thus, the logarithm of the window size is a natural upper bound on the number of octaves.

[13] In general, it is possible to use factors other than 2. Such a factor is called the *lacunarity* of the Perlin noise function. For example, a lacunarity value of ℓ means that the frequency at stage i will be ℓ^i.

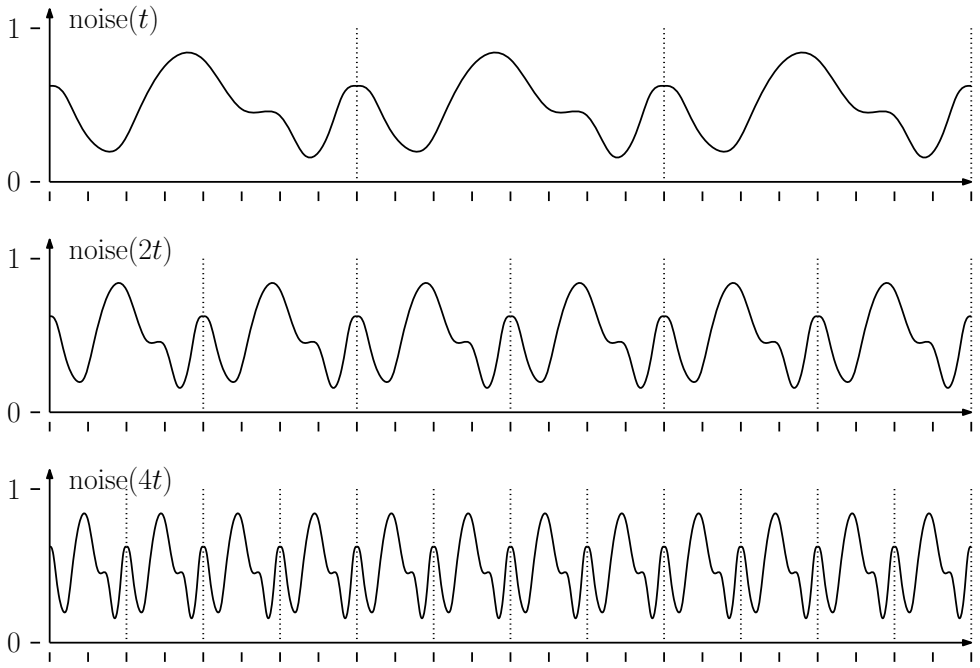


Fig. 91: The periodic noise function at various frequencies.

High frequency noise tends to be of lower amplitude. If we were in a purely self-similar situation, when the double the frequency, we should halve the amplitude. In order to provide the designer with more control, Perlin noise allows the designer to specify a separate amplitude for each frequency. A common way in which to do this is to define a parameter, called *persistence*, that specifies how rapidly the amplitudes decrease. Persistence is a number between 0 and 1. The larger the persistence value, the more noticeable are the higher frequency components. (That is, the more "jagged" the noise appears.) In particular, given a persistence of p, we define the amplitude at the ith stage to be p^i. The final noise value is the sum, over all the octaves, of the persistence-scaled noise functions. In summary, we have

$$\text{perlin}(t) \;=\; \sum_{i=0}^{k} p^i \cdot \text{noise}(2^i \cdot t),$$

where k is the highest-frequency octave.

It is possible to achieve greater control over the process by allowing the user to modify the octave scaling values (currently 2^i) and the persistence values (currently p^i).

Lecture 19: Procedural Generation: 2D Perlin Noise

Perlin Noise in 2D: In the previous lecture we introduced the concept of Perlin noise, a structured random function, in a one-dimensional setting. In this lecture we show how to generalize this concept to a two-dimensional setting. Such two-dimensional noise functions can be used for generating pseudo-random terrains and two-dimensional pseudo-random textures.

The general approach is the same as in the one-dimensional case:

- Generate a finite sample of random values

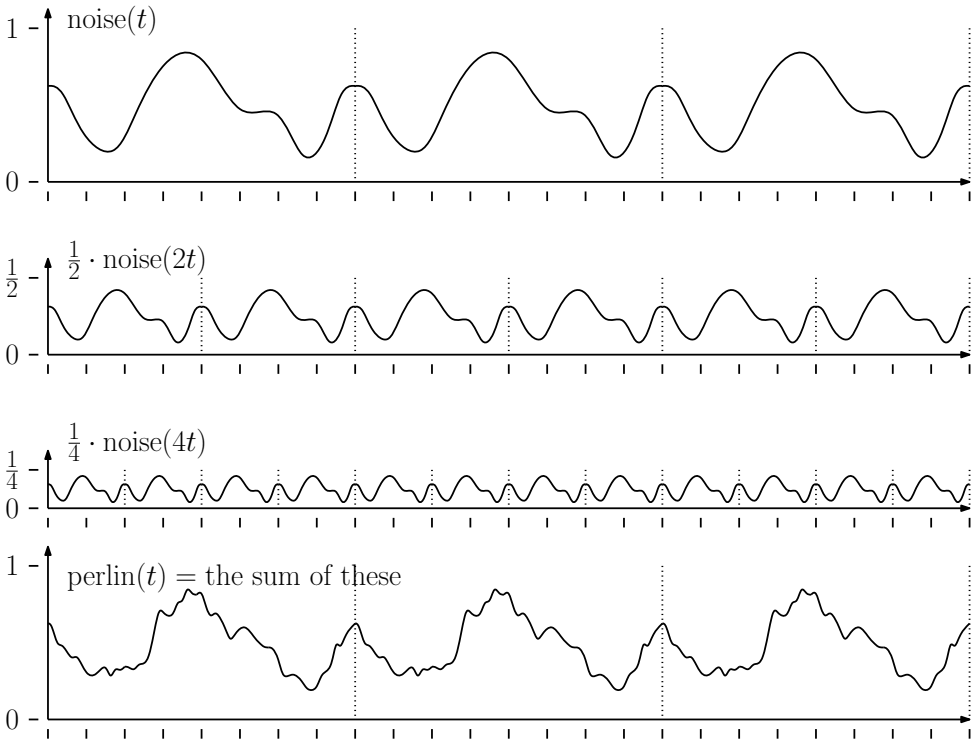


Fig. 92: Dampened noise functions and the Perlin noise function (with persistence $p = 1/2$).

- Generate a noise function that interpolates smoothly between these values
- Sum together various octaves of this function by scaling it down by factors of 1/2, and then applying a dampening persistence value to each successive octave, so that high frequency variations are diminished

Let's investigate the finer points. First, let us begin by assuming that we have an $n \times n$ grid of unit squares (see Fig. 93(a)), for a relatively small number n (e.g., n might range from 2 to 10). For each vertex $[i, j]$ of this grid, where $0 \leq i, j \leq n$, let us generate a random scalar value $z_{[i,j]}$. (Note that these values are actually not very important. In Perlin's implementation of the noise function, these value are all set to 0, and it still produces a remarkably rich looking noise function.) As in the 1-dimensional case, it is convenient to have the values wrap around, which we can achieve by setting $z_{[i,n]} = z_{[i,0]}$ and $z_{[n,j]} = z_{[0,j]}$ for all i and j.

Given any point (x, y), where $0 \leq x, y < n$, the corner points of the square containing this point are (x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1) where:

$$\begin{aligned} x_0 &= \lfloor x \rfloor \quad \text{and} \quad x_1 = (x_0 + 1) \bmod n \\ y_0 &= \lfloor y \rfloor \quad \text{and} \quad y_1 = (y_0 + 1) \bmod n \end{aligned}$$

(see Fig. 93(a)).

We could simply apply smoothing to the random values at the grid points, but this would produce a result that clearly had a rectangular blocky look (since every square would suffer the same variation). Instead, Perlin came up with a way to have every vertex behave differently, by creating a random gradient at each vertex of the grid.

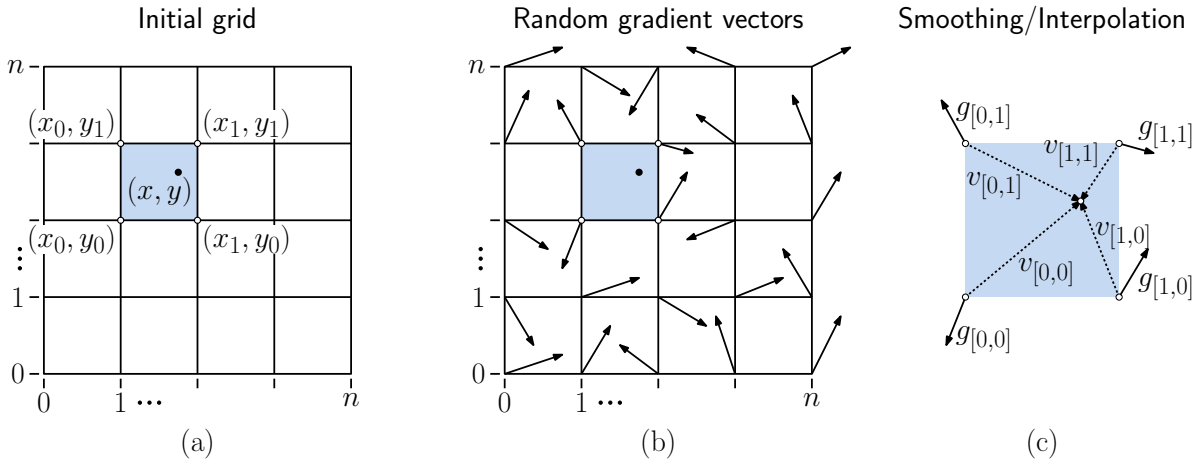


Fig. 93: Generating 2-dimensional Perlin noise.

Noise from Random Gradients: Before explaining the concept of gradients, let's recall some basics from differential calculus. Given a continuous function $f(x)$ of a single variable x, we know that the derivative of the function df/dx yields the tangent slope at the point $(x, f(x))$ on the function. If we instead consider a function $f(x, y)$ of two variables, we can visualize the function values $(x, y, f(x, y))$ as defining the height of a point on a two-dimensional terrain. If f is smooth, then each point of the terrain can be associated with tangent plane. The "slope" of the tangent plane passing through such a point is defined by the partial derivatives of the function, namely $\partial f/\partial x$ and $\partial f/\partial y$. The vector $(\partial f/\partial x, \partial f/\partial y)$ is a vector in the (x, y)-plane that points in the direction of *steepest ascent* for the function f. This vector changes from point to point, depending on f. It is called the *gradient* of f, and is often denoted ∇f.

Perlin's approach to producing a noisy 2-dimensional terrain involves computing a random 2-dimensional gradient vector at each vertex of the grid with the eventual aim that the smoothed noise function have this gradient value. Since these vectors are random, the resulting noisy terrain will appear to behave very differently from one vertex of the grid to the next. At one vertex the terrain may be sloping up to the northeast, and at a neighboring vertex it may be slopping to south-southwest. The random variations in slope result in a very complex terrain. But how do we define a smooth function that has this behavior? In the one dimensional case we used cosing interpolation. Let's consider how to generalize this to a two-dimensional setting.

Consider a single square of the grid, with corners (x_0, y_0), (x_1, y_0), (x_1, y_1), (x_0, y_1). Let $g_{[0,0]}$, $g_{[1,0]}$, $g_{[1,1]}$, and $g_{[0,1]}$ denote the corresponding randomly generated 2-dimensional gradient vectors (see Fig. 93(c)). Now, for each point (x, y) in the interior of this grid square, we need to blend the effects of the gradients at the corners. To do this, for each corner we will compute a vector from the corner to the point (x, y). In particular, define

$$\begin{aligned} v_{[0,0]} &= (x, y) - (x_0, y_0) \quad \text{and} \quad v_{[0,1]} = (x, y) - (x_0, y_1) \\ v_{[1,0]} &= (x, y) - (x_1, y_0) \quad \text{and} \quad v_{[1,1]} = (x, y) - (x_1, y_1) \end{aligned}$$

(see Fig. 93(c)).

Next, for each corner point of the square, we generate an associated *vertical displacement*, which indicates the height of the point (x, y) due to the effect of the gradient at this corner point. How should this displacement be defined? Let's fix a corner, say $(x, 0, y_0)$. Intuitively, if $v_{[0,0]}$ is directed in the same direction as the gradient vector, then the vertical displacement will increase (since we are going uphill). If it is in the opposite direction, the displacement will decrease (since we are going downhill). If the two vectors are orthogonal, then the vector $v_{[0,0]}$ is directed neither up- or downhill,

and so the displacement is zero. Among the vector operations we have studied, the *dot product* produces exactly this sort of behavior. (When two vectors are aligned, the dot-product is maximized, when they are anti-aligned it is minimized, and it is zero when they are orthogonal. It also scales linearly with the length, so that a point that is twice as far away along a given direction has twice the displacement.) With this in mind, let us define the following scalar displacement values:

$$\begin{aligned} \delta_{[0,0]} &= (v_{[0,0]} \cdot g_{[0,0]}) \quad \text{and} \quad \delta_{[0,1]} = (v_{[0,1]} \cdot g_{[0,1]}) \\ \delta_{[1,0]} &= (v_{[1,0]} \cdot g_{[1,0]}) \quad \text{and} \quad \delta_{[1,1]} = (v_{[1,1]} \cdot g_{[1,1]}). \end{aligned}$$

Fading: The problem with these scalar displacement values is that they are affected by all the corners of the square, and in fact, as we get farther from the associated corner point the displacement gets larger. We want the gradient effect to apply close to the vertex, and then have it drop off quickly as we get closer to another vertex. That is, we want the gradient effect of this vertex to *fade* as we get farther from the vertex. To do this, Perlin defines the following *fade function*. This is a function of t that will start at 0 when $t = 0$ (no fading) and will approach 1 when $t = 1$ (full fading). Perlin originally settled on a cubic function to do this, $\varphi(t) = 3t^2 - 2t^3$. (Notice that this has the desired properties, and further its derivative is zero at $t = 0$ and $t = 1$, so it will smoothly interpolate with neighboring squares.) Later, Perlin observed that this function has nonzero second derivatives at 0 and 1, and so he settled on the following improved fade function:

$$\psi(t) = 6t^5 - 15t^4 + 10t^3$$

(see Fig. 94). Observe again that $\psi(0) = 0$ and $\psi(1) = 1$, and the first and second derivatives are both zero at these endpoints.

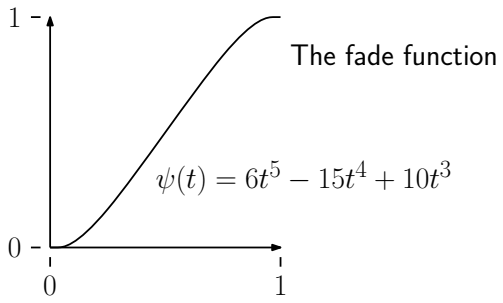


Fig. 94: The fade function.

Because we want the effects to fade as a function of both x and y, we define the *joint fade function* to be the product of the fade functions along x and y:

$$\Psi(x, y) = \psi(x)\psi(y).$$

The final noise value at the point (x, y), arises by taking the weighted average of gradient displacements, where each displacement is weighted according to the fade function.

We need to apply the joint fade function differently for each vertex. For example, consider the fading for the displacement $\delta_{[1,0]}$ of the lower right corner vertex. We want the influence of this vertex to increase as x approaches 1, which will be achieved by using a weight of $\psi(x)$. Similarly, we want the influence of this vertex to increase as y approaches 0, which will be achieved by using a weight of $\psi(1 - y)$. Therefore, to achieve both of these effects, we will use the joint weight function $\Psi(x, 1 - y)$. By applying this reasoning to the other corner vertices, we obtain the following 2-dimensional noise function.

$$\text{noise}(x, y) = \Psi(1 - x, 1 - y)\delta_{[0,0]} + \Psi(x, 1 - y)\delta_{[1,0]} + \Psi(1 - x, y)\delta_{[0,1]} + \Psi(x, y)\delta_{[1,1]}.$$

Adding Back the Random Heights: We have left one little bit out of our noise function. Remember that we started off by assigning random scalar values to each of the of the grid. We never made use of these (and indeed, Perlin's formulation of the noise function does not either). In order to achieve this extra degree of randomness, we can add these back into the vertical displacements. Suppose, for example that we are considering the grid square whose lower left corner has the indices $[i, j]$. When defining the vertical displacements, let us add in the associated random scalar values associated with each of the vertices:

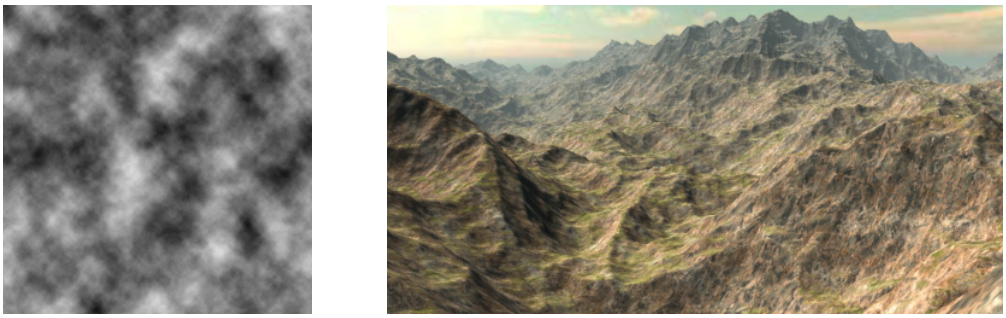
$$\begin{aligned} \delta_{[0,0]} &= z_{[i,j]} + (v_{[0,0]} \cdot g_{[0,0]}) \quad \text{and} \quad \delta_{[0,1]} = z_{[i,j+1]} + (v_{[0,1]} \cdot g_{[0,1]}) \\ \delta_{[1,0]} &= z_{[i+1,j]} + (v_{[1,0]} \cdot g_{[1,0]}) \quad \text{and} \quad \delta_{[1,1]} = z_{[i+1,j+1]} + (v_{[1,1]} \cdot g_{[1,1]}). \end{aligned}$$

The rest of the noise computation is exactly the same as described above.

Octaves and Persistence: After all of that work, we still have only a single smooth noise function, but not one that demonstrates the sort of fractal-like properties we desire. To add this, we need to perform the same scaling that we used for the 1-dimensional case. In this case, the process is entirely analogous. As before, let p be a value between 0 and 1, which will determine how quickly things dampen down. Also, as before, at each level of the process, we will double the frequency. This leads to the following final definition of the 2-dimensional Perlin noise:

$$\text{perlin}(x, y) = \sum_{i=0}^{k} p^i \cdot \text{noise}(2^i \cdot x, 2^i \cdot y).$$

(As before, recall that the value 2^i can be replaced by some parameter ℓ^i, where $\ell > 1$.) This applies to each square individually. We need to perform the usual "modding" to generalize this to any square of the grid. (An example of the final result is shown in Fig. 95(a) and a terrain resulting from applying this is shown in Fig. 95(b). Note that the terrain does not look as realistic as the *terragen* from Fig. 93(a). There are other processes, such as erosion and geological forces that need to be modeled to achieve highly realistic terrains.)



(a) (b)

Fig. 95: (a) Two-dimensional Perlin noise and (b) a terrain generated by Perlin noise.

Source Code: While the mathematical concepts that we have discussed are quite involved, it is remarkable that Perlin noise has a very simple implementation. The entire implementation can be obtained from Perlin's web page, and is shown nearly in its entirety in the code block below.

Lecture 20: Procedural Generation: L-Systems

L-Systems: In this lecture we will consider the issue of how to generate "tree-like" objects, which are characterized by a process of growth and branching. The standard approach is through a structure

```
public final class ImprovedNoise {
   static public double noise(double x, double y, double z) {
      int X = (int)Math.floor(x) & 255,                  // FIND UNIT CUBE THAT
          Y = (int)Math.floor(y) & 255,                  // CONTAINS POINT.
          Z = (int)Math.floor(z) & 255;
      x -= Math.floor(x);                                // FIND RELATIVE X,Y,Z
      y -= Math.floor(y);                                // OF POINT IN CUBE.
      z -= Math.floor(z);
      double u = fade(x),                                // COMPUTE FADE CURVES
             v = fade(y),                                // FOR EACH OF X,Y,Z.
             w = fade(z);
      int A = p[X  ]+Y, AA = p[A]+Z, AB = p[A+1]+Z,// HASH COORDINATES OF
          B = p[X+1]+Y, BA = p[B]+Z, BB = p[B+1]+Z;// THE 8 CUBE CORNERS,
      // AND ADD BLENDED RESULTS FROM 8 CORNERS OF CUBE
      return lerp(w, lerp(v, lerp(u, grad(p[AA  ], x  , y  , z   ),
                                     grad(p[BA  ], x-1, y  , z   )),
                             lerp(u, grad(p[AB  ], x  , y-1, z   ),
                                     grad(p[BB  ], x-1, y-1, z   ))),
                     lerp(v, lerp(u, grad(p[AA+1], x  , y  , z-1 ),
                                     grad(p[BA+1], x-1, y  , z-1 )),
                             lerp(u, grad(p[AB+1], x  , y-1, z-1 ),
                                     grad(p[BB+1], x-1, y-1, z-1 ))));
   }
   static double fade(double t) { return t*t*t*(t*(t*6 - 15) + 10); }
   static double lerp(double t, double a, double b)
      { return a + t*(b - a); }
   static double grad(int hash, double x, double y, double z) {
      int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE
      double u = h<8 ? x : y,                 // INTO 12 GRADIENT DIRECTIONS.
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }
   static final int p[] = new int[512], permutation[] = {
      151,160,137,91,90,15,  // ... remaining 506 entries omitted
   };
   static { for (int i=0; i < 256 ; i++)
      p[256+i] = p[i] = permutation[i]; }
}
```

called an *L-system*. L-systems, short for *Lindenmayer-systems*, were first proposed by a biologist Aristid Lindenmayer in 1968, as a mechanism for defining plant development.

Fig. 96: Examples of simple plant models generated by L-systems.

If you have taken a course in formal language theory, the concept of an L-system is very similar to the concept of a context-free grammar. We start with an alphabet, which is a finite set of characters, called *symbols* or *variables*. There is one special symbol, called the *start symbol* (or *axiom* in L-system terminology). In L-systems, symbols are categorized in two types. First, *variables* are symbols that can be replaced with other symbols. Second, *constants* are symbols that are fixed and cannot be replaced. Finally, there is a finite set of *production rules*. Each production replaces a single variable with a string (or zero or more) symbols (which may be variables or constants). Such a rule is expressed in the following form:

$$\langle \text{variable} \rangle \ \rightarrow \ \langle \text{string} \rangle.$$

To get a better grasp on this, let us consider a simple example, developed by Lindenmayer himself to describe the growth of algae.

$$\begin{aligned} \text{variables} &: \{\mathsf{A}, \mathsf{B}\} \\ \text{constants} &: \emptyset \text{ (none)} \\ \text{start} &: \mathsf{A} \\ \text{rules} &: \mathsf{A} \rightarrow \mathsf{AB}; \quad \mathsf{B} \rightarrow \mathsf{A} \end{aligned}$$

An L-system works as follows. Starting with the start symbol, we repeatedly replace each variable with a compatible rule. In this case, each occurrence of A is mapped to AB and each occurrence of B is mapped to A. This is repeated for some number of levels. (Note that this is major difference between L-systems and context-free grammars. In context-free grammars, any one rule is applied. In L-systems, *all* the applicable rules are applied in parallel. The above grammar produces the following sequence of strings (for the first 7 levels of application):

$$\begin{aligned} n = 0 &: \mathsf{A} \\ n = 1 &: \mathsf{AB} \\ n = 2 &: \mathsf{ABA} \\ n = 3 &: \mathsf{ABAAB} \\ n = 4 &: \mathsf{ABAABABA} \\ n = 5 &: \mathsf{ABAABABAABAAB} \\ n = 6 &: \mathsf{ABAABABAABAABABAABABA} \\ n = 7 &: \mathsf{ABAABABAABAABABAABABAABAABABAABAAB} \end{aligned}$$

As an aside, if you count the number of symbols in each of the strings, you'll observe the sequence $\langle 1, 2, 3, 5, 8, 13, 21, \ldots \rangle$. Is this sequence familiar? This is the famous Fibonacci sequence, which has been observed to arise in the growth patterns of many organisms.

Using L-Systems to Generate Shapes: So what does this have to do with shape generation? The idea is to let each symbol represent some sort of drawing command (e.g., draw a line segment, turn at a specified angle, etc.) This is sometimes referred to as *turtle geometry* (I guess this is because each command is thought of as being relayed to a turtle who carries out the drawing process).

Consider the following L-system:

$$\begin{aligned} \text{variables} &: \{0, 1\} \\ \text{constants} &: \{[,]\} \\ \text{start} &: 0 \\ \text{rules} &: 1 \to 11; \quad 0 \to 1[0]0 \end{aligned}$$

If we carry out the first few stages of the expansion, we have

$$\begin{aligned} n = 0 &: 0 \\ n = 1 &: 1[0]0 \\ n = 2 &: 11[1[0]0]1[0]0 \\ n = 3 &: 1111[11[1[0]0]1[0]0]11[1[0]0]1[0]0 \end{aligned}$$

There is nothing particularly pictorial about this, but now let's assign some drawing instructions. Let us assume that we associate a push-down stack with the drawing process. Define

- "0": Draw a unit length line segment with a leaf on the end and increase
- "1": Draw a unit length line segment
- "[": push the current position/scale and angle on the stack turn CCW 45°, and scale by 1/2
- "]": pop the current position/scale and angle off the stack, turn CW 45°, and scale by 1/2

(An L-system that uses [and] is sometimes referred to as an *L-system with brackets*.)

Now, if we use the above directions as the basis for generating a turtle-geometry drawing, we obtain the drawing shown in Fig. 97.

Of course, this is far from being a realistic tree, but it is not hard to enhance this basic system with more parameters and some randomness in order to generate something that looks quite like a tree.

Randomization and Stochastic L-Systems: As described, L-systems generate objects that are too regular to model natural objects. However, it is an easy matter to add randomization to the process.

The first way of introducing randomness is to randomize the graphical/geometric operations. For example, rather than mapping terminal symbols into fixed actions (e.g., draw a unit-length line segment), we could add some variation (e.g., draw a line segment whose length is a random value between 0.9 and 1.1). Examples include variations in drawing lengths, variations in branching angles, and variations in thickness and/or texture (see Fig. 96.)

While the above modifications alter the geometric properties of the generated objects, the underlying structure is still the same. We can modify L-systems to generate random structures by associating each production rule with a probability, and apply the rules randomly according to these probabilities. For example consider the following two rules:

$$\begin{aligned} \mathsf{a} &\longrightarrow_{[0.4]} \mathsf{a[b]} \\ \mathsf{a} &\longrightarrow_{[0.6]} \mathsf{b[a]b} \end{aligned}$$

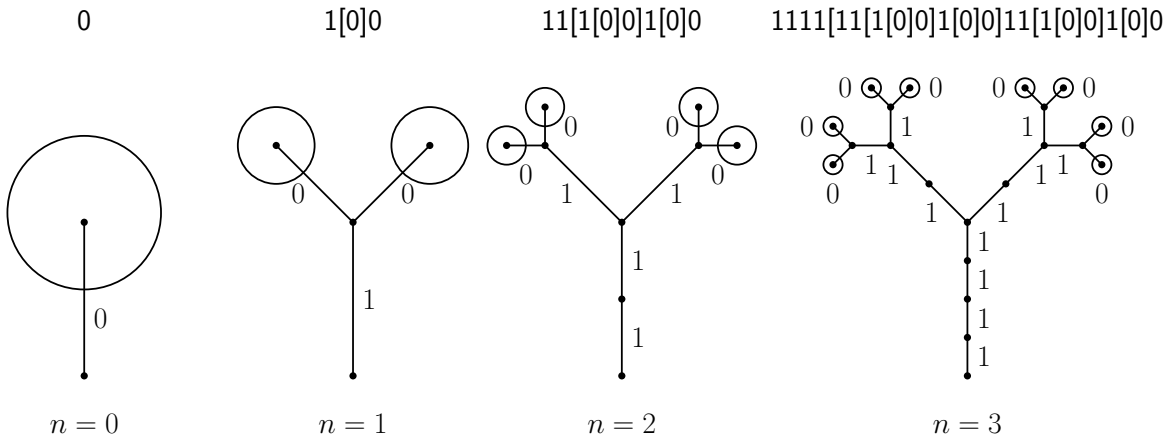


Fig. 97: A simple tree-like structure generated by an L-system. (Note that the figure is not drawn exactly to scale.) With each branching, the scale factor decreases by 1/2.

The interpretation is that the first rule is to by applied 40% of the time and the second rule 60% of the time.

Lecture 21: Motion Planning: Crowds and Pursuit/Evasion

Overview: Last time, we discussed multi-agent motion in the form of particle systems and models of animal flocking. Today, we will focus on motion planning involving multiple agents, but where the agents are assumed to be considerably more complex, either from the perspective of having intelligent problem-solving skills or to alter behavior based on the perceived intentions of other agents in the system.

Pursuit Evasion: There are many motion-planning problems where it is assumed that intelligent agents are involved. For example, one can imagine the strategies employed by a pack of wolves that are attacking a herd of elk. Both sets of agents (wolves and elk) have their own individual aims (kill or avoid being killed), and as a group they can attempt to coordinate their behaviors in order to achieve these aims. These strategies are generally quite complicated and difficult to describe formally. To make the task simpler, mathematicians have posed extremely simple special cases, called *pursuit-evasion games*, in order to better understand these strategies.

In pursuit-evasion games, the agents are divided into two groups. The pursuers attempt to track down and locate the evaders, while the latter attempt to avoid being caught by a pursuer. These games may be played in a discrete setting (e.g., where agents are located on a chess board and move one square at a time) or continuous setting (e.g., where agents are points on the real plane, and may move along any trajectory they like). There are a number of famous examples of such games whose solutions are quite complex. They come with colorful names, like the *homocidal chauffeur game* (a high-speed car driver with a limited turning radius is trying to run over a slow but highly maneuverable pedestrian) and the *princess and monster game* (a slow-moving monster in a cave is trying to catch a fast moving princess, but both operate in total darkness). Let's look at a couple of well-studied pursuit-evasion problems in greater detail.

Visibility-based Pursuit Evasion We are given a continuous domain (e.g., a simple polygon) in the plane. Two moving points, the pursuer p and the evader e, can move in time but must stay within the domain. Let $p(t)$ and $e(t)$ denote their positions at some time $t \geq 0$. The pursuer has *caught* the evader if at any time $t_1 > 0$, $p(t_1)$ can see $e(t_1)$, meaning that the line segment $\overline{p(t)e(t)}$ lies

entirely within the interior of the domain. We assume that the evader knows where the pursuer is at times and can move at arbitrarily high speeds. The question is whether it is possible to plan a path for the pursuer such that the evader cannot escape being caught eventually. (Note that the notion of being "caught" really just means being "seen.")

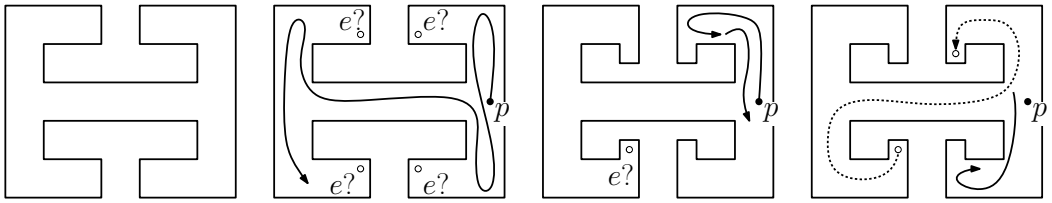


Fig. 98: Visibility-based pursuit evasion. In the case of polygon (a), the pursuer has a winning strategy, but in (c) the evader can always evade the pursuer.

For example, consider the domain shown in Fig. 98. For the domain shown in (a), the pursuer wins the game by following the path shown in (b). No matter where the evader attempts to hide, he will eventually be seen and further he cannot sneak from one hiding place to another without being seen by the pursuer. However if we add four small nobs at the end of each of the four bays, the evader now wins the game (assuming he knows the pursuer's strategy in advance).

Seeing why the evader can elude detection involves an analysis of the various cases of the sequence of bays visited by the pursuer. For example, suppose that the pursuer vists the northeast (NE) bay first, then the southeast (SE), then the northwest (NW), and then comes back to the SE bay, and suppose that the evader starts in the NW bay. The evader could reason as follows. After the pursuer leaves NE and moves down to look into SE, the evader zips from NW to NE. When traverses the central horizontal corridor and starts moving up to the NW bay, the evader is free to move up into the NE bay. Finally, when the pursuer returns to SE, the evader has left it. After a bit of thinking, you should be able to convince yourself that by looking ahead to the pursuer's next move, the evader can always identify a bay in which to hide now and escape from later.

To see whether you understand this, consider the four domains shown in Fig. 99. For which of these does the pursuer have a winning strategy, and for which does the evader?

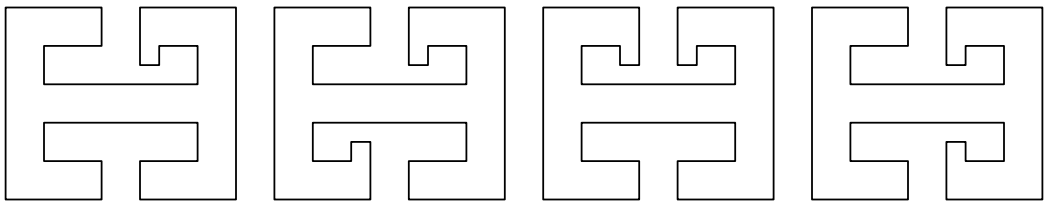


Fig. 99: Visibility-based pursuit evasion. In which of these does the pursuer win? (Hint: The evader wins in only one of them.)

Given a simple polygon, it is possible to determine whether there exists a pursuit path (that is, a winning solution for the pursuer). However, the algorithm is quite complex. It runs in time $O(n^2)$, where n is the number of vertices in the polygon. The success of the evader depends on its complete knowledge of the pursuer's path. If the pursuer is allowed to use randomization (coin

flips) to determine its moves, then it can be shown that the pursuer will eventually get lucky and find the evader, no matter how clever the evader is.

Lion and Man: This game is played in the first quadrant of the real plane ($x \geq 0$ and $y \geq 0$). There is a lion that starts at a point $L = (L_x, L_x)$ and a man at point $M = (M_x, M_y)$. At each step, the man may take one move of length up to 1 unit, and then the lion may take one move of length up to 1 unit, and this repeats (see Fig. 100(a)). Assuming both players play in the best way possible, either the lion will eventually catch the man (when the two points coincide) or the man can evade the lion forever. Based on the starting configuration, which is it?

There are a few cases that are easy to see. First off, if the man is either north ($M_y \geq L_y$) or east ($M_x \geq L_x$) then the man can escape being caught forever. But what if $M_x < L_x$ and $M_y < L_y$? It turns out that the man will eventually be caught. However, it is not easy to see exactly why.

Here is a winning strategy for the tiger for the above case. First, observe that M lies below and to the left of L, then there exists a point C with the following properties: (1) L lies along the line segment $\overline{CM}$, (2) a circle centered at C and passing through L intersects both the positive x- and positive y-axes (Fig. 100(b)).

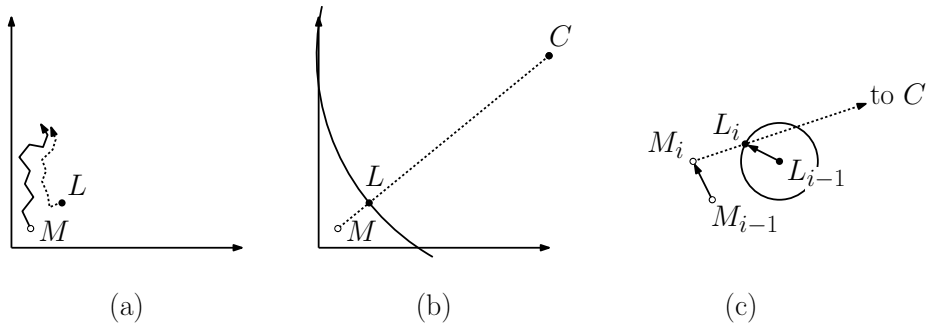


Fig. 100: Lion and Man Problem

Crowd Motion: Today, we will discuss motion simulation involving a number of intelligent autonomous agents, as arises in crowds of pedestrians. Unlike flocking systems, in which it is assumed that the agents behave homogeneously, in a crowd it is assumed that each agent has its own agenda to pursue (see Fig. 101). For example, we might imagine a group of students walking through a crowded campus on their way to their next classes. Since such agents are acting within a social system, however, it is assumed that they will tend to behave in a manner that is consistent with social conventions. (I don't want you to bump into me, and so I will act in a manner to avoid bumping into you as well.)

Crowd simulation is actually a very broad area of study, ranging from work in game programming and computer graphics, artificial intelligence, social psychology, and urban architecture (e.g., planning evacuation routes). In order to operate in the context of a computer game, such a system needs to be decentralized, where each member of the crowd determines its own action based on the perceived actions of other nearby agents. The problem with applying a simple boid-like flocking behavior is that, whereas flocking rules such as alignment naturally produce systems that avoid collisions between agents, the diverse agendas of agents in crowds naturally brings them directly into collisions with other agents (as in pedestrians traversing a crosswalk from both sides). In order to produce more realistic motion, the agents should anticipate where nearby agents are moving, and then plan their future motion accordingly. We discuss two models of crowd behavior, social-force dynamics and (reciprocal) velocity obstacles.

Social-Force Dynamics: In our presentation of flocking behavior with Boids, we presented a model in which the motion of each artificial animal is determined by a collection of simple local forces based on

Fig. 101: Crowd simulation.

the other agents in the system. In much the same manner that a physicist would simulate the motion of a collection of particles in computational fluid dynamics, we can simulate the fluid-like motion of animals in the flock. This can be applied to human crowd behavior as well. At an individual level, human behavior is quite chaotic, and it is not easy to predict future motion based on past motion. However, for many common situations the aggregate behavior of a large group of people can be quite predictable. Examples include people walking along corridors or sidewalks, moving into or evacuating from a building, milling around in a large area (such as people at a party or on the floor of a convention). Social-force dynamics attempts to model the motion of large crowds of humans in terms of simple local forces that incrementally affect their individual motions.

Recalling our earlier lecture on *agents* in AI, each person in a crowd experiences sensory stimuli, whic cause a behavioral reaction that depends on the agent's personal aims and is chosen from a set of behavioral alternatives. The choice depends on a utility maximization, which varies by individual. While each individual may have a (highly unpredictable) utility function, the utility functions of a large group can be modeled by a probability distribution. (E.g., 10% of people are just focused on getting as quickly as possible to their destination, 20% are in no hurry or are strolling together with a friend, and 70% are staring at their mobile devices and they are not paying attention to anything else.)

The physics-based model associates with each agent i and each time instant t a *current position*, denoted $p_i(t)$, a *current velocity* vector, denoted $\vec{v}_i(t)$. Based on the agent's desired path (which resulted from some path planning procedure) each agent has a *target velocity* vector, denoted $\vec{v}_i^{\,0}(t)$. As described below, various forces will be evaluated at each time step. Each force will be represented as a vector, and the sum of these forces will result in an *aggregate force* vector, denoted $F_i(t)$. (For example, the aggregate force will tend to push the agent away from other agents in the crowd and obstacles in the environment, and will also nudge it back towards the target velocity.)

At each time step Δ, we modify the current velocity vector based on the aggregate force vector:

$$\vec{v}_i(t+\Delta) \;=\; \vec{v}_i(t) + \Delta \cdot F_i(t).$$

We then advance the position of the agent accordingly

$$p_i(t+\Delta) \;=\; p_i(t) + \Delta \cdot \vec{v}_i(t+\Delta)$$

Of course, after this movement new forces will come into existence, and the process repeats.

Now that we know how to update an agent, the next question is how to compute the forces.

Avoidance: The agent would like to avoid collisions with other agents in the crowd. We can model this as a *repulsive force* vector $f_{ji}(t)$ that is directed from a nearby agent j toward agent i. This force would presumably be the greatest for other agents that are within a proximity sphere about agent i. Ideally, this force should take into consideration the velocity of the other agent j. (In the discussion below on reciprocal velocity obstacles, it is discussed how to compute such a force.)

Another form of avoidance is a repulsive force to keep the agent from colliding with obstacles in the scene. (In Boids, we discussed predators. The analog here might be pedestrians avoiding motor vehicles, which depending on the part of the world you are visiting, might seem to behave like predators!)

Attraction: An agent that is walking with a group (e.g., a single companion, within a small group of friends, or as part of a tour group) will want to maintain proximity with other agents. This can be modeled as a force $f_{ig}(t)$ that attracts agent i to the perceived center of group g to which it belongs (or perhaps to the leader of the group in the case of a tour group).

Traffic signals and social conventions: In urban settings, signals like walk-lights and cross-walks naturally induce forces that encourages or discourages movement across roadways. In certain regions, there is a tendency to walk on the right side or left side of a corridor or sidewalk, presumably drawn from the conventions for driving in the region.

Velocity Obstacles: [The material that follows is optional.]

Another approach to planning the motion of individuals in crowds is based on the observation that each agent should consider its own current velocity in conjunction with its estimates of the velocities of agents lying in its immediate vicinity. Based on relative speeds and directions, each agent can determine a set of forbidden velocities (represented as a region of some velocity space) that will lead to an imminent collision. The agent then selects a velocity that is close to its desired velocity, but lying outside of the region of forbidden velocities. A robust method for doing this is based on the notion of *velocity obstacles*.

Suppose that an agent a is moving in a crowded environment where many other agents are also moving. We assume that a can perceive the motions of nearby agents and generate rough estimates on their velocities. Agent a is also interested in traveling to some destination, and (based on our path finding algorithm) we assume that a has a *preferred velocity*, v_a^*, which it would assume if there were no other agents to consider.

For the sake of simplicity, we will model agents as circular disks translating in the plane. Consider two such agents, a and b, of radii r_a and r_b and currently positioned at points p_a and p_b, respectively (see Fig. 102(a)). If we assume that agent a moves at velocity v_a, then at time t it will have moved a distance of $t \cdot v_a$ from its initial position, implying that it will be located at $p_a + t \cdot v_a$. We will refer to this as $p_a(t)$.

For now, let's assume that object b is not moving, that is, $p_b(t) = p_b$ for all t. Let's consider the question of how to select a velocity for a that avoids hitting b any time in the future. Two disks intersect if the distance between their centers ever falls below the sum of their radii. More formally, let $\|u\|$ denote the Euclidean length of a vector u. Then $\|p_a(t) - p_b(t)\|$ is the distance between p_a and p_b at time t (that is, the length of the vector from b to a). Thus, the two agents collide if $\|p_a(t) - p_b(t)\| < r_a + r_b$. By substituting in the definitions of $p_a(t)$ and $p_b(t)$, we find that a velocity v_a will result in a collision some time in the future if there exists a $t > 0$, such that

$$\|(p_a + t \cdot v_a) - p_b\| \;<\; r_a + r_b. \tag{3}$$

We would like to express the velocities v_a that satisfy the above criterion as lying within a certain "forbidden region" of space. To do this, define $B(p, r)$ to be the open Euclidean ball of radius r centered at point p. That is

$$B(p, r) \;=\; \{q : \|q - p\| < r\}.$$

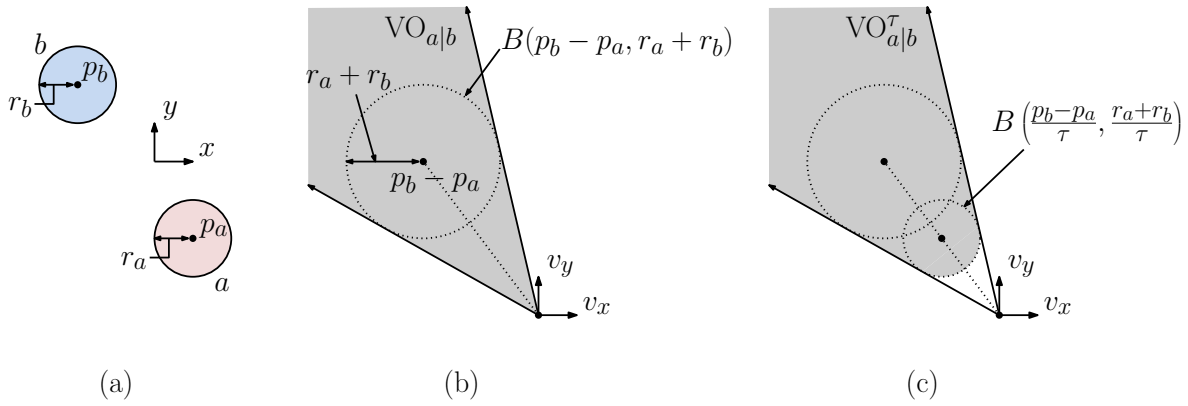


Fig. 102: Velocity obstacle for a induced by b (assuming b is stationary).

We can rewrite Eq. (3) as

$$\|t \cdot v_a - (p_b - p_a)\| \ < \ r_a + r_b,$$

which is equivalent to saying that $t \cdot v_a$'s distance from the point $p_b - p_a$ is less than $r_a + r_b$, or equivalently, the (parametrized) vector $t \cdot v_a$ lies within a ball of radius $r_a + r_b$ centered at the point $p_b - p_a$. Thus, we can rewrite Eq. (3) as

$$t \cdot v_a \in B(p_b - p_a, r_a + r_b),$$

As t varies from 0 to $+\infty$, the vector $t \cdot v_a$ travels along a ray that starts at the origin and travels in the direction of v_a. Therefore, the set of forbidden velocities are those that lie within a cone that is centered at the origin and encloses the ball $B(p_b - p_a, r_a + r_b)$.

We define the *velocity obstacle* of a induced by b, denoted $\mathrm{VO}_{a|b}$ to be the set of velocities of a that will result in a collision with the stationary object b.

$$\mathrm{VO}_{a|b} \ = \ \{v : \exists t \geq 0, \ t \cdot v \in B(p_b - p_a, r_a + r_b)\}.$$

(See Fig. 102(b).)

One problem with the velocity object is that it considers time going all the way to infinity. In a dynamic setting, we cannot predict the motion of objects very far into the future. So, it makes sense to truncate the velocity obstacle so that it only involves a limited time window. Given a time value $\tau > 0$, what the set of velocities that will result in agent a colliding with agent b at any time t, where $0 < t \leq \tau$ is the *truncated velocity obstacle*:

$$\mathrm{VO}^\tau_{a|b} \ = \ \{v : \exists t \in [0, \tau], \ t \cdot v \in B(p_b - p_a, r_a + r_b)\}.$$

This is a subset of the (unbounded) velocity obstacle that eliminates very small velocities (since collisions farther in the future result when a is moving more slowly). The truncated velocity obstacle is a truncated cone, where the truncation occurs at the boundary of the $(1/\tau)$-scaled ball $B((p_b - p_a)/\tau, (r_a + r_b)/\tau)$ (see Fig. 102(c)). Observe that there is an obvious symmetry here. Moving a with velocity v will result in a collision with (the stationary) b if and only if moving b with velocity $-v$ will result in an intersection with (the stationary) a. Therefore, we have

$$\mathrm{VO}^\tau_{a|b} \ = \ -\mathrm{VO}^\tau_{b|a}.$$

Collision-Avoiding Velocities: Next, let us consider how the velocity obstacle changes if b is moving. If we assume that b is moving with velocity v_b, then a velocity v_a will generate a collision precisely if the

differences in their velocities $v_a - v_b$ generates a collision under the assumption that b is stationary, that is, $v_a - v_b \in \mathrm{VO}^{\tau}_{a|b}$. Equivalently, v_a will generate a collision if b if v_a lies in the offset velocity obstacle $\mathrm{VO}^{\tau}_{a|b} + v_b$ (see Fig. 103(a)).

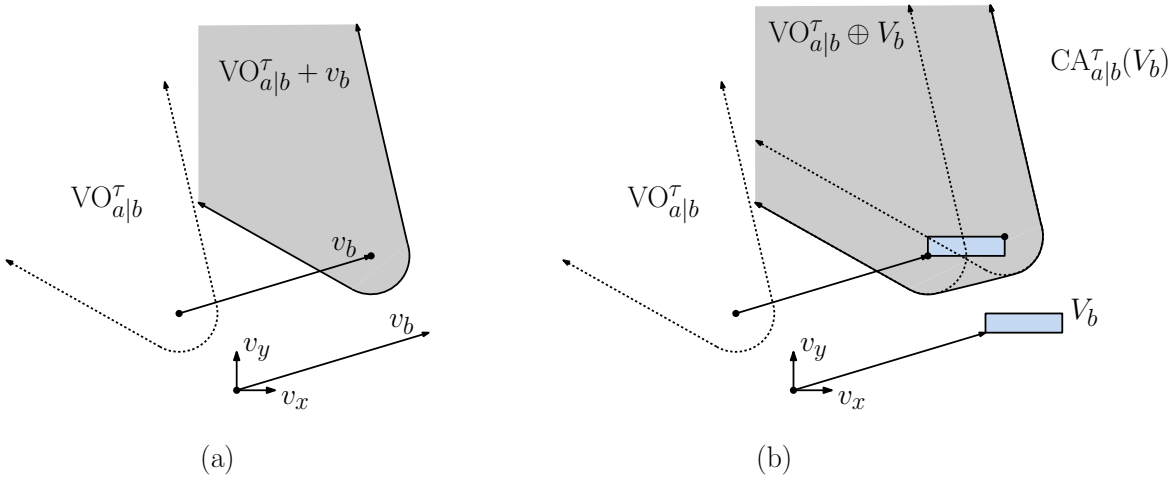


Fig. 103: Velocity obstacles where (a) object b is moving at velocity v_b and (b) object b is moving at any velocity in the set V_b.

We can further generalize this. We usually do not know another object's exact velocity, but we can often put bounds on it. Suppose that rather than knowing b's exact velocity, we know that b is moving at some velocity v_b that is selected from a region V_b of possible velocities. (For example, V_b might be square or circular region in space, based on the uncertainty in its motion estimate.)

Let us consider the velocities of a that *might* result in a collision, assuming that v_b is chosen from V_b. To define this set, we first define the *Minkowski sum* of two sets of vectors X and Y to be the set consisting of the pairwise sums of vectors from X and Y, that is,

$$X \oplus Y \;=\; \{x + y : x \in X \text{ and } y \in Y\}.$$

Then, clearly a might collide with b if a's velocity is chosen from $\mathrm{VO}^{\tau}_{a|b} \oplus V_b$. Therefore, if we want to avoid collisions with b altogether, then a should select a velocity from outside this region. More formally, we define the set of *collision-avoiding velocities* for a given than b selects a velocity from V_b is

$$\mathrm{CA}^{\tau}_{a|b}(V_b) \;=\; \{v : v \notin \mathrm{VO}^{\tau}_{a|b} \oplus V_b\}$$

(see Fig. 103(b).)

Just to recap, if a selects its velocity vector from anywhere outside $\mathrm{VO}^{\tau}_{a|b} \oplus V_b$ (that is, anywhere inside $\mathrm{CA}^{\tau}_{a|b}(V_b)$), then no matter what velocity b selects from V_b, a is guaranteed not to collide with b within the time interval $[0, \tau]$.

This now provides us with a strategy for selecting the velocities of the agents in our system:

- Compute velocity bounds V_b for all nearby agents
- Compute the intersection of all collision-avoiding velocities for these objects, that is

$$\mathrm{CA}^{\tau}_{a} = \bigcap_{b} \mathrm{CA}^{\tau}_{a|b}(V_b)$$

Any velocity chosen from this set is guaranteed to avoid collisions from now until time τ.

- Select the vector from CA_a^τ that is closest to a's ideal velocity v_a^*.

In practice, we need to take some care in the implementation of this last step. First, there will be upper limits on fast an object can move or change directions. So, we may not be free to select any velocity we like. Subject to whatever practical limitations we have on what the future velocity can be, we select the closest one that lies within CA_a^τ. If there is no such vector, then we must consider the possibility that we cannot avoid a collision. If so, we can select a vector that overlaps the smallest number of velocity obstacles.

Issues: While this might seem to be the end of the story with respect to velocity obstacles, there are still issues that arise. One of these issues was hinted at in the last paragraph. In cases where there are lots of agents and the velocity estimates are poor, the collision-avoiding area may be empty. There are a number of strategies that one might consider for selecting a good alternative.

There is a more significant problem with this approach, however, which arises even if we consider only two agents. The problem is that agents that are moving towards each other can engage in a very unnatural looking oscillating motion. The situation is illustrated in Fig. 104. Two agents are moving to each other. They see that they are on a collision course, and so the divert from their initial velocities. Let's imagine the best-case scenario, where they have successfully resolved their impending collision (whew!) as a result of this diversion (see Fig. 104(b)). Now, each agent sees that the other agent has diverted from its trajectory and reasons, "Great! The other guy has veered off, and I have won the game of chicken. So, now I can resume on my original velocity" (see Fig. 104(c)). So, both agents return to their original velocity, and they are now right back on a collision course (see Fig. 104(d)) and so again they divert (see Fig. 104(e)). Clearly, this vicious cycle of zig-zagging motion will repeat until they manage to make it around one another.

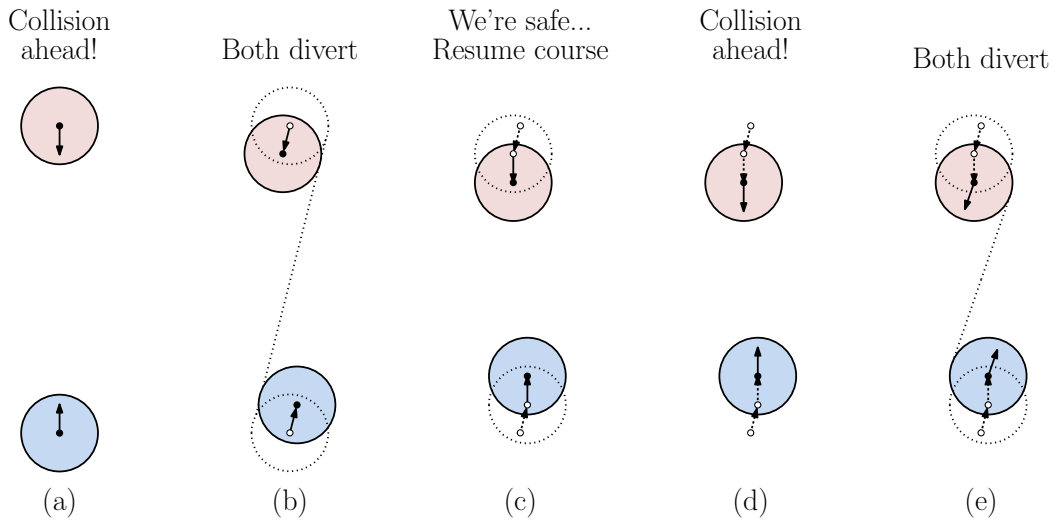


Fig. 104: Oscillation that can result from standard velocity-obstacle motion planning.

Although even humans sometimes engage in this sort of brief oscillation when meeting each other head-on, this sort of repeated oscillation is very unnatural, and is due to the fact that both agents are acting without consideration for what the other agent might reasonable do. The question then is how to fix this?

Reciprocal Velocity Obstacles: The intuition behind fixing the oscillation issue is to share responsibility. We assume that whenever a collision is possible between two agents, both agents perceive the danger and (since they are running the same algorithm) they both know how to avoid it. Rather than having

each agent assume total responsibility for correcting its velocity, we instead ask each agent to take on *half* of the responsibility for avoiding the collision. In other words, each agent diverts its path (that is, alters its velocity) by exactly half the amount needed to avoid the collision. It assumes that the other agent will *reciprocate*, by performing the other half of the diversion. It turns out that this greatly reduces the oscillation problem, since two head-on agents will now divert just enough to avoid each other.

This leads to the concept of *reciprocal velocity obstacles*. Before defining this notion, let us recall the sets $\mathrm{CA}^{\tau}_{a|b}(V_b)$, the collision-avoiding velocities for a assuming that b selects its velocity from V_b, and $\mathrm{CA}^{\tau}_{b|a}(V_a)$, the collision-avoiding velocities for b assuming that a selects its velocity from V_a. We say that two sets of candidate velocities V_a and V_b are *reciprocally collision avoiding* if

$$V_a \subseteq \mathrm{CA}^{\tau}_{a|b}(V_b) \qquad \text{and} \qquad V_b \subseteq \mathrm{CA}^{\tau}_{b|a}(V_a).$$

This implies a very harmonious set of candidate velocities, since for any choice $v_a \in V_a$ and $v_b \in V_b$, we can be assured that these two agents will not collide.

Note that there is a complimentary relationship between these two candidate sets. As we increase the possible velocities in V_a, we reduce the possible set of velocities that b can use to avoid a collision, and vice versa. Of course, we would like be as generous as we can, by giving each agent as much flexibility as possible. We say that two such candidate velocity sets are *reciprocally maximal* if

$$V_a = \mathrm{CA}^{\tau}_{a|b}(V_b) \qquad \text{and} \qquad V_b = \mathrm{CA}^{\tau}_{b|a}(V_a).$$

Note that we face a tradeoff here, since we could make V_a very large, but at the expense of making V_b very small, and vice versa. There are infinitely many reciprocally maximal collision avoiding sets. So what should guide our search for the best combination of candidate sets? Recall that each agent has its preferred velocity, v_a^* and v_b^*. It would seem natural to generate these sets in a manner that gives each agent the greatest number of options that are close to its preferred velocity. We seek a pair of candidate velocity sets that are *optimal* in the sense that they provide each agent the greatest number of velocities that are close to the agent's preferred velocity.

There are a number of ways of making this concept more formal. Here is one. Consider two pairs (V_a, V_b) and (V_a', V_b') of reciprocally maximal collision avoiding sets. For any radius r, $B(v_a^*, r)$ denotes the set of velocities that are within distance r of a's preferred velocity and $B(v_b^*, r)$ denotes the set of velocities that are within distance r of b's preferred velocity. The quantity $\mathrm{area}(V_a \cap B(v_a^*, r))$ can be thought of as the "number" (more accurately the measure) of candidate velocities for a that are close (within distance r) of its preferred velocity. Ideally, we would like both $\mathrm{area}(V_a \cap B(v_a^*, r))$ and $\mathrm{area}(V_b \cap B(v_b^*, r))$ to be large, so that both agents have access to a large number of preferred directions. One way to guarantee that two numbers are large is to guarantee that their minimum is large. Also, we would like the pair (V_a, V_b) to be fair to both agents, in the sense that $\mathrm{area}(V_a \cap B(v_a^*, r)) = \mathrm{area}(V_b \cap B(v_b^*, r))$. This means that they both agents have access to the same "number" of nearby velocities.

Combining the concepts of fairness and maximality, we say that a pair (V_a, V_b) of reciprocally maximal collision avoiding sets is *optimal* if, for all radii $r > 0$, we have

Fair: $\mathrm{area}(V_a \cap B(v_a^*, r)) = \mathrm{area}(V_b \cap B(v_b^*, r))$

Maximal: For any other reciprocal collision avoiding set (V_a', V_b'),

$$\min(\mathrm{area}(V_a \cap B(v_a^*, r)), \mathrm{area}(V_b \cap B(v_b^*, r))) \geq \min(\mathrm{area}(V_a' \cap B(v_a^*, r)), \mathrm{area}(V_b' \cap B(v_b^*, r))).$$

Now that we have defined this concept, it is only natural to ask whether we have any hope of computing a pair of sets satisfying such lofty requirements. The remarkable answer is yes, and in fact, it is not that hard to do! The solution is described in a paper by J. van den Berg, M. C. Lin, D. Manocha (see the readings at the start of these notes). They define an *optimal reciprocal collision avoiding pair*

of candidate velocities, which they denote by $(\mathrm{ORCA}^\tau_{a|b}, \mathrm{ORCA}^\tau_{b|a})$, to be a pair of velocity sets that satisfy the above optimality properties.

They show how to compute these two sets as follows. First, let us assume that the preferred velocities of the two agents puts them on a collision course. (This is just for the sake of illustration. The construction works even if this is not the case.) That is, $v^*_a - v^*_b \in \mathrm{VO}^\tau_{a|b}$. Clearly, we need to divert one agent or both to avoid the collision, and we will like this diversion to be as small as possible. Let u denote the vector on the boundary of $\mathrm{VO}^\tau_{a|b}$ that lies closest to $v^*_a - v^*_b$ (see Fig. 105(a)). Since $\mathrm{VO}^\tau_{a|b}$ is just a truncated cone, it is not too hard to compute the vector u. (There are basically two cases, depending on whether the closest boundary point lies on one of the flat sides or on the circular arc at the base.)

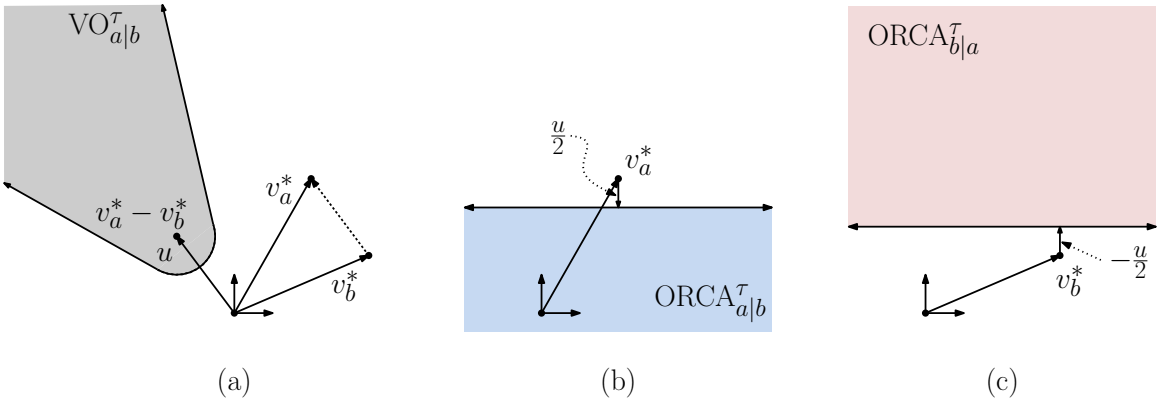


Fig. 105: Computing an optimal reciprocal collision avoiding pair of candidate velocities.

Intuitively, u reflects the amount of relative velocity diversion needed to just barely escape from the collision zone. That is, together a's diversion plus b's diversion (negated) must sum to u. We could split the responsibility however we like to. As we had discussed earlier, for the sake reciprocity, we would prefer that each agent diverts by exactly half of the full amount. That is, a will divert by $u/2$ and b will divert by $-u/2$. (To see why this works, suppose that $v'_a = v^*_a + u/2$ and $v'_b = v^*_b - u/2$. The resulting relative velocity is $v'_a - v'_b = v^*_a - v^*_b + u$, which is a collision-free velocity.)

In general, there are a number of choices that a and b could make to avoid a collision. Let n denote a vector of unit length that points in the same direction as u. We would like a to change its velocity from v^*_a to a velocity whose orthogonal projection onto the vector n is of length at least $\|u/2\|$. The set of allowable diversions defines a half-space (that is, the set of points lying to one side of a line), and is given by the following formula:

$$\mathrm{ORCA}^\tau_{a|b} \;=\; \left\{ v : \left(v - \left(v^*_a + \frac{u}{2} \right) \right) \cdot n \geq 0 \right\},$$

(where the $\cdot$ denotes the dot product of vectors). This formula defines a halfspace that is orthogonal to u and lies at distance $\|u/2\|$ from v^*_a (see Fig. 105(b)). Define $\mathrm{ORCA}^\tau_{b|a}$, symmetrically, but using $-\frac{u}{2}$ instead (see Fig. 105(c)). In their paper, van den Berg, Lin, and Manocha claim that the resulting pair of sets $(\mathrm{ORCA}^\tau_{a|b}, \mathrm{ORCA}^\tau_{b|a})$ define an optimal reciprocally maximal pair of collision avoiding. In other words, if a selects *any* velocity from $\mathrm{ORCA}^\tau_{a|b}$ and b selects any velocity from $\mathrm{ORCA}^\tau_{b|a}$, and these two sets are both fair and provide the greatest number of velocities that are close to both a and b's ideal velocities.

This suggests a solution to the problem of planning the motion of n bodies. Let $B = \{b_1, \ldots, b_n\}$ denote the set of bodies other than a. Compute the ORCA sets for a relative to all the other agents in the system. That is, $\bigcap_{i=1}^n \mathrm{ORCA}^\tau_{a|b_i}$. Since each of these regions is a halfplane, there intersection

defines a convex polygon. Next, find the point v'_a in this convex polygon that minimizes the distance to v^*_a. This point defines a's next velocity. By repeating this for every agent in your system, the result is a collection of velocities that are mutually collision free, and are as close as possible to the ideal velocities.

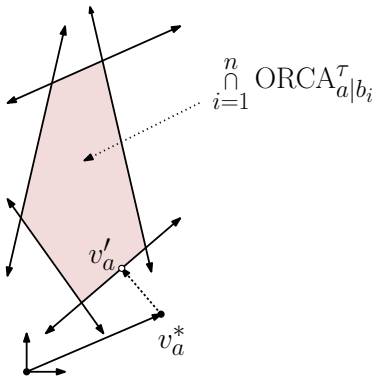


Fig. 106: Computing agent a's velocity.

There are two shortcomings with this approach. First, if the agents are very close to one another, it may be that the intersection of the collision-free regions is empty. In this case, we may need to find an alternate strategy for computing a's velocity (or simply accept the possibility of an intersection).

The other shortcoming is that it requires every agent to know the preferred velocity $v^*_{b_i}$ for each of the other objects in the system. While the simulator may know this, it is not reasonable to assume that every agent knows this information. A reasonable alternative is to form an estimate of the *current velocity*, and use that instead. The theory is that most of the time, objects will tend to move in their preferred direction.

Lecture 22: Multiplayer Games and Networking

Multiplayer Games: Today we will discuss games that involve one or more players communicating through a network. There are many reasons why such games are popular, as opposed say to competing against an AI system.

- People are "better" (less predictable/more complex/more interesting) at strategy than AI systems
- Playing with people provides a social element to the game, allowing players to communicate verbally and engage in other social activities
- Provides larger environments to play in with more characters, resulting in a richer experience
- Some online games support an economy, where players can buy and sell game resources

Multiplayer games come in two broad types:

Transient Games: These games do not maintain a persistent state. Instead players engage in ad hoc, short lived sessions. Examples include games like *Doom*, which provided either head-to-head (one-on-one) or death-match (multiple player) formats. The are characterized as being fast-paced and providing intense interaction/combat. Because of their light-weight nature, any client can be a server.

Persistent Games: These games are run by a centralized authority that maintains a persistent world. Examples include *massively multiplayer online games* (MMOGs), such as "World of Warcraft" (more specifically an MMORPG), which are played over the Internet.

Performance Issues: The most challenging aspects of the design of multiplayer networked games involve achieving good performance given a shared resource (the network).

Bandwidth: This refers to the amount of data that can be sent through the network in steady-state.

Latency: In games where real-time response is important, a more important issue than bandwidth is the responsiveness of the network to sudden changes in the state. Latency refers to the time it takes for a change in state to be transmitted through the network.

Reliability: Network communication occurs over physical media that are subject to errors, either due to physical problems (interference in wireless signals) or exceeding the network's capacity (packet losses due to congestion).

Security: Network communications can be intercepted by unauthorized users (for the purpose of stealing passwords or credit-card numbers) or modified (for the sake of cheating). Since cheating can harm the experience of legitimate users, it is important to detect and minimize the negative effects of cheaters.

Of course, all of these considerations interact and trade-offs must be made. For example, enhancing security or reliability may require more complex communication protocols, which can have the effect of reducing the useable bandwidth or increasing latency.

Network Structure: Networks are complex entities to engineer. In order to bring order to this topic, networks are often described in a series of layers, which is called the *Open System Interconnect* (OSI) model. Here are the layers of the model, from bottom (physical) to the top (applications).

Physical: This is the physical medium that carries the data (e.g., copper wire, optical fiber, wireless, etc.)

Data Link: Deals with low-level transmission of data between machines on the network. Issues at this level include things like packet structure, basic error control, and machine (MAC) addresses.

Network: This controls end-to-end delivery of individual packets. It is responsible for routing and balancing network flow. This is the layer where the Internet Protocol (IP) and IP addresses are defined.

Transport: This layer is responsible for transparent end-to-end transfer of data (not just individual packets) between two hosts. This layer defines two important protocols, TCP (transmission control protocol) and UDP (user datagram protocol). This layer defines the notion of a *net address*, which consists of an IP address and a port number. Different port numbers can be used to partition communication between different functions (http, https, smtp, ftp, etc.)

Session: This layer is responsible for establishing, managing, and terminating long-term connections between local and remote applications (e.g., logging in/out, creating and terminating communication sockets).

Presentation: Provides for conversion between incompatible data representations based on differences system or platform, such as character encoding (e.g., ASCII versus Unicode) and byte ordering (highest-order byte first or lowest-order byte first) and other issues such as encryption and compression.

Application: This is the layer where end-user applications reside (e.g., email (smtp), data transfer (ftp, sftp), web browsers (http, https)).

If you are programming a game that will run over the internet, you could well be involved in issues that go as low as the transport layer (as two which protocol, TCP or UDP, you will use), but most programming takes place at the application level.

Packets and Protocols: Online games communicate through a *packet-switched network*, like the Internet, where communications are broken up into small data units, called *packets*, which are then transmitted through the network from the sender and reassembled on the other side by the receiver. (This is in contrast to direct-link communication, such as through a USB cable or circuit-switched communication, which was used for traditional telephone communication.)

In order for communication to be possible, both sides must agree on a *protocol*, that is, the convention for decomposing data into packets, routing and transferring data through the network, and dealing with errors. Communication networks may be unreliable and may connect machines having widely varying manufacturers, operating systems, speed, data formats. Examples of issues in the design of a network protocol include the following:

Packet size/format: Are packets of fixed or variable size? How is data to be laid out within each packet.

Handshaking: This involves the communication exchange to ascertain how data will be transmitted (format, speed, etc.)

Acknowledgments: When data is received, should its reception be acknowledged and, if so, how?

Error checking/correction: If data packets have not been received or if their contents have been corrupted, some form of corrective action must be taken.

Compression: Because of limited bandwidth, it may be necessary to reduce the size of the data being transmitted (either with or without loss of fidelity).

Encryption: Sensitive data may need to be protected from eavesdroppers.

The Problem of Latency: Recall that latency is the time between when the user acts and when the result is perceived (either by the user or by the other players). Because most computer games involve rapid and often unpredictable action and response, latency is arguably the most important challenge in the design of real-time online games. Too much latency makes the game-play harder to understand because the player cannot associate cause with effect. Latency also makes it harder to target objects, because they are not where you predict them to be.

Note that latency is a very different issue from bandwidth. For example, your cable provider may be able to stream a high-definition movie to your television after a 5 second start-up delay. You would not be bothered if the movie starts after such a delay, but you would be very annoyed if your game were to impose this sort of delay on you every time you manipulated the knobs on your game controller.

The amount of latency that can be tolerated depends on the type of game. For example, in a Real-Time Strategy (RTS) game, below 250ms (that is, 1/4 of a second) would be ideal, 250–500ms would be playable, and over 500ms would be noticeable. In a typical First-Person Shooter (FPS), the latency should be smaller, say 150ms would be acceptable. In car racing game or other game that involves fast (twitch) movements, latencies below 100ms would be required. Latencies in excess of 500ms would make it impossible to control the car. Note that the average latency for the simplest transmission (a "ping") on the internet to a geographically nearby server is typically much smaller than these numbers, say on the order of 10–100ms.

There are a number of sources of latency in online games:

Frame rate latency: Data is sent to/received from the network layer once per frame, and user interaction is only sampled once per frame.

Network protocol latency: It takes time for the operating system to put data onto the physical network, and time to get it off a physical network and to an application.

Transmission latency: It takes time for data to be transmitted to/from the server.

Processing latency: The time taken for the server (or client) to compute a response to the input.

There are various techniques that can be used to reduce each of these causes of latency. Unfortunately, some elements (such as network transmission times) are not within your control.

Coping with Latency: Since you cannot eliminate latency, you can try to conceal it. Of course, any approach that you take will introduce errors in some form. The trick is how to create the illusion to your user that he/she is experiencing no latency.

Sacrifice accuracy: Given that the locations and actions of other players may not be known to you, you can attempt to render them approximately. One approach is to ignore the time lag and show a given player information that is known to be out of date. The second is to attempt to estimate (based on recent behavior) where the other player is at the present time and what this player is doing. Both approaches suffer from problems, since a player may make decisions based on either old or erroneous information.

Sacrifice game-play: Deliberately introduce lag into the local player's experience, so that you have enough time to deal with the network. For example, a sword thrust does not occur instantaneously, but after a short wind-up. Although the wind-up may only take a fraction of a second, it provides the network time to send the information through the network that the sword thrust is coming.

Dealing with Latency through Dead Reckoning: One trick for coping with latency from the client's side is to attempt to estimate another player's current position based on its recent history of motion. Each player knows that the information that it receives from the server is out of date, and so we (or actually our game) will attempt extrapolate the player's current position from its past motion. If our estimate is good, this can help compensate for the lag caused by latency. Of course, we must worry about how to patch things up when our predictions turn out to be erroneous.

- Each client maintains precise state for some objects (e.g. local player).
- Each client receives periodic updates of the positions of everyone else, along with their current velocity information, and possibly the acceleration.
- On each frame, the non-local objects are updated by *extrapolating* their most recent position using the available information.
- With a client-server model, each player runs their own version of the game, while the server maintains absolute authority.

Inevitably, inconsistencies will be detected between the extrapolated position of the other player and its actual position. Reconciling these inconsistencies is a challenging problem. There are two obvious options. First, you could just have the player's avatar jump instantaneously to its most recently reported position. Of course, this will not appear to be realistic. The alternative is to smoothly interpolate between the player's hypothesized (but incorrect) position and its newly extrapolated position.

Dealing with Latency through Lag Compensation: As mentioned above, dead reckoning relies on extrapolation, that is, producing estimates of future state based on past state. An alternative approach, called *lag compensation*, is based on *interpolation*. Lag compensation is a server-side technique, which attempts to determine a player's intention.

Here is the idea. Players are subject to latency, which delays in their perception of the world, and so their decisions are based on information that is slightly out of date with the current world state. However, since we can estimate the delay that they are experiencing, we can try to roll-back the world state to a point where we can see exactly what the user saw when they made their decision. We can

then determine what the effect of the user's action would have been in the rolled-back world, and apply that to the present world.

Here is how lag compensation works.

(1) Before executing a player's current user command, the server:
   (a) Computes a fairly accurate estimate of the player's latency.
   (b) Searches the server history (for the current player) for the last world update that was sent to the player and received by the player (just before the player would have issued the movement command).
   (c) From that update (and the one following it based on the exact target time being used), for each player in the update, move the other players *backwards* in time to exactly where they would have been when the current player's user command was generated. (This moving backwards must account for both connection latency and the interpolation amount the client was using that frame.)

(2) Allow the user command to execute, including any weapon firing commands, etc., that will run ray casts against all of the other players in their interpolated, that is, old positions.

(3) Move all of the moved/time-warped players back to their correct/current positions

The idea is that, if a user was aiming accurately based on the information that he/she was seeing, then the system can determine this (assuming it has a good estimate of each player's latency), and credit the player appropriately.

Note that in the step where we move the player backwards in time, this might actually require forcing additional state information backwards, too (for example, whether the player was alive or dead or whether the player was ducking). The end result of lag compensation is that each local client is able to directly aim at other players without having to worry about leading his or her target in order to score a hit. Of course, this behavior is a game design tradeoff.

Reliability: Let us move on from latency to another important networking issue, reliability. As we mentioned before, in packet-switched networks, data are broken up into packets and then may be sent by various routes. Packets may arrive out of order, they may be corrupted, or they may fail to arrive at all (or after such a long delay that the receiver gives up on them). Some network protocols (TCP in particular) attempt to ensure that every packet is delivered and they arrive in order. (For example, if you are sending an email message, you would expect the entire message to arrive as sent.)

As we shall see, achieving such a high level of reliability comes with associated costs. For example, the user sends packets. The receiver acknowledges the receipt of packets to the sender. If a packet receipt is not acknowledged, the sender resends the packet. The additional communication required for sending, receiving, and processing acknowledgments can increase latency and use more of the available bandwidth.

In many online games, however, we may be less concerned that every packet arrives on time or in order. Consider for example a series of packets, each of which tells us where an enemy player is located. If one of these packets does not arrive (or arrives late) the information is now out of date anyway, and there is no point in having the sender resend the packet. Of course, some information is of a much more important nature. Information about payments or certain changes to the discrete state of the game (player X is no longer in the game), must be communicated reliably. In short, not all information in a game is of equal importance with respect to reliability.

Communication reliability is handled by protocols at the transport level of the OSI model. The two most common protocols are TCP (transmission control protocol) and UDP (user datagram protocol).

Transmission Control Protocol: We will not delve into the details of the TCP protocol, but let us highlight its major elements. First, data are transferred in a particular order. Each packet is assigned a unique *sequence number*. When packets are received, they are reordered according to these sequence numbers. Thus, packets may arrive out of order without affecting the overall flow of data. Also, through the use of sequence numbers, the receiver can determine whether any packets were lost. Second, the transmission contains *check-sums*, to ensure that any (random) corruption of the data will be discovered. The receiver sends acknowledgments of the receipt of packets. Thus, if a packet is not received, the sender will discover this and can resend it.

TCP also has a basic capability for *flow control*. If the sender observes that too many packets are failing to arrive, it decreases the rate at which it is sending packets. If almost all packets are arriving, it slowly increases this rate. In this way, the network will not become too congested.

Advantages:

- Guaranteed packet delivery
- Ordered packet delivery
- Packet check-sum checking (basic error detection)
- Transmission flow control

Disadvantages:

- Point-to-point transport (as opposed to more general forms, like multi-cast)
- Bandwidth and latency overhead
- Packets may be delayed to preserve order

TCP is used in applications where data must be reliably sent and/or maintained in order. Since it is a reliable protocol, it can be used in games where latency is not a major concern.

User Datagram Protocol: UDP is a very light-weight protocol, lacking the error control and flow control features of TCP. It is a *connectionless protocol*, which provides no guarantees of delivery. The sender merely sends packets, with no expectation of any acknowledgment. As a result, the overhead is much smaller than for TCP.

Advantages:

- Packet based—so it works with the internet
- Lower overhead than TCP in terms of both bandwidth and latency
- Immediate delivery—as soon as it arrives it goes to the client

Disadvantages:

- Point to point connectivity (as with TCP)
- No reliability guarantees
- No ordering guarantees
- Packets can be corrupted
- Can cause problems with some firewalls

UDP is popular in games, since much state information is nonessential and quickly goes out of date. Note that although the UDP protocol has no built-in mechanisms for error checking or packet acknowledgments, the application can add these to the protocol. For example, if some packets are non-critical, they can be sent by the standard UDP protocol. Certain *critical* packets can be flagged by your application, and as part of the packet payload, it can insert its own sequence numbers and/or check-sums. Thus, although UDP does not automatically support TCPs features, there is nothing preventing your application from adding these to a small subset of important packets.

Area-of-Interest Management: In large massively multiplayer games, it would be inefficient to inform every player on the state of every other player in the system. This raises the question of what information does a player need to be aware of, and how to transmit just that information. This is the subject of the topic of *area-of-interest management*. This subject is to networking what visibility is to collision detection. This is typically employed in large games, and so it is the server's job to determine what information each player receives.

There are two common approaches, grid methods and aura methods. *Grid methods* partition the world into a grid (which more generally may be something like a quadtree). Each cell is associated with the players and other entities that reside within this cell. Then, the information transmitted to a player is based on the entities residing within its own and perhaps neighboring grid cells.

One shortcoming of this method is that it neglects the fact that some entities may not correspond to individual points, but to entire regions of space. For example, a cloud of poisonous gas cannot be associated with a single point in space. The alternative is called an *aura method*, in which each entity is associated with a region of space, its *sphere of influence*. All players that lie within this region are provided information on this entity.

Lecture 23: Detecting and Preventing Cheating in Multiplayer Games

Cheating in Multiplayer Games: "Cheating" is defined to be acting dishonestly or unfairly in order to gain an advantage. In online games, players often strive to obtain an unfair advantage over others, for various reasons. One of the first analyses of cheating in online games appeared around 2000 in a Gamasutra article by Matthew Pritchard. He makes the following observations:

- If you build it, they will come to hack and cheat
- Hacking attempts increase as a game becomes more successful
- Cheaters actively try to control knowledge of their cheats
- Your game, along with everything on the cheater's computer, is not secure—not memory, not files, not devices and networks
- Obscurity $\neq$ security
- Any communication over an open line is subject to interception, analysis and modification
- There is no such thing as a harmless cheat
- Trust in the server is everything in client-server games
- Honest players would like the game to tip them off to cheaters

Pritchard identifies a number of common cheating attacks and discusses how to counter them. His list includes the following:

Information Exposure: Clients obtain/modify information that should be hidden.

Reflex Augmentation: Improve physical performance, such as the firing rate or aiming

Authoritative Clients: Although the server should have full authority, some online games grant clients authority over game execution for the sake of efficiency. Cheaters then modify the client software.

Compromised servers: A hacked server that biases game-play towards the group that knows of the hacks.

Bugs and Design Loopholes: Bugs and design flaws in the game are exploited.

Infrastructure Weaknesses: Differences or problems with the operating system or network environment are exploited.

We will discuss some of these in greater detail below.

Reflex Augmentation: Reflex augmentation systems involve the use of software that, through various methods, circumvents the user-based aiming/firing systems to a software-based system.

One example is an *aimbot*. An aimbot is implemented by modifying the game client program or running an external program in order to generate simulated user input. Network packets are intercepted and interpreted to determine the location of enemies and obstacles. Then computer AI is used to completely control the player's avatar and automate repetitive tasks, progressing the player's avatar through the game. Another example is a *reflex enhancer*, which augment a user's input in reflex games to achieve better results. For example, in a shooter game, the software can automatically aim at opponents.

Reflex augmentation typically involves modifying the underlying game executable or modifying one of the system's library functions that the game invokes. Techniques borrowed from the area of virus detection can be employed to be sure that the user has not tampered with the game's binary executable. Some approaches are static, using fingerprinting to scan the player's host memory in search of bit patterns of known cheating applications. A more dynamic approach is to periodically download the original game executable and compare its behavior to the user's game's behavior. If the executable has not been tampered with, then the two should behave identically. If not, the user must have tampered with the code somehow.

A final detection method is to perform statistical analysis of a user's performance. If a player is playing too good to be a human, then he/she probably isn't. Of course, if an aimbot is sufficiently well designed to fly "just below the radar," (playing just slightly above the level of an expert), it is possible to defeat any such analysis.

Information Exposure: This method of cheating involves the cheater gaining access to information that they are not entitled to, such as their opponent's health, weapons, resources, troops. This cheat is possible as developers often incorrectly assume that the client software can be trusted not to reveal secrets. Secret information is revealed by either modifying the client or running another program that extracts it from memory.

Another approach for doing this is to modify elements of the graphics model. For example, suppose that you have a shooter game, where enemies may hide behind walls, bushes, or may rely on atmospheric effects like smoke or fog. The cheater then modifies the parameters that control these obscuring elements, say by making walls transparent, removing the foliage on bushes, and changing the smoke parameters so it effectively disappears. The cheating application alone now has an un-obscured view of the battlefield.

This is sometimes called an *infrastructure-level cheat*, since it usually involves accessing or modifying elements of the infrastructure in which the program runs. In a client-server setting, this can be dealt with is using a technique called on-demand-loading (ODL). Using this technique a trusted third party (the server) stores all secret information and only transmits it to the client when they are entitled to it. Therefore, the client does not have any secret information that may be exposed. Another approach for avoiding information exposure is to encrypt all secret information. This makes it difficult to determine where the information is and how to interpret its meaning.

Protocol-level cheats: Because most multiplayer games involve communication through a network, many cheats are based on interfering with the manner in which network packets are processed. Packets may be inserted, destroyed, duplicated, or modified by an attacker. Many of these cheats are dependent on the architecture used by the game (client-server or peer-to-peer). Below we describe some protocol-level cheats.

Suppressed update: As we mentioned last time, the Internet is subject latency and packet loss. For this reason, most networked games use some form of dead-reckoning. In the event of a lost/delayed updates, the server will extrapolate (dead-reckon) the players movement from their most recent position and velocity, creating a smooth movement for all other players. Dead-reckoning usually allows clients to drop some fixed number of consecutive packets before they are disconnected. In the suppressed update cheat, a cheater purposely *suppresses* the transmission of some fixed number consecutive updates (but not so many to be disconnected), while still accepting opponent updates. The attacker (who is able to see the other player's movements during this time) calculates the optimal move using the updates from their opponents and transmits it to the server. Thus, the cheater knows their opponents actions before committing to their own, allowing them to choose the optimal action.

Architectures with a trusted entity (e.g., the server), can prevent this cheat by making the server's dead-reckoned state authoritative (as opposed to allowing the client to do it). Players are forced to follow the dead-reckoned path in the event of lost/delayed updates. This gives a smooth and cheat-free game for all other players; however, it may disadvantage players with slow Internet connections. In a less authoritative environment (e.g., peer-to-peer) it may be possible for other players to monitor the delay in their opponents and compare it with the timestamps of updates. Late updates indicate that a player is either suffering delay, or is cheating.

Fixed delay: Fixed delay cheating involves introducing a fixed amount of delay to all outgoing packets. This results in the local player receiving updates quickly, while delaying information to opponents. For fast paced games this additional delay can have a dramatic impact on the outcome. This cheat is usually used in peer-to-peer games, when one peer is elevated to act as the server. Thus, they can add delay to all other peers.

One way to prevent this cheat in peer-to-peer games can use distributed event ordering and consistency protocols to avoid elevating one peer above the rest. Note, the fixed delay cheat only delays updates, in contrast to dropping them in the suppressed update cheat.

Another solution is to force all players to use a protocol that divides game time into rounds and requires that every player in the game submit their move for that round before the next round is allowed to begin. (One such protocol is called *lockstep*.) To prevent cheating, all players commit to a move, and once all players have committed, each player reveals their move. A player commits to a move by transmitting either the hash of a move or an encrypted copy of a move, and it is revealed by sending either the move or encryption key respectively. Lockstep is provably secure against these and other protocol level cheats. Unfortunately, this approach is unacceptably slow for many fast-paced games, since it forces all players to wait on the slowest one.

Another example of a protocol to prevent packet suppression/delaying is called *sliding pipeline* (SP). SP works by constantly monitoring the delay between players to determine the maximum allowable delay for an update without allowing times-stamp cheating (see below). SP does not lock all players into a fixed time step, and so can be applied to faster-paced games. Unfortunately, SP cannot always differentiate between players suffering delay and cheaters (false positives).

More Protocol-Level Cheats: The above (suppressed update and fixed delay) are just two examples of protocol-level cheats. There are many others, which we will just summarize briefly here.

Inconsistency: A cheater induces inconsistency amongst players by sending different game updates to different opponents. An honest player attacked by this cheat may have his game state corrupted, and hence be removed from the game, by a cheater sending a different update to him than was sent to all other players. To prevent this cheat updates sent between players must be verified by either a trusted authority, or a group of peers.

Time-stamp: This cheat is enabled in games where an untrusted client is allowed to time-stamp their updates for event ordering. This allows cheaters to time-stamp their updates in the past,

after receiving updates from their opponents. Hence, they can perform actions with additional information honest players do not have. To prevent this, rather than using timestamps, processing should be based on the arrival order of updates to the server.

Collusion: Collusion involves two or more cheaters working together (rather than in competition) to gain an unfair advantage. One common example is of players participating in an all-against-all style match, where two cheaters will team up (collude) against the other players. Colluding players may communicate over an external channel (e.g., over the phone or instant messaging). This is very hard to detect and prevent.

Spoofing: Spoofing is where a cheater sends a message masquerading as a different player. For example, a cheater may send an update causing an honest player to drop all of their items. To prevent this cheat, updates should be either digitally signed or encrypted.

If a cheater receives digitally signed/encrypted copies of an opponent's updates he may still be able to disadvantage an opponent by resending them at a later time. Since the updates are correctly signed or encrypted, they will be assumed valid by the receiver. To prevent updates should include a unique number, such as a round number or sequence number, which the receiver can then check to ensure the message is genuine.