CMSC 425
Game Programming[1]

David M. Mount
Department of Computer Science
University of Maryland
Spring 2013

Lecture 1: Introduction to Game Programming

Sources: Further information can be found in the first chapter of *Introduction to Game Development* (2nd edition), ed. S. Rabin, 2009.

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 hand-held 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 hand-held devices, like the iPhone.

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: There has been a great deal of software produced to aid in the generation of large-scale software systems for computer games: geometric modelers, game engines, physics engines, and more. Employees of a large game software companies do not design games from first principles, but rather by these tools.

Unfortunately, teaching to the technology is not really our interest here. The technology that is current today may be obsolete in a few years. Our focus in this course will not be on how to use existing software systems,

but rather on the principles that underlie them. 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 and practice that underlies the implementation of these systems.

Of course, the amount of material is vast, and we could easily fill a number of courses covering this material. 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 "theory 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 structure: The structure of a generic game and basic components of game engines.

Interactive computer graphics: Interactive 2- and 3-dimensinoal computer graphics, including how GPUs work, basic OpenGL and GLUT, event-driven I/O, simple illumination models, and texture mapping.

Modeling for games: Geometric models and representations, and algorithms for manipulating them.

AI for games: Agents and how to control them. Agent state transition, navigation algorithms, flocking behavior.

Physics for games: A short introduction to kinematics and kinetics, integration methods, and the elements of a typical physics engine.

Networking for games: Basic elements of network programming, including sockets, ports, and basic protocols, such as IP and UDP, and latency hiding.

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

Lecture 2: Computer Game and Graphics System Architectures

Sources: The first half of the lecture is taken from Chapt 1 of Gregory, *Game Engine Architecture*. The second half comes from any modern computer graphics text.

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 (e.g., *Tetris*, *Mahjong*, or *Bejewled*) is very different from a high-end 3-dimensional first-person shooter game (e.g., *Unreal*, *Call of Duty*, or *Halo*).

In order to show the contrast of requirements among different games, here are some examples of a few common game genres and the typical technical elements required to implement these games.

First-Person Shooters (FPS): These games involve simulating a relatively slow entity moving on foot through a highly structured world, typified by passageways and corridors. Modern versions can include more complex types of locomotion, such as climbing. Typicial features of these games include:

- real-time rendering of large 3-dimensional virtual worlds
- highly responsive camera control/aiming mechanics
- high fidelity animations of player's virtual arms and weapons
- availability of a separate (through the scope) camera mode for accurate aiming
- a range of hand-held weapons
- forgiving player character motion and a loose collision model, which gives the feeling of floating,
- high fidelity animations of non-player characters (both enemies and team members)

- multiplayer capabilities

In order to achieve real-time rendering of complex environments, these games frequently limit player movement to well-defined regions, and use various culling techniques that allow the game to draw only the small amount of the total world, which is visible to the player. This is among the most challenging genres from the perspective of the complexity and scale of the technologies required.

Platform Games: This genre is typified by games like *Donkey Kong*, *Super Mario Bros* and *Sonic the Hedgehog*. The player controls an animated character that is seen externally, that is, from a *third-person perspective*. This character moves through an obstacle course in pursuit of a goal. This may involve running, jumping, swinging, and climbing. Features of these games include:

- moving platforms, ladders, ropes, trellises and generally a variety of locomotion modes
- puzzle-like environmental elements
- third-person camera follows the player character
- (particularly for 3-dimensional platform games) complex camera collision system that ensures that the player character is alwars in view

In contrast to FPS games, accuracy in locomotion is a key element to the game play. Players must accurately time their jumps, or combine running with jumping to avoid specific obstacles. In order to keep the player character continuously in view, it may be necessary to "cheat" on the geometry by allowing the camera to see through obstacles (e.g., walls) that are very close to the character, since they would otherwise block the player's ability to see the character.

Fighting Games: These involve two players contolling humanoid characters that battle one another. Common examples include *Street Fighter*, *Tekken*, and includes sports boxing games like *Fight Night*. Features of these games include:

- a variety of fighting animations
- accurate hit detection
- input system capable of complex button/joystick combinations
- simple backgrounds
- simple camera motion focused on the center of the action

Racing Games: These games involve simulating driving a vehicle (typically an automobile) on a track. These include realistic games, such as *Need for Speed* and *Gran Turismo*. and cartoonish ones, such as *Mario Kart*. Like FPS games, these are highly linear. (Note that this does not include games like *Grand Theft Auto* (GTA). Although GTA involves driving simulation, it is not considered racing since the driving is not on a track. Games like GTA where the player can explore vast regions of the world are referred to as *open world* games.) Some features of racing games include:

- real-time rendering of highly detailed backgrounds that are moving very rapidly through the field of view (which means that various tricks can be used to render them, since the viewer cannot focus on them for long)
- the AI for nonplaying vehicles can be simplified due to the need to stick to the racing circuit
- camera may follow the vehicle from a third-person perspective or may be situated behind the steering wheel from a first-person perspective
- to accurately render the effects of collisions, vehicle modeling may need to be flexible to render dented or scratched vehicle bodies

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 the *Unreal Engine*, Microsoft's *XNA Game Studio*, and *Ogre3D*.

Game engines vary along a spectrum of ease of use and flexibility. Simple game engines can generate only a single type of game, 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.

System: This is the hardware and operating system on which the game runs, such as general personal computers (running, say, Microsoft Windows, Linux, or Mac OS), game consoles (XBox, Playstation, Wii), or mobile devices (handheld game consoles or smart-phones).

Third-Party SDKs and Middleware: These are libraries and sofware development toolkits (SDKs), usually provided from a third party. Examples include graphics (OpenGL and DirectX), physics (Havok, PhysX), basic algorithms and data structures (C++ STL, Boost++), character animation (Granny), networking support (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 (e.g., for xml-based input files), 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. As we will see below when we discuss OpenGL, your program interacts with the GPU by asking it to render *objects*. Each object may be a single triangle (or generally a polygon) or 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. In our discussion of OpenGL, we will discuss how this is done. 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 30–60 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
- image processing effects

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 consistutes 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: 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. It has its own memory, separate from the CPU's memory, in which it stores the various graphics objects (e.g., object

coordinates and texture images) that it needs in order to do its job. 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. Another entity, called the *video controller*, reads the contents of the frame buffer and generates the actual image on the monitor. This process is illustrated in schematic form in Fig. 1.

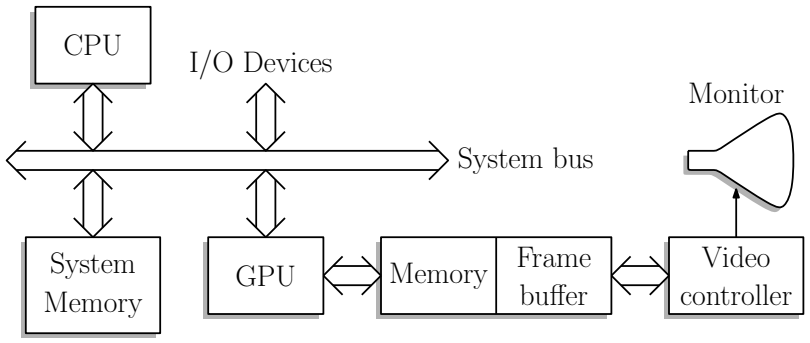


Fig. 1: Architecture of a simple GPU-based graphics system.

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 four major stages. (This is mostly a conceptual aid, since the GPU architecture is not divided so cleanly.) The process is illustrated in Fig. 2.

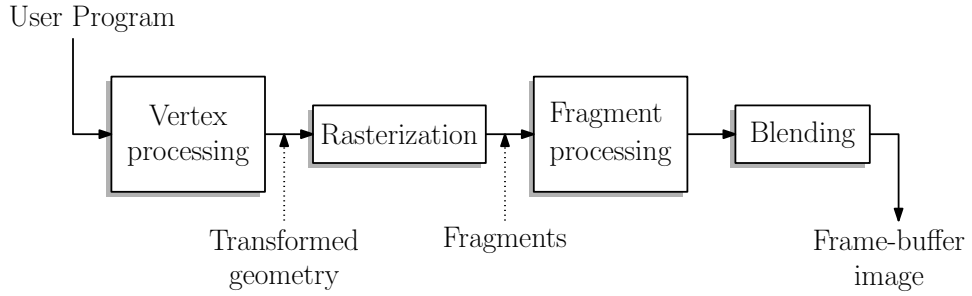


Fig. 2: Stages of the graphics pipeline.

Vertex Processing: Geometric objects 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). In the vertex processing stage, the graphics system *transforms* these coordinates 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*.

This stage involves other tasks as well. For one, *clipping* is performed to snip off any parts of your geometry that lie outside the viewing area of the window on your display. Another operation is *lighting*, where computations are performed to determine the colors and intensities of the vertices of your objects. (How the lighting is performed depends on commands that you send to the GPU, indicating where the light sources are and how bright they are.)

Rasterization: The job of the rasterizer is to convert the geometric shape given in terms of its screen coordinates into individual pixels, called *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.

Blending: 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. The final output of this stage is the *frame-buffer image*.

Lecture 3: Basic Elements of OpenGL and GLUT

Sources: See any standard reference on OpenGL or GLUT.

Graphics Libraries: In this lecture we will discuss programming a 3-dimensional interactive graphics system. In particular, we will discuss OpenGL and GLUT. The essence of producing interactive graphics involves generating a description of the scene being rendered, and repeating this process at a rate of roughly 30 frames per second. We call each such redrawing a *display cycle* or a *refresh cycle*, since your program is refresh the current contents of the image.

In modern graphics systems, the image is generated by the graphics processing unit (GPU), and the job of your program is to describe the scene to be displayed to the GPU so it can be rendered. The commands sent to the GPU take the form of procedure calls made by your program to a graphics library. The procedures of the library are defined according to an application programmer's interface (API). There are a few common graphics APIs. Generally, they come in two different types:

Retained Mode: The system maintains the state of the computation in its own internal data structures. With each refresh cycle, this data is transmitted to the GPU for rendering. A *scene graph* is typically employed as the method for storing this information. Examples of retained-mode systems include *Java3d*, *Ogre3D*, and *Open Scenegraph*. The principal advantage is that global optimizations can be performed, since all the scene information is known to the system. This approach is less well suited to time-varying data sets, since the internal representation of the data set needs to be updated frequently.

Immediate Mode: In this sort of system, your program provides all the information needed to draw each scene with each display cycle. In other words, your program transmits commands directly to the GPU for execution. Examples of immediate-mode systems include *OpenGL* and *DirectX*. Immediate-mode systems tend to be more efficient, since they provide closer control of the hardware. The principal disadvantage is that it not possible for the system to perform the sort of global optimizations (such as culling non-visible entities), which is possible with retained-mode approaches.

OpenGL: OpenGL is a widely used industry standard graphics API. It has been ported to virtually all major systems, and can be accessed from a number of different programming languages (C, C++, Java, Python, ...). Because it

works across many different platforms, it is very general. This is in contrast to the principal alternative, DirectX, which has been designed to work primarily on Microsoft systems.

For the most part, OpenGL operates in *immediate mode*, which means that each function call results in a command being sent directly to the GPU. There are some retained elements, however. For example, transformations, lighting, and texturing need to be set up, so that they can be applied later in the computation.

Because of the design goal of being independent of the window system and operating system, OpenGL does *not* provide capabilities for windowing tasks or user input and output. For example, there are no commands in OpenGL to create a window, to resize a window, to determine the current mouse coordinates, or to detect whether a keyboard key has been hit. Everything is focused just on the process of generating an image. In order to achieve these other goals, it is necessary to use an additional *toolkit*. There are a number of different toolkits, which provide various capabilities. We will cover a very simple one in this class, called *GLUT*, which stands for the *GL Utility Toolkit*. GLUT has the virtue of being very simple, but it does not have a lot of features. To get these features, you will need to use a more sophisticated toolkit.

There are many, many tasks needed in a typical large graphics system. As a result, there are a number of software systems available that provide utility functions. For example, suppose that you want to draw a sphere. OpenGL does not have a command for drawing spheres, but it can draw triangles. What you would like is a utility function which, given the center and radius of a sphere, will produce a collection of triangles that approximate the sphere's shape. OpenGL provides a simple collection of utilities, called the *GL Utility Library* or *GLU* for short.

Since we will be discussing a number of the library functions for OpenGL, GLU, and GLUT during the next few lectures, let me mention that it is possible to determine which library a function comes from by its prefix. Functions from the OpenGL library begin with "gl" (as in "glTriangle"), functions from GLU begin with "glu" (as in "gluLookAt"), and functions from GLUT begin with "glut" (as in "glutCreateWindow").

The Main Program: Before discussing how to draw shapes, we will begin with the basic elements of how to create a window. OpenGL was intentionally designed to be independent of any specific window system. Consequently, a number of the basic window and system operations are not provided. This is the principal reason for *GLUT*. This toolkit which provides the necessary tools for requesting that windows be created and providing interaction with I/O devices.

Let us begin by considering a typical main program (see the following code fragment). Throughout, we will assume that programming is done in C/C++, but our examples can be readily adapted to other languages. (Do not worry for now if you do not understand the meanings of the various calls. We will discuss them in greater detail below.) This program creates a window that is 400 pixels wide and 300 pixels high, located in the upper left corner of the display.

Note that the call to glutMainLoop turns control over to the system. After this, the only return to your program will occur due to various events, called *callbacks*. (The final "return 0" is only there to keep the compiler from issuing a warning.) Here is an explanation of these functions.

glutInit: The arguments given to the main program (argc and argv) are the command-line arguments supplied to the program. This assumes a typical Unix environment, in which the program is invoked from a command line. We pass these into the main initialization procedure, glutInit. This procedure must be called before any others. It processes (and removes) command-line arguments that may be of interest to GLUT (e.g., allowing the user to override the default window size) and the window system and does general initializations. Any remaining arguments are then left for the user's program to interpret, if desired.

glutInitDisplayMode: This performs initializations informing OpenGL how to set up its frame buffer. Recall that the frame buffer is a special 2-dimensional array in the GPU's memory where the graphical image is stored. OpenGL maintains an enhanced version of the frame buffer with additional information. For example, this includes depth information for hidden surface removal. The system needs to know how we are representing colors of our general needs in order to determine the *depth* (that is, the number of bits)

Typical OpenGL/GLUT Main Program

```
#include <GL/glut.h>                          // GLUT, GLU, and OpenGL defs
int main(int argc, char** argv)               // program arguments
{
    glutInit(&argc, argv);                    // initialize glut and openGL
                                              // double buffering and RGBA
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowSize(400, 300);             // initial window size
    glutInitWindowPosition(0, 0);             // initial window position
    glutCreateWindow(argv[0]);                // create window

    ...initialize callbacks here (described below)...

    myInit();                                 // your own initializations
    glutMainLoop();                           // turn control over to glut
    return 0; // we never return here; this just keeps the compiler happy
}
```

to assign for each pixel in the frame buffer. The argument to glutInitDisplayMode is a logical-or (using the operator "|") of a number of possible options, which are given in Table 1.

Display Mode	Meaning
GLUT_RGB	Use RGB colors
GLUT_RGBA	Use RGB plus α (recommended)
GLUT_INDEX	Use colormapped colors (not recommended)
GLUT_DOUBLE	Use double buffering (recommended)
GLUT_SINGLE	Use single buffering (not recommended)
GLUT_DEPTH	Use depth buffer (needed for hidden surface removal)

Table 1: Arguments to glutInitDisplayMode. (Constants defined in glut.h)

.

Let us discuss each of these elements in greater detail.

Color: We need to tell the system how colors will be represented. There are three methods, of which two are fairly commonly used: GLUT_RGB or GLUT_RGBA. The first uses standard RGB colors (24-bit color, consisting of 8 bits of red, green, and blue), and is the default. The second requests RGBA coloring. In this color system there is a fourth component (designated "A" or α), which indicates the opaqueness of the color (1 = fully opaque, 0 = fully transparent). This is useful in creating transparent effects. We will discuss how this is applied later this semester. (It turns out that there is no advantage in trying to save space using GLUT_RGB over GLUT_RGBA, since according to the GLUT documentation, both are treated the same.)

Single or Double Buffering: The next option specifies whether single or double buffering is to be used, GLUT_SINGLE or GLUT_DOUBLE, respectively. To explain the difference, we need to understand a bit more about how the frame buffer works. In raster graphics systems, whatever is written to the frame buffer is immediately transferred to the display. This process is repeated frequently, say 30–60 times a second. To do this, the typical approach is to first erase the old contents by setting all the pixels to some background color, say black. After this, the new contents are drawn. However, even though it might happen very fast, the process of setting the image to black and then redrawing everything produces a noticeable flicker in the image.

Double buffering is a method to eliminate this flicker. In double buffering, the system maintains two separate frame buffers. The *front buffer* is the one which is displayed, and the *back buffer* is the other

one. Drawing is always done to the back buffer. Then to update the image, the system simply swaps the two buffers (see Fig. 3). The swapping process is very fast, and appears to happen instantaneously (with no flicker). Double buffering requires twice the buffer space as single buffering, but since memory is relatively cheap these days, it is the preferred method for interactive graphics.

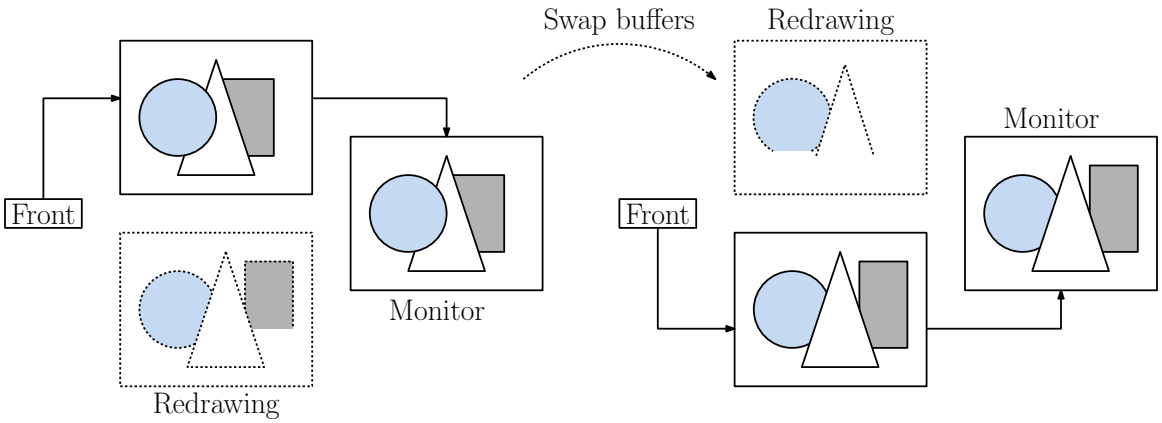


Fig. 3: Double buffering.

Depth Buffer: One other option that we will need later with 3-dimensional graphics will be hidden surface removal. Virtually all raster-based interactive graphics systems perform hidden surface removal by an approach called the *depth-buffer* or z*-buffer*. In such a system, each fragment stores its distance from the eye. When fragments are rendered as pixels only the closest is actually drawn. The depth buffer is enabled with the option GLUT_DEPTH. For this program it is not needed, and so has been omitted. When there is no depth buffer, the last pixel to be drawn is the one that you see.

glutInitWindowSize: This command specifies the desired width and height of the graphics window. The general form is glutInitWindowSize(int width, int height). The values are given in numbers of pixels.

glutInitPosition: This command specifies the location of the upper left corner of the graphics window. The form is glutInitWindowPosition(int x, int y) where the (x, y) coordinates are given relative to the upper left corner of the display. Thus, the arguments $(0, 0)$ places the window in the upper left corner of the display. Note that glutInitWindowSize and glutInitWindowPosition are both considered to be only *suggestions* to the system as to how to where to place the graphics window. Depending on the window system's policies, and the size of the display, it may not honor these requests.

glutCreateWindow: This command actually creates the graphics window. The general form of the command is glutCreateWindow(char* title), where title is a character string. Each window has a title, and the argument is a string which specifies the window's title. We pass in argv[0]. In Unix argv[0] is the name of the program (the executable file name) so our graphics window's name is the same as the name of our program.

Asynchonous Creation: Note that the call glutCreateWindow does not really create the window, but rather merely sends a request to the system that the window be created. Why do you care? In some OpenGL implementations, certain operations cannot be performed unless the window (and the associated graphics context) exists. Such operations should be performed only *after* your program has received notification that the window really exists. Your program is informed of this event through an event callback, either *reshape* or *display*. We will discuss them below.

The general structure of an OpenGL program using GLUT is shown in Fig. 4. (Don't worry if some elements are unfamiliar. We will discuss them below.)

Event-driven Programming and Callbacks: Virtually all interactive graphics programs are *event driven*. Unlike traditional programs that read from a standard input file, a graphics program must be prepared at any time for

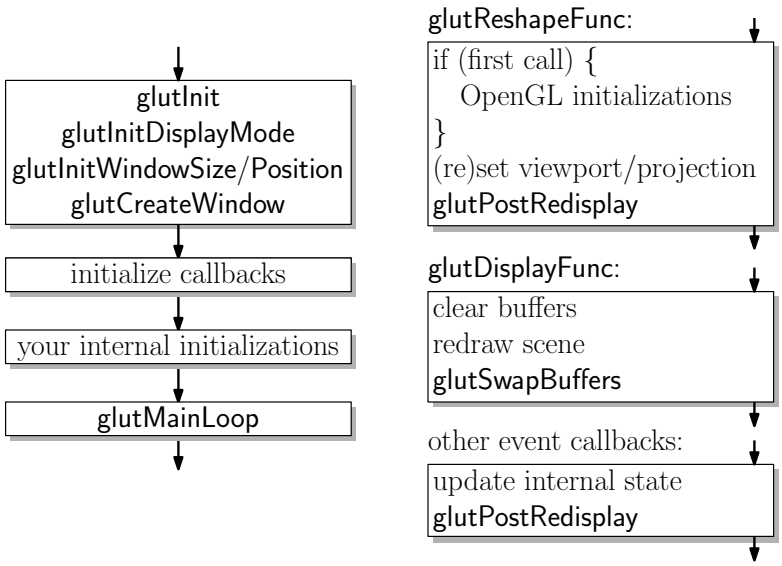


Fig. 4: General structure of an OpenGL program using GLUT.

input from any number of sources, including the mouse, or keyboard, or other graphics devises such as trackballs and joysticks.

In OpenGL this is done through the use of *callbacks*. The graphics program instructs the system to invoke a particular procedure whenever an event of interest occurs, say, the mouse button is clicked. The graphics program indicates its interest, or *registers*, for various events. This involves telling the window system which event type you are interested in, and passing it the name of a procedure you have written to handle the event.

Note: If you program in C++, note that the Glut callback functions you define must be "standard" procedures; they cannot be class member functions.

Types of Callbacks: Callbacks are used for two purposes, *user input events* and *system events*. User input events include things such as mouse clicks, the motion of the mouse (without clicking) also called *passive motion*, keyboard hits. Note that your program is only signaled about events that happen to your window. For example, entering text into another window's dialogue box will not generate a keyboard event for your program. Here are some common examples. A summary is given in Table 2. There are a number of others as well (special keys, status of shift/control/alt, detecting key releases.)

System Event	*Callback request*	*User callback function prototype* (return void)
(Re)display	glutDisplayFunc	myDisplay()
(Re)size window	glutReshapeFunc	myReshape(int w, int h)
Timer event	glutTimerFunc	myTimer(int id)
Idle event	glutIdleFunc	myIdle()
Input Event	*Callback request*	*User callback function prototype* (return void)
Mouse button	glutMouseFunc	myMouse(int b, int s, int x, int y)
Mouse motion	glutPassiveMotionFunc	myMotion(int x, int y)
Keyboard key	glutKeyboardFunc	myKeyboard(unsigned char c, int x, int y)

Table 2: Common callbacks and the associated registration functions.

Display Event: At a minimum, every OpenGL program must handle this event. It is invoked when the system senses that the contents of the graphics window need to be redrawn, for example:

- the window is created initially or it has been resized,
- the window system has determined that the graphics window has been "damaged" (which might occur, for example, if an obscuring window has been moved away, thus revealing all or part of the graphics window),
- the program explicitly requests redrawing, for example, because the internal state has changed in a way that affects the scene, by calling glutPostRedisplay.

Recall from above that the command glutCreateWindow does not actually create the window, but merely requests that creation be started. In order to inform your program that the creation has completed, the system generates a display event. This is how you know that you can now start drawing into the graphics window.

Reshape Event: This happens whenever the window's size is altered, including its initial creation. The callback provides information on the new size of the window. Recall that your initial call to glutInitWindowSize is only taken as a suggestion of the actual window size. When the system determines the actual size of your window, it generates such a callback to inform you of the actual size.

Idle/Timer Events: Often in an interactive graphics program, the user may not be providing any input at all, but it may still be necessary to update the image. For example, in a flight simulator the plane keeps moving forward, even without user input. To do this, the program goes to sleep and requests that it be awakened in order to draw the next image. There are two ways to do this, a *timer event* and an *idle event*. An idle event is generated every time the system has nothing better to do. This is often fine, since it means that your program wastes no cycles.

Often, you want to have more precise control of timing (e.g., when trying to manage parallel threads such as artificial intelligence and physics modeling). If so, an alternate approach is to request a timer event. In a timer event you request that your program go to sleep for some period of time and that it be "awakened" by an event some time later, say 1/50 of a second later. In glutTimerFunc the first argument gives the sleep time as an integer in milliseconds and the last argument is an integer identifier, which is passed into the callback function.

Input Events: These events include keyboard key presses and mouse key presses and mouse motion. When any such event occurs, the system will inform you which key has been pressed (or released), and where the cursor is at the instant of the event. Here are the principal input events your program can listen for.

For example, the function calls shown in the code fragment below shows how to register for the following events: display events, reshape events, mouse clicks, keyboard strikes, and timer events. The functions like myDraw and myReshape are supplied by the user, and will be described later.

Typical Callback Setup

```
int main(int argc, char** argv)
{
    ...
    glutDisplayFunc(myDraw);                // set up the callbacks
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);
    glutTimerFunc(20, myTimeOut, 0);        // timer in 20/1000 seconds
    ...
}
```

Most of these callback registrations simply pass the name of the desired user function to be called for the corresponding event. The one exception is glutTimeFunc whose arguments are the number of milliseconds to wait (an unsigned int), the user's callback function, and an integer identifier. The identifier is useful if there are multiple timer callbacks requested (for different times in the future), so the user can determine which one caused this particular event.

Callback Functions: What does a typical callback function do? This depends entirely on the application that you are designing. Some examples of general form of callback functions is shown below.

Examples of Callback Functions for System Events

```
void myDraw() {                          // called to display window
   // ...insert your drawing code here ...
   glutSwapBuffers();                    // make the new stuff visible
}
void myReshape(int w, int h) {           // called if reshaped
    windowWidth = w;                     // save new window size
    windowHeight = h;
    // ...may need to update the projection ...
    glutPostRedisplay();                 // request window redisplay
}
void myTimeOut(int id) {                 // called if timer event
    // ...advance the state of animation incrementally...
    glutTimerFunc(20, myTimeOut, 0);     // schedule next timer event
}
```

Note that the timer callback and the reshape callback both invoke the function glutPostRedisplay. This procedure informs OpenGL that the state of the scene has changed and should be redrawn (by calling your drawing procedure). This might be requested in other callbacks as well.

Examples of Callback Functions for User Input Events

```
                                         // called if mouse click
void myMouse(int b, int s, int x, int y) {
    switch (b) {                         // b indicates the button
        case GLUT_LEFT_BUTTON:
            if (s == GLUT_DOWN)          // button pressed
                // ...
            else if (s == GLUT_UP)       // button released
                // ...
            break;
        // ...                           // other button events
    }
}
                                         // called if keyboard key hit
void myKeyboard(unsigned char c, int x, int y) {
   switch (c) {                          // c is the key that is hit
      case 'q':                          // 'q' means quit
          exit(0);
          break;
      // ...                            // other keyboard events
   }
}
```

Note that each callback function is provided with information associated with the event. For example, a reshape event callback passes in the new window width and height. A mouse click callback passes in four arguments, which button was hit (b: left, middle, right), what the buttons new state is (s: up or down), the (x, y) coordinates of the mouse when it was clicked (in pixels). The various parameters used for b and s are described in Table 3. A keyboard event callback passes in the character that was hit and the current coordinates of the mouse. The timer event callback passes in the integer identifier, of the timer event which caused the callback. Note that each call to glutTimerFunc creates only one request for a timer event. (That is, you do not get automatic repetition of

timer events.) If you want to generate events on a regular basis, then insert a call to glutTimerFunc from within the callback function to generate the next one.

GLUT Parameter Name	Meaning
GLUT_LEFT_BUTTON	left mouse button
GLUT_MIDDLE_BUTTON	middle mouse button
GLUT_RIGHT_BUTTON	right mouse button
GLUT_DOWN	mouse button pressed down
GLUT_UP	mouse button released

Table 3: GLUT parameter names associated with mouse events. (Constants defined in glut.h)

Lecture 4: More about OpenGL and GLUT

Sources: See any standard reference on OpenGL or GLUT.

Basic Drawing: In the previous lecture, we showed how to create a window in GLUT, how to get user input, but we have not discussed how to get graphics to appear in the window. Here, we begin discussion of how to use OpenGL to draw objects.

Before being able to draw a scene, OpenGL needs to know the following information: what are the *objects* to be drawn, how is the image to be *projected* onto the window, and how *lighting* and *shading* are to be performed. To begin with, we will consider a very the simple case. There are only 2-dimensional objects, no lighting or shading. Also we will consider only relatively little user interaction.

Idealized Drawing Window: Because we generally do not have complete control over the window size, it is a good idea to think in terms of drawing on a rectangular *idealized drawing region*, whose size and shape are completely under our control. Then we will scale this region to fit within the actual graphics window on the display. There are many reasons for doing this. For example, if you design your program to work specifically for 400×400 window, but then the user resizes the window, what do you do? It is a much better idea to assume a window of arbitrary size. For example, you could define two variables w and h, that indicate the width and height of the window, respectively. The values w and h could be measured in whatever units are most natural to your application: pixels, inches, light years, microns, furlongs or fathoms.

OpenGL allows for the grahics window to be broken up into smaller rectangular subwindows, called *viewports*. We will then have OpenGL scale the image drawn in the idealized drawing region to fit within the viewport. The main advantage of this approach is that it is very easy to deal with changes in the window size.

We will consider a simple drawing routine for the picture shown in the figure. We assume that our idealized drawing region is a unit square over the real interval $[0, 1] \times [0, 1]$. (Throughout the course we will use the notation $[a, b]$ to denote the interval of real values z such that $a \leq z \leq b$. Hence, $[0, 1] \times [0, 1]$ is a unit square whose lower left corner is the origin.) This is illustrated in Fig. 5.

GLUT uses the convention that the origin is in the upper left corner and coordinates are given as integers. This makes sense for GLUT, because its principal job is to communicate with the window system, and most window systems use this convention. On the other hand, OpenGL uses the convention that coordinates are (generally) floating point values and the origin is in the lower left corner. Recalling the OpenGL goal is to provide us with an idealized drawing surface, this convention is mathematically more elegant.

The Display Callback: Recall that the *display callback function* is the function that is called whenever it is necessary to redraw the image, which arises for example:

- the initial creation of the window

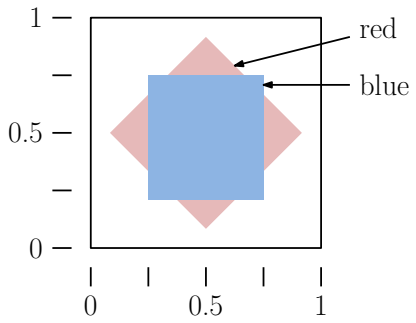


Fig. 5: Drawing produced by the simple display function.

- (possibly) whenever the window is uncovered by the removal of some overlapping window,
- whenever your program explicitly requests that it be redrawn, through the use of the function glutPostRedisplay

The display callback function for our program is shown in the following code block. We first erase the contents of the image window, then do our drawing, and finally swap buffers so that what we have drawn becomes visible. (Recall double buffering from the previous lecture.) This function first draws a red diamond and then (on top of this) it draws a blue rectangle. Let us assume double buffering is being performed, and so the last thing to do is invoke glutSwapBuffers() to make everything visible.

Sample Display Function

```
void myDisplay() {                          // display function
    glClear(GL_COLOR_BUFFER_BIT);           // clear the window

    glColor3f(1.0, 0.0, 0.0);               // set color to red
    glBegin(GL_POLYGON);                    // draw a diamond
        glVertex2f(0.90, 0.50);
        glVertex2f(0.50, 0.90);
        glVertex2f(0.10, 0.50);
        glVertex2f(0.50, 0.10);
    glEnd();

    glColor3f(0.0, 0.0, 1.0);               // set color to blue
    glRectf(0.25, 0.25, 0.75, 0.75);        // draw a rectangle

    glutSwapBuffers();                      // make it all visible
}
```

Clearing the Window: The command glClear() clears the window, by overwriting it with the background color. The background color is black by default, but generally it may be set by the call:

glClearColor(GLfloat Red, GLfloat Green, GLfloat Blue, GLfloat Alpha).

The type GLfloat is OpenGL's redefinition of the standard float. To be correct, you should use the approved OpenGL types (e.g. GLfloat, GLdouble, GLint) rather than the obvious counterparts (float, double, and int). Typically the GL types are the same as the corresponding native types, but not always.

Colors components are given as floats in the range from 0 to 1, from dark to light. Recall that the A (or α) value is used to control transparency. For opaque colors A is set to 1. Thus to set the background color to black, we would use glClearColor(0.0, 0.0, 0.0, 1.0), and to set it to blue use glClearColor(0.0, 0.0, 1.0, 1.0). (**Tip:**

When debugging your program, it is often a good idea to use an uncommon background color, like a random shade of pink, since black can arise as the result of many different bugs.) Since the background color is usually independent of drawing, the function glClearColor() is typically set in one of your initialization procedures, rather than in the drawing callback function.

Clearing the window involves resetting information within the drawing buffer. As we mentioned before, the drawing buffer may store different types of information. This includes color information, of course, but depth or distance information is used for hidden surface removal. Typically when the window is cleared, we want to clear everything. (Occasionally it is useful to achieve certain special effects by clearing only some of the buffers.) The glClear() command allows the user to select which buffers are to be cleared. In this case we only have color in the depth buffer, which is selected by the option GL_COLOR_BUFFER_BIT. If we had a depth buffer to be cleared it as well we could do this by combining these using a "bitwise or" operation:

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

Drawing Attributes: The OpenGL drawing commands describe the geometry of the object that you want to draw. More specifically, all OpenGL is based on drawing *convex polygons*, so it suffices to specify the *vertices* of the object to be drawn. The manner in which the object is displayed is determined by various *drawing attributes* (color, point size, line width, etc.).

The command glColor3f() sets the drawing color. The arguments are three GLfloat's, giving the R, G, and B components of the color. In this case, RGB $= (1, 0, 0)$ means pure red. Once set, the attribute applies to all subsequently defined objects, until it is set to some other value. Thus, we could set the color, draw three polygons with the color, then change it, and draw five polygons with the new color.

This call illustrates a common feature of many OpenGL commands, namely flexibility in argument types. The suffix "3f" means that three floating point arguments (actually GLfloat's) will be given. For example, glColor3d() takes three double (or GLdouble) arguments, glColor3ui() takes three unsigned int arguments, and so on. For floats and doubles, the arguments range from 0 (no intensity) to 1 (full intensity). For integer types (byte, short, int, long) the input is assumed to be in the range from 0 (no intensity) to its maximum possible positive value (full intensity).

But that is not all! The three argument versions assume RGB color. If we were using RGBA color instead, we would use glColor4d() variant instead. Here "4" signifies four arguments. (Recall that the A or alpha value is used for various effects, such an transparency. For standard (opaque) color we set $A = 1.0$.)

In some cases it is more convenient to store your colors in an array with three elements. The suffix "v" means that the argument is a vector. For example glColor3dv() expects a single argument, a vector containing three GLdouble's. (Note that this is a standard C/C++ style array, not the class vector from the C++ Standard Template Library.) Using C's convention that a vector is represented as a pointer to its first element, the corresponding argument type would be "const GLdouble$*$".

Whenever you look up the prototypes for OpenGL commands, you often see a long list with various suffixes to indicate the argument types. Some examples for glColor are shown in the following code block.

Drawing commands: OpenGL supports drawing of a number of different types of objects. The simplest is glRectf(), which draws a filled rectangle. All the others are complex objects consisting of a (generally) unpredictable number of elements. This is handled in OpenGL by the constructs glBegin(*mode*) and glEnd(). Between these two commands a list of vertices is given, which defines the object. The sort of object to be defined is determined by the *mode* argument of the glBegin() command. Some of the possible modes are illustrated in Fig. 6. For details on the semantics of the drawing methods, see the reference manuals.

Note that in the case of GL_POLYGON only *convex polygons* (internal angles less than 180 degrees) are supported. You must subdivide nonconvex polygons into convex pieces, and draw each convex piece separately.

```
glBegin(mode);
    glVertex(v0); glVertex(v1); ...
glEnd();
```

```
void glColor3d(GLdouble red, GLdouble green, GLdouble blue)
void glColor3f(GLfloat red, GLfloat green, GLfloat blue)
void glColor3i(GLint red, GLint green, GLint blue)
... (and forms for byte, short, unsigned byte and unsigned short) ...

void glColor4d(GLdouble red, GLdouble green, GLdouble blue, GLdouble alpha)
... (and 4-argument forms for all the other types) ...

void glColor3dv(const GLdouble *v)
... (and other 3- and 4-argument forms for all the other types) ...
```

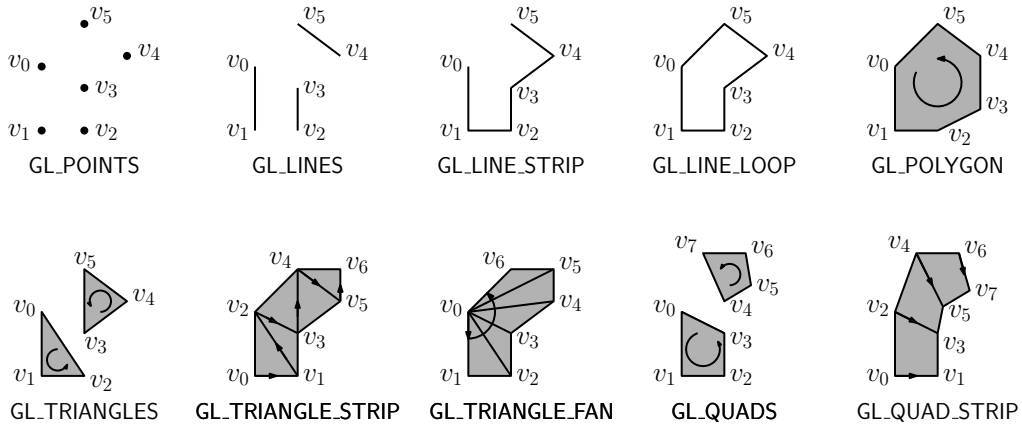


Fig. 6: Some OpenGL object definition modes. (It is a good idea to draw primitives using a consistent direction, say counterclockwise.)

In the example above we only defined the x- and y-coordinates of the vertices. How does OpenGL know whether our object is 2-dimensional or 3-dimensional? The answer is that it does not know. OpenGL represents all vertices as 3-dimensional coordinates internally. This may seem wasteful, but remember that OpenGL is designed primarily for 3-d graphics. If you do not specify the z-coordinate, then it simply sets the z-coordinate to 0.0. By the way, glRectf() always draws its rectangle on the $z = 0$ plane.

Between any glBegin()...glEnd() pair, there is a restricted set of OpenGL commands that may be given. This includes glVertex() and also other command attribute commands, such as glColor3f(). At first it may seem a bit strange that you can assign different colors to the different vertices of an object, but this is a very useful feature. Depending on the shading model, it allows you to produce shapes whose color *blends* smoothly from one end to the other.

There are a number of drawing attributes other than color, but until we start discussing lighting and 3-dimensional drawing, color is by far the most common attribute. (See the OpenGL documentation for other examples.)

After drawing the diamond, we change the color to blue, and then invoke glRectf() to draw a rectangle. This procedure takes four arguments, the (x, y) coordinates of any two opposite corners of the rectangle, in this case $(0.25, 0.25)$ and $(0.75, 0.75)$. (There are also versions of this command that takes double or int arguments, and vector arguments as well.) We could have drawn the rectangle by drawing a GL_POLYGON, but this form is easier to use.

Viewports: OpenGL does not assume that you are mapping your graphics to the entire window. Often it is desirable to subdivide the graphics window into a set of smaller subwindows and then draw separate pictures in each window. The subwindow into which the current graphics are being drawn is called a *viewport*. The viewport is

typically the entire display window, but it may generally be any rectangular subregion.

The size of the viewport depends on the dimensions of our window. Thus, every time the window is resized (and this includes when the window is created originally) we need to readjust the viewport to ensure proper transformation of the graphics. For example, in the typical case, where the graphics are drawn to the entire window, the reshape callback would contain the following call which resizes the viewport, whenever the window is resized.

Setting the Viewport in the Reshape Callback

```
void myReshape(int winWidth, int winHeight)       // reshape window
{
    ...
    glViewport (0, 0, winWidth, winHeight);       // reset the viewport
    ...
}
```

The other thing that might typically go in the myReshape() function would be a call to glutPostRedisplay(), since you will need to redraw your image after the window changes size.

The general form of the command is

glViewport(GLint x, GLint y, GLsizei width, GLsizei height),

where (x, y) are the pixel coordinates of the *lower-left corner* of the viewport, as defined relative to the lower-left corner of the window, and *width* and *height* are the width and height of the viewport in pixels.

For example, suppose you had a $w \times h$ window, which you wanted to split in half by a vertical line to produce two different drawings. This is presented in the following code block.

Setting up viewports for a split screen

```
glClear(GL_COLOR_BUFFER_BIT);       // clear the window
glViewport (0, 0, w/2, h);          // set viewport to left half
  // ...drawing commands for the left half of window
glViewport (w/2, 0, w/2, h);        // set viewport to right half
  // ...drawing commands for the right half of window
glutSwapBuffers();                  // swap buffers
```

Projection Transformation: In the simple drawing procedure, we said that we were assuming that the "idealized" drawing area was a unit square over the interval $[0, 1]$ with the origin in the lower left corner. The transformation that maps the idealized drawing region (in 2- or 3-dimensions) to the window is called the *projection*. We did this for convenience, since otherwise we would need to explicitly scale all of our coordinates whenever the user changes the size of the graphics window.

However, we need to inform OpenGL of where our "idealized" drawing area is so that OpenGL can map it to our viewport. This mapping is performed by a transformation matrix called the *projection matrix*, which OpenGL maintains internally. (In future lectures, we will discuss OpenGL's transformation mechanism in greater detail. In the mean time some of this may seem a bit arcane.)

Since matrices are often cumbersome to work with, OpenGL provides a number of relatively simple and natural ways of defining this matrix. For our 2-dimensional example, we will do this by simply informing OpenGL of the rectangular region of two dimensional space that makes up our idealized drawing region. This is handled by the command

gluOrtho2D(left, right, bottom, top).

First note that the prefix is "glu" and not "gl", because this procedure is provided by the GLU library. Also, note that the "2D" designator in this case stands for "2-dimensional." (In particular, it does *not* indicate the argument types, as with, say, glColor3f()).

All arguments are of type GLdouble. The arguments specify the x-coordinates (*left* and *right*) and the y-coordinates (*bottom* and *top*) of the rectangle into which we will be drawing. Any drawing that we do outside of this region will automatically be clipped away by OpenGL. The code to set the projection is given below.

Setting a Two-Dimensional Projection

```
glMatrixMode(GL_PROJECTION);          // set projection matrix mode
glLoadIdentity();                     // initialize to identity
gluOrtho2D(0.0, 1.0, 0.0, 1.0);       // map unit square to viewport
glMatrixMode(GL_MODELVIEW);           // set matrix mode back to default
```

The first command tells OpenGL that we are modifying the projection transformation. (OpenGL maintains three different types of transformations, as we will see later.) Most of the commands that manipulate these matrices do so by multiplying some matrix times the current matrix. Thus, we initialize the current matrix to the identity, which is done by glLoadIdentity(). This code usually appears in some initialization procedure or possibly in the reshape callback.

Where does this code fragment go? It depends on whether the projection will change or not. If we make the simple assumption that are drawing will always be done relative to the $[0, 1]^2$ unit square, then this code can go in some initialization procedure. If our program decides to change the drawing area (for example, growing the drawing area when the window is increased in size) then we would need to repeat the call whenever the projection changes.

At first viewports and projections may seem confusing. Remember that the viewport is a rectangle within the actual graphics window on your display, where you graphics will appear. The projection defined by gluOrtho2D() simply defines a rectangle in some "ideal" coordinate system, which you will use to specify the coordinates of your objects. It is the job of OpenGL to map everything that is drawn in your ideal window to the actual viewport on your screen. This is illustrated in Fig. 7.

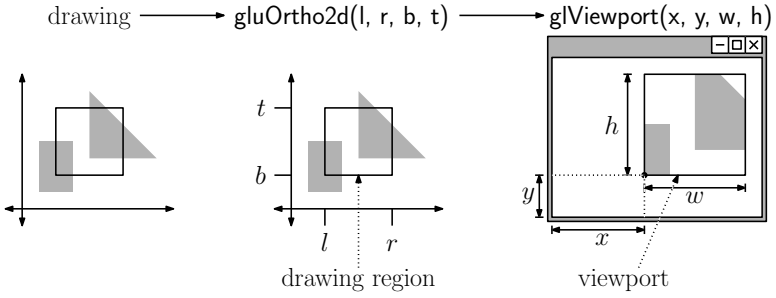


Fig. 7: Projection and viewport transformations.

The complete program is shown in the code block below.

Lecture 5: Drawing in OpenGL: Transformations

Sources: See any standard reference on OpenGL or GLUT.

Complex Drawing in OpenGL: So far we have discussed how to draw simple 2-dimensional objects using OpenGL. Suppose that we want to draw more complex scenes. For example, we want to draw objects that move and rotate

```
#include <cstdlib>                      // standard definitions
#include <iostream>                     // C++ I/O

#include <GL/glut.h>                    // GLUT (also loads gl.h and glu.h)

void myReshape(int w, int h) {                  // window is reshaped
    glViewport (0, 0, w, h);                    // update the viewport
    glMatrixMode(GL_PROJECTION);                // update projection
    glLoadIdentity();
    gluOrtho2D(0.0, 1.0, 0.0, 1.0);             // map unit square to viewport
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();                        // request redisplay
}

void myDisplay(void) {                          // (re)display callback
    glClearColor(0.5, 0.5, 0.5, 1.0);           // background is gray
    glClear(GL_COLOR_BUFFER_BIT);               // clear the window
    glColor3f(1.0, 0.0, 0.0);                   // set color to red
    glBegin(GL_POLYGON);                        // draw the diamond
        glVertex2f(0.90, 0.50);
        glVertex2f(0.50, 0.90);
        glVertex2f(0.10, 0.50);
        glVertex2f(0.50, 0.10);
    glEnd();
    glColor3f(0.0, 0.0, 1.0);                   // set color to blue
    glRectf(0.25, 0.25, 0.75, 0.75);            // draw the rectangle
    glutSwapBuffers();                          // swap buffers
}

int main(int argc, char** argv) {               // main program
    glutInit(&argc, argv);                      // OpenGL initializations
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);// double buffering and RGB
    glutInitWindowSize(400, 400);               // create a 400x400 window
    glutInitWindowPosition(0, 0);               // ...in the upper left
    glutCreateWindow(argv[0]);                  // create the window

    glutDisplayFunc(myDisplay);                 // setup callbacks
    glutReshapeFunc(myReshape);
    glutMainLoop();                             // start it running
    return 0;                                   // ANSI C expects this
}
```

or to change the projection. We could do this by computing (ourselves) the coordinates of the transformed vertices. However, this would be inconvenient for us. It would also be inefficient. OpenGL provides methods for downloading large geometric specifications directly to the GPU. However, if the coordinates of these object were changed with each display cycle, this would negate the benefit of loading them just once.

For this reason, OpenGL provides tools to handle transformations and to apply these transformations to the objects we wish to draw.

Affine Transformations: Linear and affine transformations are central to computer graphics. Recall from your linear algebra class that a *linear transformation* is a mapping in a vector space that preserves linear combinations. Such transformations include rotations, scalings, shearings (which stretch rectangles into parallelograms), and combinations thereof.

Linear transformations are limited in that they cannot represent translation. The origin is always mapped to the origin. If you combine linear transformations with translation, you get a broader class of transformations called *affine transformations*. Affine transformations have a number of nice properties:

Preserves linearity: Lines are mapped to lines

Preserves parallelism: Parallel lines remain parallel

Preserves affine combinations: This is harder to explain, but as an example, if p and q are two points and m is their midpoint, and T is an affine transformation, then the midpoint of $T(p)$ and $T(q)$ is $T(m)$.

Points in affine space are represented in *homogeneous coordinates*. This is done by adding an extra coordinate to the end, which is set to 1. For example, in two dimensional space, the point with coordinates (x, y) would be expressed in homogeneous coordinates as the 3-element vector $[x, y, 1]$. Given in this form, any affine transformation can be expressed as the product of such a vector and a 3×3 matrix. In general, a point in standard d-dimensional space can be expressed as a $(d+1)$-dimensional homogeneous vector, by appending a 1 to the end. Affine transformations can be expressed as the multiplication of this vector times a $(d+1) \times (d+1)$ matrix.

How are Affine Transformations Used? Affine transformations have many uses in computer graphics and game programming in general.

Moving Objects: As needed in animations, physical simulations, etc.

Change of Coordinates: This is used when objects that are stored relative to one coordinate frame are to be accessed in a different coordinate frame. One important case of this is that of mapping objects stored in a standard coordinate system to a coordinate system that is associated with the camera (or viewer).

Parallel Camera Projection: Projection is the task of mapping three-dimensional objects onto a two-dimensional image plane. The most common type of projections are called *perspective projections*, where the image rays converge at a fixed focal point point in space. These are *not* affine transformations. However, if the image rays are parallel to each other (they converge at a focal point at infinity), the resulting transformation is affine. An important special case are *orthogonal projections*, where the projection direction is parallel to one of the coordinate axes.

Applying Textures: This is useful when textures are mapped from an image onto an object surface as part of texture mapping process.

OpenGL and Transformations: OpenGL has a very particular model for how transformations are performed. Recall that when drawing, it was convenient for us to first define the drawing attributes (such as color) and then draw a number of objects using that attribute. OpenGL uses much the same model with transformations. You specify a transformation *first*, and then this transformation is automatically applied to every object that is drawn *afterwards*, until the transformation is set again. It is important to keep this in mind, because it implies that you must always set the transformation prior to issuing drawing commands.

Because transformations are used for different purposes, OpenGL maintains three sets of matrices for performing various transformation operations. These are:

Modelview matrix: (GL_MODELVIEW) Used for transforming objects in the scene and for changing the coordinates into a form that is easier for OpenGL to deal with. (It is used for the first two tasks above, moving objects and converting between coordinate systems).

Projection matrix: (GL_PROJECTION) Handles both parallel and perspective projections. (Used for the third task above.)

Texture matrix: (GL_TEXTURE) This is used in specifying how textures are mapped onto objects. (Used for the last task above.)

We will discuss the texture matrix later in the semester, when we talk about texture mapping. There is one more transformation that is not handled by these matrices. This is the transformation that maps the viewport to the display. It is set by glViewport().

Understanding how OpenGL maintains and manipulates transformations through these matrices is central to understanding how OpenGL and other modern immediate-mode rendering systems (such as DirectX) work.

Matrix Stacks: For each matrix type, OpenGL maintains a *stack* of matrices. The *current matrix* is the one on the top of the stack. It is the matrix that is being applied at any given time. The stack mechanism allows you to save the current matrix (by pushing the stack down) and restoring it later (by popping the stack). We will discuss the entire process of implementing affine and projection transformations later in the semester. For now, we'll give just basic information on OpenGL's approach to handling matrices and transformations.

OpenGL has a number of commands for handling matrices. In order to know which matrix (Modelview, Projection, or Texture) to which an operation applies, you can set the current *matrix mode*. This is done with the following command

glMatrixMode(⟨*mode*⟩);

where ⟨*mode*⟩ is either GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE. The default is GL_MODELVIEW.

GL_MODELVIEW is by far the most common mode, the convention in OpenGL programs is to assume that you are always in this mode. If you want to modify the mode for some reason, you first change the mode to the desired mode (GL_PROJECTION or GL_TEXTURE), perform whatever operations you want, and then immediately change the mode back to GL_MODELVIEW.

Once the matrix mode is set, you can perform various operations to the stack. OpenGL has an unintuitive way of handling the stack. Note that most operations below (except glPushMatrix()) alter the contents of the matrix at the top of the stack.

glLoadIdentity(): Sets the current matrix to the identity matrix.

glLoadMatrix*(M): Loads (copies) a given matrix over the current matrix. (The '*' can be either 'f' or 'd' depending on whether the elements of M are GLfloat or GLdouble, respectively.)

glMultMatrix*(M): Post-multiplies the current matrix by a given matrix and replaces the current matrix with this result. Thus, if C is the current matrix on top of the stack, it will be replaced with the matrix product $C \cdot M$. (As above, the '*' can be either 'f' or 'd' depending on M.)

glPushMatrix(): Pushes a copy of the current matrix on top the stack. (Thus the stack now has two copies of the top matrix.)

glPopMatrix(): Pops the current matrix off the stack.

Warning: OpenGL assumes that all matrices are 4×4 homogeneous matrices, stored in column-major order. That is, a matrix is presented as an array of 16 values, where the first four values give column 0 (for x), then

column 1 (for y), then column 2 (for z), and finally column 3 (for the homogeneous coordinate, usually called w). For example, given a matrix M and vector v, OpenGL assumes the following representation:

$$M \cdot v = \begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix} \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

An example is shown in Fig. 8. We will discuss how matrices like M are presented to OpenGL later in the semester. There are a number of other matrix operations, which we will also discuss later.

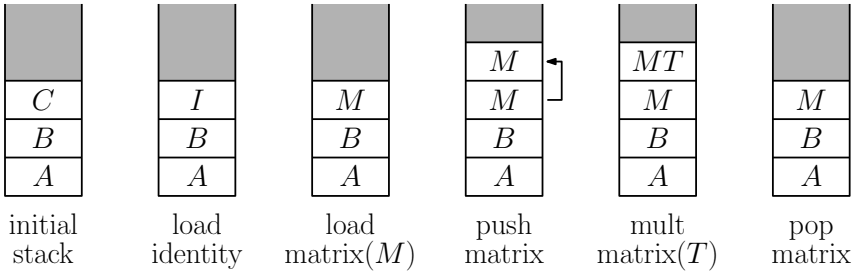


Fig. 8: Matrix stack operations.

Automatic Evaluation and the Transformation Pipeline: Now that we have described the matrix stack, the next question is how do we apply the matrix to some point that we want to transform? Understanding the answer is critical to understanding how OpenGL (and actually display processors) work. The answer is that it happens *automatically*. In particular, *every* vertex (and hence virtually every geometric object that is drawn) is passed through a series of matrices, as shown in Fig. 9. This may seem rather inflexible, but it is because of the simple uniformity of sending every vertex through this transformation sequence that makes it possible for GPUs to run so fast. As mentioned above, these transformations behave much like drawing attributes—you set them, do some drawing, alter them, do more drawing, etc.

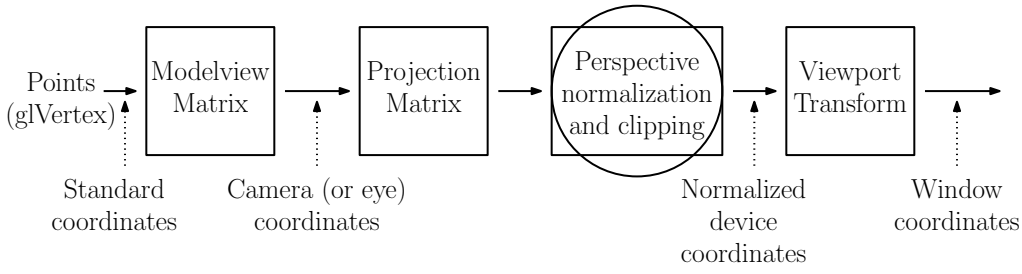


Fig. 9: Transformation pipeline.

A second important thing to understand is that OpenGL's transformations do not alter the state of the objects you are drawing. They simply modify things before they get drawn. For example, suppose that you draw a unit square ($U = [0, 1] \times [0, 1]$) and pass it through a matrix that scales it by a factor of 5. The square U itself has not changed; it is still a unit square. If you wanted to change the actual representation of U to be a 5×5 square, then you need to perform your own modification of U's representation.

You might ask, "what if I do *not* want the current transformation to be applied to some object?" The answer is, "tough luck." There are no exceptions to this rule (other than commands that act directly on the viewport). If you do not want a transformation to be applied, then to achieve this, you load an identity matrix on the top of the transformation stack, then do your (untransformed) drawing, and finally pop the stack.

Example: Rotating a Rectangle (first attempt): The Modelview matrix is useful for applying transformations to objects, which would otherwise require you to perform your own linear algebra. Suppose that rather than drawing a rectangle that is aligned with the coordinate axes, you want to draw a rectangle that is rotated by 20 degrees (counterclockwise) and centered at some point (x, y). The desired result is shown in Fig. 10. Of course, as mentioned above, you could compute the rotated coordinates of the vertices yourself (using the appropriate trigonometric functions), but OpenGL provides a way of doing this transformation more easily.

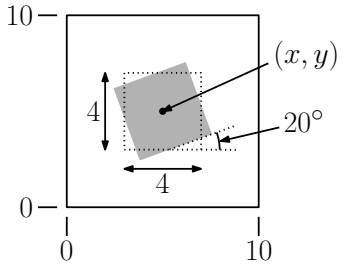


Fig. 10: Desired drawing. (Rotated rectangle is shaded).

Suppose that we are drawing within the square, $0 \leq x, y \leq 10$, and we have a 4×4 sized rectangle to be drawn centered at location (x, y). We could draw an unrotated rectangle with the following command:

```
glRectf(x - 2, y - 2, x + 2, y + 2);
```

Formally, the arguments should be of type GLfloat (2.0f rather than 2), but we will let the compiler cast the integer constants to floating point values for us.

Now let us draw a rotated rectangle. Let us assume that the matrix mode is GL_MODELVIEW (this is the default). Generally, there will be some existing transformation (call it M) currently present in the Modelview matrix. This usually represents some more global transformation, which is to be applied on top of our rotation. For this reason, we will compose our rotation transformation with this existing transformation.

Because the OpenGL rotation function destroys the contents of the Modelview matrix, we will begin by saving it, by using the command glPushMatrix(). Saving the Modelview matrix in this manner is not always required, but it is considered good form. Then we will compose the current matrix M with an appropriate rotation matrix R. Then we draw the rectangle (in upright form). Since all points are transformed by the Modelview matrix prior to projection, this will have the effect of rotating our rectangle. Finally, we will pop off this matrix (so future drawing is not rotated).

To perform the rotation, we will use the command glRotatef(ang, x, y, z). All arguments are GLfloat's. (Or, recalling OpenGL's naming convention, we could use glRotated() which takes GLdouble arguments.) This command constructs a matrix that performs a rotation in 3-dimensional space counterclockwise by angle *ang* degrees, about the vector (x, y, z). It then *composes* (or multiplies) this matrix with the current Modelview matrix. In our case the angle is 20 degrees. To achieve a rotation in the (x, y) plane the vector of rotation would be the z-unit vector, $(0, 0, 1)$. Here is how the code might look (but beware, this conceals a subtle error).

Drawing an Rotated Rectangle (First Attempt)

```
glPushMatrix();                 // save the current matrix
  glRotatef(20, 0, 0, 1);       // rotate by 20 degrees CCW
  glRectf(x-2, y-2, x+2, y+2);  // draw the rectangle
glPopMatrix();                  // restore the old matrix
```

The order of the rotation relative to the drawing command may seem confusing at first. You might think, "Shouldn't we draw the rectangle first and then rotate it?". The key is to remember that whenever you draw

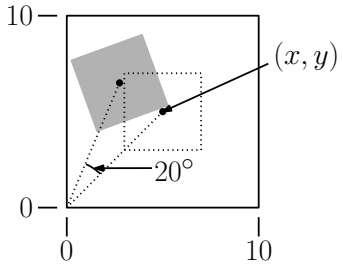


Fig. 11: The actual drawing produced by the previous example. (Rotated rectangle is shaded).

(using glRectf() or glBegin()...glEnd()), the points are automatically transformed using the current Modelview matrix. So, in order to do the rotation, we must first modify the Modelview matrix, then draw the rectangle. The rectangle will be automatically transformed into its rotated state. Popping the matrix at the end is important, otherwise future drawing requests would also be subject to the same rotation.

Unfortunately, something is wrong with this example given above. What is it? The answer is that the rotation is performed *about the origin* of the coordinate system, not about the center of the rectangle as we want.

Correct Rotation (Through the Looking Glass): Fortunately, there is an easy fix. Conceptually, we will draw the rectangle centered at the origin, then rotate it by 20 degrees, and finally *translate* (or move) it by the vector (x, y). To do this, we will need to use the command glTranslatef(x, y, z) (see Fig. 12). All three arguments are GLfloat's. (And there is version with GLdouble arguments.) This command creates a matrix which performs a translation by the vector (x, y, z), and then composes (or multiplies) it with the current matrix. Recalling that all 2-dimensional graphics occurs in the $z = 0$ plane, the desired translation vector is $(x, y, 0)$.

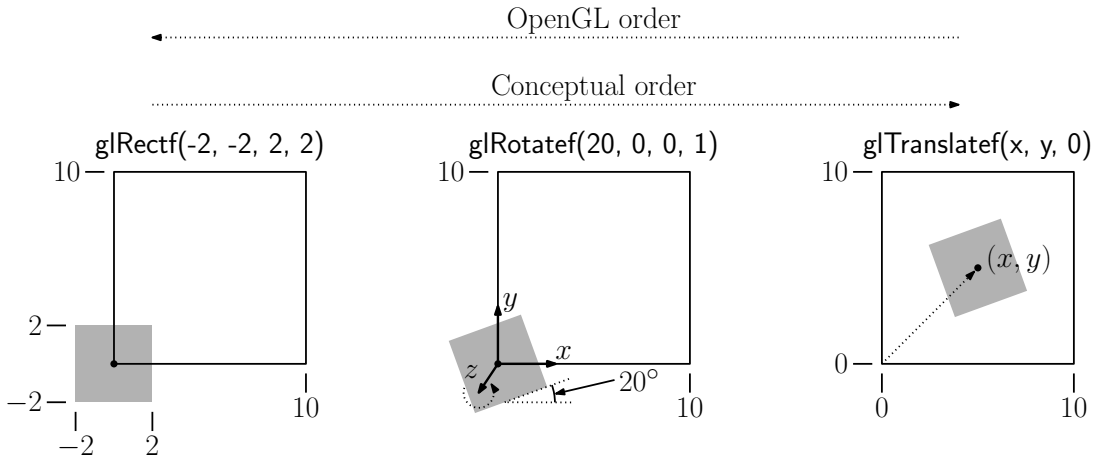


Fig. 12: Final drawing/transformation sequence.

So the conceptual order is (1) draw, (2) rotate, (3) translate. But remember that you need to set up the transformation matrix *before* you do any drawing. That is, if $\vec{v}$ represents a vertex of the rectangle, and R is the rotation matrix and T is the translation matrix, and M is the current Modelview matrix, then we want to compute the product

$$M(T(R(\vec{v}))) = M \cdot T \cdot R \cdot \vec{v}.$$

Since M is on the top of the stack, we need to first apply translation (T) to M, and then apply rotation (R) to the result, and then do the drawing ($\vec{v}$). Note that the order of application is the exact *reverse* from the conceptual order. This may seems confusing (and it is), so remember the following rule.

Drawing/Transformation Order in OpenGL's

First, conceptualize your intent by drawing about the origin and then applying the appropriate transformations to map your object to its desired location. Then implement this by applying transformations in *reverse order*, and do your drawing. It is always a good idea to enclose everything in a push-matrix and pop-matrix pair.

Although this may seem backwards, it is the way in which almost all object transformations are performed in OpenGL:

(1) Push the matrix stack,

(2) Apply (i.e., multiply) all the desired transformation matrices with the current matrix, but *in the reverse order* from which you would like them to be applied to your object,

(3) Draw your object (the transformations will be applied automatically), and

(4) Pop the matrix stack.

The final and correct fragment of code for the rotation is shown in the code block below.

Drawing an Rotated Rectangle (Correct)

```
glPushMatrix();                 // save the current matrix (M)
  glTranslatef(x, y, 0);        // apply translation (T)
  glRotatef(20, 0, 0, 1);       // apply rotation (R)
  glRectf(-2, -2, 2, 2);        // draw rectangle at the origin
glPopMatrix();                  // restore the old matrix (M)
```

Projection Revisited: Last time we discussed the use of gluOrtho2D() for doing simple 2-dimensional projection. This call does not really do any projection. Rather, it computes the desired projection transformation and multiplies it times whatever is on top of the current matrix stack. So, to use this we need to do a few things. First, set the matrix mode to GL_PROJECTION, load an identity matrix (just for safety), and the call gluOrtho2D(). Because of the convention that the Modelview mode is the default, we will set the mode back when we are done.

Two Dimensional Projection

```
glMatrixMode(GL_PROJECTION);            // set projection matrix
glLoadIdentity();                       // initialize to identity
gluOrtho2D(left, right, bottom top);    // set the drawing area
glMatrixMode(GL_MODELVIEW);             // restore Modelview mode
```

If you only set the projection once, then initializing the matrix to the identity is typically redundant (since this is the default value), but it is a good idea to make a habit of loading the identity for safety. If the projection does not change throughout the execution of our program, and so we include this code in our initializations. It might be put in the reshape callback if reshaping the window alters the projection.

How is it done (Optional): How does gluOrtho2D() and glViewport() set up the desired transformation from the idealized drawing window to the viewport? Well, actually OpenGL does this in two steps, first mapping from the window to canonical 2×2 window centered about the origin, and then mapping this canonical window to the viewport. The reason for this intermediate mapping is that the clipping algorithms are designed to operate on this fixed sized window (recall the figure given earlier). The intermediate coordinates are often called *normalized device coordinates*.

As an exercise in deriving linear transformations, let us consider doing this all in one shot. Let W denote the idealized drawing window and let V denote the viewport. Let w_r, w_l, w_b, and w_t denote the left, right, bottom and top of the window. Define v_r, v_l, v_b, and v_t similarly for the viewport. We wish to derive a linear

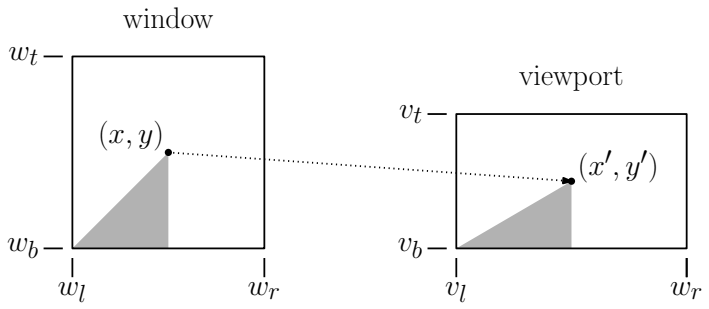


Fig. 13: Window to Viewport transformation.

transformation that maps a point (x, y) in window coordinates to a point (x', y') in viewport coordinates. See Fig. 13.

Let $f(x, y)$ denote the desired transformation. Since the function is linear, and it operates on x and y independently, we have

$$(x', y') = f(x, y) = (s_x x + t_x, s_y y + t_y),$$

where s_x, t_x, s_y and t_y, depend on the window and viewport coordinates. Let's derive what s_x and t_x are using simultaneous equations. We know that the x-coordinates for the left and right sides of the window (w_l and w_r) should map to the left and right sides of the viewport (v_l and v_r). Thus we have

$$s_x w_l + t_x = v_l \qquad \text{and} \qquad s_x w_r + t_x = v_r.$$

We can solve these equations simultaneously. By subtracting them to eliminate t_x we have

$$s_x = \frac{v_r - v_l}{w_r - w_l}.$$

Plugging this back into to either equation and solving for t_x we have

$$t_x \;=\; v_l - s_x w_l \;=\; v_l - \frac{v_r - v_l}{w_r - w_l} w_l \;=\; \frac{v_l w_r - v_r w_l}{w_r - w_l}.$$

A similar derivation for s_y and t_y yields

$$s_y = \frac{v_t - v_b}{w_t - w_b} \qquad t_y = \frac{v_b w_t - v_t w_b}{w_t - w_b}.$$

These four formulas give the desired final transformation.

$$f(x, y) = \left(\frac{(v_r - v_l)x + (v_l w_r - v_r w_l)}{w_r - w_l}, \frac{(v_t - v_b)y + (v_b w_t - v_t w_b)}{w_t - w_b} \right).$$

This can be expressed in matrix form as

$$\begin{pmatrix} \frac{v_r - v_l}{w_r - w_l} & 0 & \frac{v_l w_r - v_r w_l}{w_r - w_l} \\ 0 & \frac{v_t - v_b}{w_t - w_b} & \frac{v_b w_t - v_t w_b}{w_t - w_b} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

which is essentially what OpenGL stores internally.

Lecture 6: 3-d Viewing and Projections

Sources: See any standard reference on OpenGL or GLUT.

Viewing in OpenGL: Today we will discuss how viewing and perspective transformations are handled for 3-dimensional scenes. In OpenGL, and most similar graphics systems, the process involves the following basic steps, of which the perspective transformation is just one component. We assume that all objects are initially represented relative to a standard 3-dimensional coordinate frame, in what are called *world coordinates*.

Modelview transformation: Maps objects (actually vertices) from their world-coordinate representation to one that is centered around the viewer. The resulting coordinates are variously called *camera coordinates*, *view coordinates*, or *eye coordinates*. (Specified by the OpenGL command gluLookAt.)

Projection: This projects points in 3-dimensional eye-coordinates to points on a plane called the *image plane*. This projection process consists of three separate parts: the projection transformation (affine part), clipping, and perspective normalization. Each will be discussed below. The output coordinates are called *normalized device coordinates*. (Specified by the OpenGL commands such as gluOrtho2D, glOrtho, glFrustum, and gluPerspective.)

Mapping to the viewport: Convert the point from these idealized normalized device coordinates to the viewport. The coordinates are called *window coordinates* or *viewport coordinates*. (Specified by the OpenGL command glViewport.)

We have ignored a number of issues, such as lighting and hidden surface removal. These will be considered separately later. The process is illustrated in Fig. 14. We have already discussed the viewport transformation, so it suffices to discuss the first two transformations.

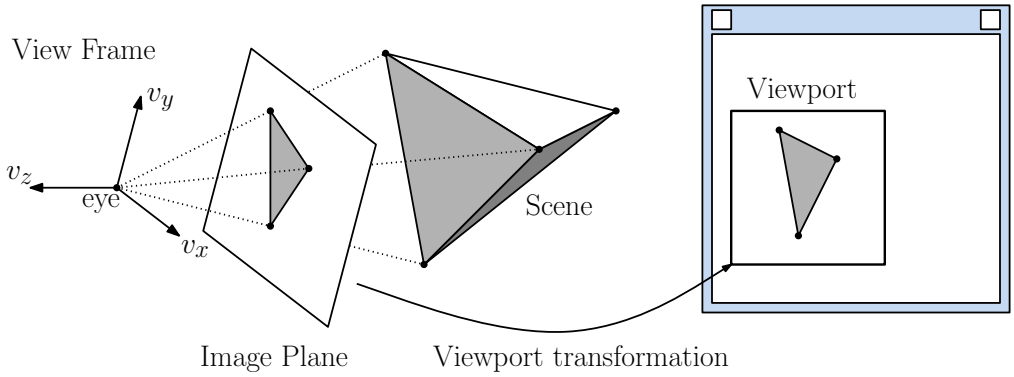


Fig. 14: OpenGL Viewing Process.

Converting to Camera-Centered Coordinate System: As we shall see below, the perspective transformation is simplest when the *center of projection*, the location of the viewer, is the origin and the *image plane* (sometimes called the *projection plane* or *view plane*), onto which the image is projected, is orthogonal to one of the axes, say the z-axis. Let us call these *camera coordinates*. However the user represents points relative to a coordinate system that is convenient for his/her purposes. Let us call these *world coordinates*. This suggests that, prior to performing the perspective transformation, we perform a change of coordinate transformation to map points from world coordinates to camera coordinates.

In OpenGL, there is a nice utility for doing this. The procedure gluLookAt generates the desired transformation to perform this change of coordinates and multiplies it times the transformation at the top of the current transformation stack. (Recall OpenGL's transformation structure from the previous lecture on OpenGL transformations.) This should be done in Modelview mode.

Conceptually, this change of coordinates is performed *last*, after all other Modelview transformations are performed, and immediately before the projection. By the "reverse rule" of OpenGL transformations, this implies that this change of coordinates transformation should be the *first* transformation on the Modelview transformation matrix stack. Thus, it is almost always preceded by loading the identity matrix. Here is the typical calling sequence. This should be called when the camera position is set initially, and whenever the camera is (conceptually) repositioned in space.

Typical Structure of Redisplay Callback

```
void myDisplay() {
                                        // clear the buffer
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();                     // start fresh
                                        // set up camera frame
  gluLookAt(eyeX, eyeY, eyeZ,  atX, atY, atZ,  upX, upY, upZ);
  myWorld.draw();                       // draw your scene
  glutSwapBuffers();                    // make it all appear
}
```

The arguments are all of type GLdouble. The arguments consist of the coordinates of two points and vector, in the standard coordinate system. The point *eye* $= (e_x, e_y, e_z)^\mathsf{T}$ is the *viewpoint*, that is the location of they viewer (or the camera). To indicate the direction that the camera is pointed, a central point at which the camera is directed is given by *at* $= (a_x, a_y, a_z)^\mathsf{T}$. The "at" point is significant only in that it defines the *viewing vector*, which indicates the direction that the viewer is facing. It is defined to be *at* $-$ *eye* (see Fig. 15).

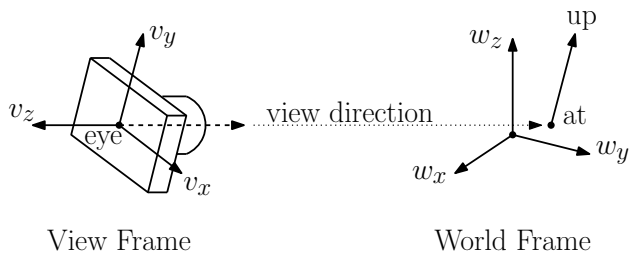


Fig. 15: The world frame, parameters to gluLookAt, and the camera frame.

These points define the position and direction of the camera, but the camera is still free to rotate about the viewing direction vector. To fix last degree of freedom, the vector $\overrightarrow{up} = (u_x, u_y, u_z)^\mathsf{T}$ provides the direction that is "up" relative to the camera. Under typical circumstances, this would just be a vector pointing straight up (which might be $(0, 0, 1)^\mathsf{T}$ in your world coordinate system). In some cases (e.g. in a flight simulator, when the plane banks to one side) you might want to have this vector pointing in some other direction (e.g., up relative to the pilot's orientation). This vector *need not* be perpendicular to the viewing vector. However, it cannot be parallel to the viewing direction vector.

The Camera Frame: OpenGL uses the arguments to gluLookAt to construct a coordinate frame centered at the viewer. The x- and y-axes are directed to the right and up, respectively, relative to the viewer. It might seem natural that the z-axes be directed in the direction that the viewer is facing, but this is not a good idea.

To see why, we need to discuss the distinction between right-handed and left-handed coordinate systems. Consider an orthonormal coordinate system with basis vectors $\vec{v}_x$, $\vec{v}_y$ and $\vec{v}_z$. This system is said to be *right-handed* if $\vec{v}_x \times \vec{v}_y = \vec{v}_z$ (see Fig. 16), and left-handed otherwise ($\vec{v}_x \times \vec{v}_y = -\vec{v}_z$). Right-handed coordinate systems are used by default throughout mathematics. (Otherwise computation of orientations is all screwed up.) Given that the x- and y-axes are directed right and up relative to the viewer, if the z-axis were to point in the direction that the viewer is facing, this would result in left-handed coordinate system. The designers of OpenGL wisely

decided to stick to a right-handed coordinate system, which requires that the z-axes is directed opposite to the viewing direction.

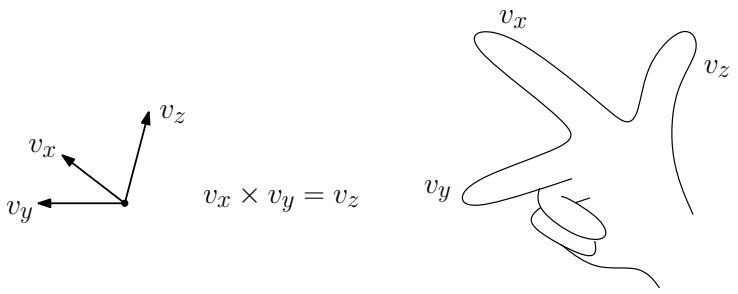


Fig. 16: Right-handed coordinate system.

Building the Camera Frame: (Optional) How does OpenGL implement this change of coordinate transformation? In order to answer this, you will need to recall a bit of linear algebra (assuming you studied linear algebra). We need to know two things. First, given two points p and q, the difference $p-q$ yields a vector that points from q to p. Given a vector $\vec{v}$, we define normalize to be a function that returns a vector pointing in the same direction as $\vec{v}$, but is of unit length. The Euclidean length of a vector $\vec{v} = (v_x, v_y, v_z)$ is defined to be $\|v\| = \sqrt{v_x^2 + v_y^2 + v_z^2}$. Therefore, we can define normalize$(v) = v/\|v\|$. Finally, recall the *cross product* operation. Given two vectors $\vec{u}$ and $\vec{v}$, their cross product, $\vec{u} \times \vec{v}$ is a vector that is orthogonal to both of these vectors. (It length is equal to the area of the parallelogram spanner between these two vectors, but we won't need this.) The direction of the cross product is given by *right-hand rule*. A *coordinate frame* in three dimensional space consists of an *origin point* and three *axis vectors*. The most useful frame is *orthonormal*, which means that each of the three axis vectors is of unit length and they are pairwise orthogonal to each other.

Given this background, computing the camera frame turns out to be an exercise in linear algebra. We want to construct an orthonormal frame whose origin is the point *eye*, whose $-z$-basis vector is parallel to the view vector, and such that the $\overrightarrow{up}$ vector projects to the up direction in the final projection. (This is illustrated in the Fig. 17, where the x-axis is pointing outwards from the page.)

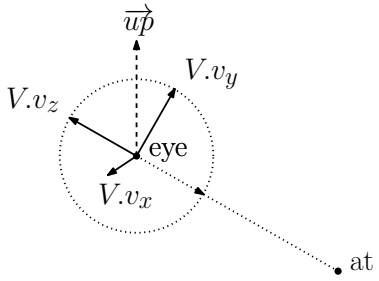


Fig. 17: The camera frame.

Let $V = (V.\vec{v}_x, V.\vec{v}_y, V.\vec{v}_z, V.o)^\mathsf{T}$ denote this frame, where $(\vec{v}_x, \vec{v}_y, \vec{v}_z)^\mathsf{T}$ are the three axis vectors for the frame and o is the origin. Clearly $V.o = eye$. As mentioned earlier, the view vector $\overrightarrow{view}$ is directed from *eye* to *at*. The z-basis vector is the normalized negation of this vector.

$$\overrightarrow{view} \;=\; \text{normalize}(at - eye) \qquad \text{and} \qquad V.\vec{v}_z \;=\; -\overrightarrow{view}$$

Next, we want to select the x-axis vector for our camera frame. It should be orthogonal to the viewing direction, it should be orthogonal to the up vector, and it should be directed to the camera's right. Recall that the cross

product will produce a vector that is orthogonal to any pair of vectors, and directed according to the right-hand rule. Also, we want this vector to have unit length. Thus we choose

$$V.\vec{v}_x = \text{normalize}(\overrightarrow{view} \times \overrightarrow{up}).$$

The result of the cross product must be a nonzero vector. This is why we require that the view direction and up vector are not parallel to each other. We have two out of three vectors for our frame. We can extract the last one by taking a cross product of the first two:

$$V.\vec{v}_y = (V.\vec{v}_z \times V.\vec{v}_x).$$

There is no need to normalize this vector, because it is the cross product of two orthogonal vectors, each of unit length. Once the frame has been computed, it is an exercise in linear algebra to compute a matrix that performs this change-of-coordinates transformation. We'll skip the messy details, but if you are interested, most books on computer graphics will explain how it is done.

Projections: The next part of the process involves performing the projection. Projections fall into two basic groups, *parallel projections*, in which the lines of projection are parallel to one another, and *perspective projection*, in which the lines of projection converge a point.

In spite of their superficial similarities, parallel and perspective projections behave quite differently with respect to geometry. Parallel projections are *affine transformations*, while perspective projections are not. (In particular, perspective projections do not preserve parallelism, as is evidenced by a perspective view of a pair of straight train tracks, which appear to converge at the horizon.) Because parallel projections are rarely used, we will skip them and consider perspective projections only.

OpenGL's Perspective Projection: OpenGL provides a couple of ways to specify the perspective projection. The most general method is through glFrustum. We will discuss a simpler method called gluPerspective, which suffices for almost all cases that arise in practice. In particular, this simpler procedure assumes that the viewing window is centered about the view direction vector (the negative z-axis), whereas glFrustum does not.

Consider the following viewing model. In front of his eye, the user holds rectangular window, centered on the view direction, onto which the image is to be projected. The viewer sees any object that lies within a rectangular pyramid, whose axis is the $-z$-axis, and whose apex is his eye. In order to indicate the height of this pyramid, the user specifies its angular height, called the y *field-of-view* and denoted *fovy* (see Fig. 18). It is given in degrees.

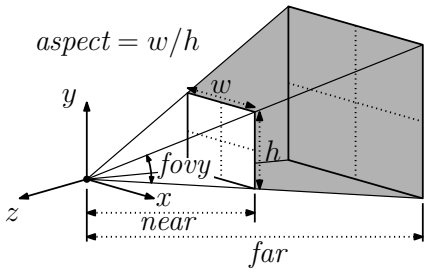


Fig. 18: OpenGL's perspective specification.

To specify the angular diameter of the pyramid, we could specify the x field-of-view, but the designers of OpenGL decided on a different approach. Recall that the *aspect ratio* is defined to be the width/height ratio of the window. The user presumably knows the aspect ratio of his viewport, and typically users want an undistorted view of the world, so the ratio of the x and y fields-of-view should match the viewport's aspect ratio. Rather than forcing the user to compute the number of degrees of angular width, the user just provides the aspect ratio of the viewport, and the system then derives the x field-of-view from this value.

Finally, for technical reasons related to depth buffering, we need to specify a distance along the $-z$-axis to the *near clipping plane* and to the *far clipping plane*. Objects in front of the near plane and behind the far plane will be clipped away. We have a limited number of bits of depth-precision, and supporting a greater range of depth values will limit the accuracy with which we can represent depths. The resulting shape is called the *viewing frustum*. (A *frustum* is the geometric shape that arises from chopping off the top of a pyramid. An example appears on the back of the US one dollar bill.) These arguments form the basic elements of the main OpenGL command for perspective.

```
gluPerspective(fovy, aspect, near, far);
```

All arguments are positive and of type GLdouble. This command creates a matrix which performs the necessary depth perspective transformation, and multiplies it with the matrix on top of the current stack. This transformation should be applied to the projection matrix stack. So this typically occurs in the following context of calls, usually as part of your initializations.

```
void myDisplay() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt( ... );                          // set up camera frame
  glMatrixMode(GL_PROJECTION);               // set up projection
  glLoadIdentity();
  gluPerspective(fovy, aspect, near, far);   // or glFrustum(...)
  glMatrixMode(GL_MODELVIEW);
  myWorld.draw();                            // draw everything
  glutSwapBuffers();
}
```

The function gluPerspective does not have to be called again unless the camera's projection properties are changed (e.g., increasing or decreasing zoom). For example, it does not need to be called if the camera is simply moved to a new location.

Canonical View Volume: In applying the perspective transformation, all points in projective space will be transformed. This includes point that are not within the viewing frustum (e.g., points lying behind the viewer). One of the important tasks to be performed by the system, prior to perspective division (when all the bad stuff might happen) is to clip away portions of the scene that do not lie within the viewing frustum.

OpenGL has a very elegant way of simplifying this clipping. It adjusts the perspective transformation so that the viewing frustum (no matter how it is specified by the user) is mapped to the same canonical shape. Thus the clipping process is always being applied to the same shape, and this allows the clipping algorithms to be designed in the most simple and efficient manner. This idealized shape is called the *canonical view volume* (also called *normalized device coordinates*).

The canonical view volume (after perspective division) is just a 3-dimensional rectangle. It is defined by the following constraints.

$$-1 \le x \le +1, \qquad -1 \le y \le +1, \qquad -1 \le z \le +1.$$

(See Fig. 19 for example. Imagine that the x-axis is pointing out of the paper towards you. The viewing frustum of Fig. 19(a) is mapped to the cube shown in Fig. 19(b).)

After applying this transformation, the (x, y) coordinates indicate the location of the projected point on the final viewing window. The z-coordinate is used for depth. There is a reversal of the z-coordinates, in the sense that before the transformation, points farther from the viewer have smaller z-coordinates (larger in absolute value, but smaller because they are on the negative z side of the origin). Now, the points with $z = -1$ are the closest to the viewer (lying on the near clipping plane) and the points with $z = +1$ are the farthest from the viewer (lying on the far clipping plane). Points that lie on the top (resp., bottom) of the canonical volume correspond to points that lie on the top (resp., bottom) of the viewing frustum.

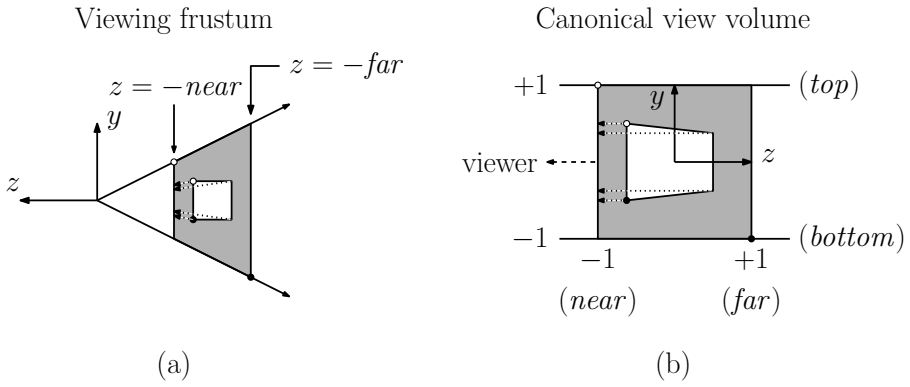


Fig. 19: Perspective with depth and the canonical view volume.

Mapping to the Viewport, Rasterization, and Hidden Surface Removal: The last step of the process involves scaling the (x, y)-coordinates of each vertex in normalized device coordinates onto the viewport. The normalized device coordinates form a square $[-1, +1] \times [-1, +1]$. Given the corners of the viewport, it is a simple exercise to transform this to the corners of the viewport (see Fig. 20(a)). This transformation was derived at the end of Lecture 5.

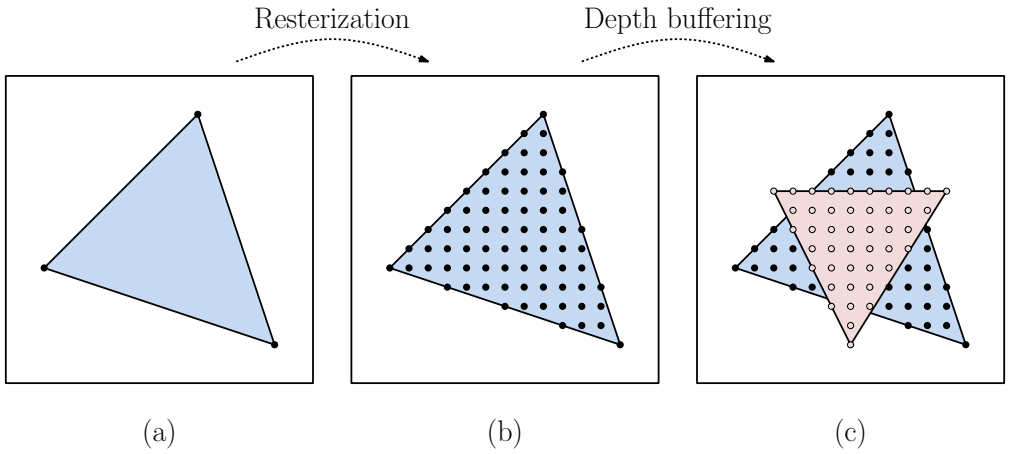


Fig. 20: Rasterization and hidden surface removal by depth buffering.

After the vertices of each polygon have been mapped to the viewport, OpenGL then converts the interior of the polygon into a collection of pixels, called *fragments*. Each fragment is associated with the color of the associated point on the polygon and its distance from the viewer (stored in the z-coordinate). These fragments are then sent to the color buffer for drawing. This is called *rasterization* (see Fig. 20(b)). If depth buffering is enabled, the z-coordinates of the fragments are stored in the depth buffer. If two fragments from different objects share the same (x, y)-coordinates then there is a conflict. If depth buffering is not enabled, fragments are written directly to the color buffer, and so you see the last one that was written. If depth buffering is enabled, then their depth values are compared, and the closer pixel is written to the color buffer.

Lecture 7: Tips on Programming in C++

Sources: See the links at the end of this document for resources on the Web.

Overview: This is a brief overview of C++, for people who know Java and C. We will focus particularly on aspects of C++ that will be the most useful for graphics and game programming. We will also focus on the aspects of C++ that are different from C and Java.

Primitive types and pointers: In Java, everything is either a primitive type (int, char, float, etc.) or an object. Objects in Java are essentially references to objects. C++ inherits its basic types from C, and adds an two additional types, *classes* and *references*. The C++ types include primitive types (int, char, float etc.), enumerations, C-style structures and classes, pointers, and references. As in C, a pointer is an address in memory, and an array is a pointer to the first element of an array. We will discuss references later.

Constants and Enumerations: One interesting feature of C++ is the ability to declare that an object is constant. This means that, once initialized, its value cannot be changed. This avoids the need for many #define constants that crop up in many C programs (which are not type checked) and is analogous to static final in Java. Here is an example:

```
const float FREEZING_POINT = 32.0;
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 300;
const char QUIT = 'q';
```

By convention, constants are expressed using ALL CAPITAL letters.

A common use for constants is for indicating quantities that take on a discrete set of values. As in C and never versions of Java, C++ offers *enumerations*, which provide an easy way to define such constants. Here are some examples:

```
enum DayOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT };
enum GameState { RUNNING, PAUSED, FINISHED };

DayOfWeek today = TUE;
GameState gameState = PAUSED;
```

Classes: Class syntax in C++ is quite similar to Java. (Note however, that a semicolon is placed at the end of a C++ class.) As in Java, class members may be public (accessible externally), private (accessible only internally), or protected (accessible only internally or by derived classes). Unlike Java, it is possible to define class methods outside the class body as well as inside. (In C++, class methods are usually called *member functions*.)

```
//------------------------------------------------------------
// This would appear in file Vector2d.h
//------------------------------------------------------------
class Vector2d {
  public:
    Vector2d() { x = y = 0; }           // default constructor
    Vector2d(double xx, double yy);     // constructor
    double getX() { return x; }         // getters
    double getY() { return y; }
    void setX(double xx) { x = xx; }   // setters
    void setY(double yy) { y = yy; }
    Vector2d addTo(Vector2d v);         // add to vector v
  private:
    double x;                           // member data
    double y;
};
```

We have defined the getters and setters inside the class. We have chosen to define the constructor and the addTo function outside the class (see below).

```
//-------------------------------------------------------------
// This would appear in file Vector2d.cpp
//-------------------------------------------------------------
#include "Vector2d.h"
Vector2d::Vector2d(double xx, double yy)
  { x = xx;  y = yy; }

Vector2d Vector2d::addTo(Vector2d v)
  { x += v.x; y += v.y; }
```

Notice that, when outside the class, it is necessary to use the *scope resolution operator*, "::", to indicate that a name is associated with a particular class. Thus, when we define the function addTo outside the class, we need to specify Vector2d::addTo, so the compiler knows we are talking about a member of Vector2d.

What why would you prefer defining a member function outside a class body rather than inside? Most C++ style manuals insist that *all* member functions be defined outside the class. The rationale is that a user should see only the public interface, not the implementation. If you define a function within the class, it should be very short, no loops or conditionals. C++ takes a definition inside the class body to be a hint that this should be an *inline function*, which means that the function is expanded, rather than being called. This produces more efficient code, but excessive use of this feature results in unnecessarily long executable files (so called, "code bloat").

Stream I/O: Input and output in C++ is performed by the operators ¿¿ and ¡¡, respectively. The standard output stream is called "cout" and the standard stream is called "cin". Here is a simple example.

```
int x, y;
cin >> x >> y;    // input x and y
cout << "The value of x is " << x << " and y is " << y << "\n";
```

The character "\n" generates an end-of-line. It is also possible to use standard C I/O (printf and scanf), but it is not a good idea to mix C++ stream I/O with C standard I/O in the same program.

Include files and namespaces: Following C's convention, declarations are stored in files ending in ".h" and most code is stored in files ending in ".cpp" or ".cc". Objects like cin and cout are defined by the system. At the start of each program, it is common to begin with a number of directives that include common system declarations. Here are some of the most useful ones.

```
#include <cstdlib>    // standard definitions from C (such as NULL)
#include <cstdio>     // standard I/O for C-style I/O (scanf, printf)
#include <cmath>      // standard C math definitions (sqrt, sin, etc.)
#include <iostream>   // C++ stream I/O
#include <string>     // C++ string manipulation
#include <vector>     // C++ STL vector (an expandable vector object)
#include <list>       // C++ STL list (a linked list)
```

In order to keep your program names from clashing with C++ program objects, many named entities are organized into *namespaces*. Most system objects are stored in a namespace called "std". This includes, for example, cin and cout, mentioned above. To access objects from this namespace, you need to use a scope resolution operator, "::". For example, to refer to cin, you would use std::cin and the refer to cout you would use std::cout. To avoid this extra verbiage, you can invoke the "using" command to provide direct access to these names.

```
using std::cin;              // make std::cin accessible
using std::cout;             // make std::cout accessible
using namespace std;         // make all of std accessible
```

Memory Allocation and Deallocation: One of the principal differences with C++ and Java is the need to explicitly deallocate memory that has been allocated. Failure to deallocate memory that has been allocated results in a *memory leak*, which if not handled, can cause your program to exhaust all its available memory prematurely and crash. As in Java, memory is allocated using new. This returns a pointer to the newly allocated object. Unlike the primitive C function malloc, the new operator returns an object of the specified type, and performs initialization by invoking the constructor. For example, to allocate an object of type Vector2d, we could do the following.

```
Vector2d* p;                  // p is a pointer to a Vector2d
p = new Vector2d(3, -4);      // allocate vector, initialized to (3,-4)
p->setY(2.6);                 // set its y-coordinate to 2.6
cout << p->getX();            // print its x-coordinate
delete p;                     // deallocate p's memory
```

Recall from C that, when dealing with pointers, the "*" operator is used to dereference its value and if p is a pointer to a structure or class, then "p->xxx" is used to access member xxx.

Array Allocation: It is also possible to use new and delete to allocate arrays. This is a common way to generate vectors whose size is known only at execution time. When deleting such an array, use "delete []". Here is any example.

```
int n = 100;
Vector2d* p = new Vector2d[n];    // allocate an array of n vectors
p[5] = Vector2d(3, -4);           // set p[5] = (3,-4)
delete [] p;
```

Constructors and Destructors: The most common place where memory is allocated and deallocated is when classes are first constructed or destroyed, or in class member functions that insert new entries into a dynamic object. If a class allocates memory, it is important that when the class object ceases to exist, it must deallocate all the memory that it allocated. Whenever an object is about to cease to exist (e.g., the scope in which it was defined is exiting), the system automatically invokes a special class function called a *destructor*. Given a class X, the corresponding destructor is named ~X. Here is a simple example, of a class, which allocates an array.

```
class VectorArray {
  public:
    VectorArray(int capac);         // constructor
    Vector2d at(int i) { return A[i]; }
    // some functions omitted...
    ~VectorArray();                 // destructor
  private:
    int n;                          // array capacity
    Vector2d* A;                    // array storage
};
                                    // constructor
VectorArray::VectorArray(int capac) {
  n = capac;
  A = new Vector2d[n];              // allocate array storage
}

VectorArray::~VectorArray() {
  delete [] A;                      // deallocate array storage
}
```

Note that you do not invoke the destructor (in fact, you can't). The system does it automatically.

Using STL Data Structures to Avoid Memory Allocation: Memory allocation and deallocation is a pain, and it is easy to make mistakes, which can result in memory leaks or attempting to access a pointer that references a piece of memory that has been deleted. There is a remarkably easy way to avoid memory allocation and deallocation.

A remarkably large amount of memory allocation and deallocation arises when dealing with very common dynamic structures, such as vectors, lists, hash-maps, and the like. By *vector*, I mean a variable-sized array (which may be appended to), and by *list*, I mean a doubly linked list. Rather than going through the hassles of implementing your own dynamic structures from scratch, and dealing with the headaches of memory allocation and deallocation, it is much simpler to use the built-in vector and list types provided by the C++ *Standard Template Library*, or STL. The STL provides data structures for a number standard containers, such as stacks, queues, deques (double ended queues), vectors, lists, priority queues, and maps.

One of the important features of the STL is that each data structure can store objects of any one type. Such a class whose definition depends on a user-specified type is called a *template*. The type of the object being stored in the container is given in angle brackets. For example, we could define vectors to hold 100 integers or 500 characters as follows:

```
#include <vector>             // class vector definitions
using namespace std;          // make std accessible
vector<int> scores(100);      // a vector of (initially) 100 integers
vector<char> buffer(500);     // a vector of (initially) 500 characters
```

Here are a couple of examples of how to use an STL vector.

```
int n = 100;
vector<int> myInts(n);        // a vector with 100 ints
vector<Vector2d> myVects;     // an empty vector of Vector2d objects
myVects.push_back(Vector2d(1,5));  // use push_back to append items
myInts[5] = 14;               // use "[]" to index entries
myVects[0].setX(2);           // invoke a method on an element
```

STL vectors and lists provide too many capabilities to be listed here. I will refer you to online documentation for more details. There are a number of other useful STL data structures, including dictionaries and priority queues.

STL vectors are superior to standard C++ arrays in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([ ]). They can also be accessed by the at member function, which also performs array bounds checking. (As in C, arrays in C++ do not even know their size, and hence range checking is not even possible.) In contrast, a vector object's size is given by its size member function. Unlike standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the resize member function. Here are some examples:

```
int i = // ...
cout << scores[i];                  // index (range unchecked)
buffer.at(i) = buffer.at(2 * i);    // index (range checked)
vector<int> newScores = scores;     // copy scores to newScores
scores.resize(scores.size() + 10);  // add room for 10 more elements
```

Another handy STL container is the list, which implements a doubly linked list. Here is how to declare a list of floats. By default, the initial list is empty.

```
#include <list>               // class list definitions
using namespace std;          // make std accessible
list<float> myList;           // an (initially empty) list of floats
```

```
myList.push_back(5);          // append 5 to back of the list
myList.push_front(2);         // prepend 2 to the front of list
myList.pop_back();            // remove the last elements (5)
```

List supports operations such as size (number of elements), empty (is the list empty?), front and back (return reference to first/last elements), push_front and push_back (insert to front/back), pop_front and pop_back (remove from front/back).

Iterators: The STL container classes introduced above all define a special associated class called an *iterator*. An iterator is an object that specifies a position within a container and which is endowed with the ability to navigate to other positions. If p is an iterator that refers to some position within a container, then $*p$ yields a reference to the associated element.

Advancing to the next element of the container is done by incrementing the iterator. For example, either $++p$ or p++ advances p to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value. Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container. The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element *just beyond* the last element of the container.

Let us see how we can use iterators to enumerate the elements of an STL container C. Suppose, for example, that C is of type `vector<int>`, that is, it is an STL list of integers. The associated iterator type is denoted "`vector<int>::iterator`". For example, the code below demonstrates how to sum the elements of an STL vector V using an iterator.

```
vector<int> v;
typedef vector<int>::iterator Iterator;     // iterator type
int sum = 0;
for (Iterator p = v.begin(); p != v.end(); ++p) {
  sum += *p;
}
cout << "The sum of elements in the list is " << sum << "\n";
```

Different containers provide iterators with different capabilities. Most STL containers (including lists, sets, and maps) provide the ability to move not only forward, but backward as well. For such containers the decrement operators $--p$ and $p--$ are also defined for their iterators. This is called a *bidirectional iterator*.

A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, p, the value $p + 3$ references the element three positions after p in the container. This is called a *random-access iterator*. As with pointers, care is needed in the use of iterators. For example, it is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it. Attempting to dereference an invalid iterator can result in your program aborting. As mentioned earlier, iterators can be *invalid* for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

C++11 and Range For Loops: The iterator syntax is quite messy. The latest version of C++, called C++11, has a vastly simpler way in which to iterate through STL structures. In C++11, the previous code block would be written as follows.

```
vector<int> v;
int sum = 0;
for (int x : v) {
  sum += x;
}
cout << "The sum of elements in the list is " << sum << "\n";
```

C++11 is supported in Visual Studio 2013 and the latest releases of the GCC compiler.

References: In C, all parameter passing to functions is performed *by value*. This has two important implications. First, altering the value of an formal parameter inside a function has no effect on the actual parameter in the calling function. If you want to modify the value of a parameter, you need to pass a pointer to the parameter. This is messy, since it implies that the function needs to de-reference the resulting parameter whenever it uses it. Second, passing a large class or structure to a function means that its entire contents will be copied. This can be inefficient for very large structures. (Note that this does not apply to C++ arrays, however, since an array is just a pointer to its first element. However, this would apply if you were to pass an STL vector by value. Such an operation would involve making a duplicate copy of the entire vector.)

In Java, this was handled very elegantly by making all objects in references. Thus, small primitive types, such as int and float are passed by value, and all objects are passed by reference. If it is desired to change the value of a primitive type, standard wrappers, like Integer were defined.

In C++, this issue was addressed by defining a special type, called a *reference*. The following line defines the variable i to be an integer, and r to be a reference to this integer. All references to r are effectively "aliases" to references to i.

```
int i = 34;
int& r = i;                   // r is an alias for i
r = 27;                       // this is equivalent to i = 27
```

References are rarely used as shown above. Instead, they are principally used for passing parameters, "by reference" to functions. A reference parameter has two advantages over a value parameter. First, it can be modified (without any messy pointer de-referencing) and it is more efficient for class objects, since only the address of the object (not the entire object) needs to be conveyed to the function.

```
void f(int& r, Vector2d& u) {
  r = 27;                     // changes the actual parameter to 27
  u.setY(4);                  // alters y-coordinate of actual parameter
}

// ... in your main program
  int i = 34;
  Vector2d v(2, 1);
  f(i, v);
  cout << i << " " << v.getY() << "\n"; // outputs "27 4"
```

Although passing class objects by reference is more efficient than by value, it has the downside that the function may inadvertently modify the value of the parameter, without the compiler being able to detect it. To handle this, C++ allows for something called a *constant reference*, which is a reference to an object that can be read, but not modified. Since the above function does not modify its arguments, it would more aptly be written in the following form:

```
Vector2d add(const Vector2d& u, const Vector2d& v) {
  return Vector2d(u.getX() + v.getX(), u.getY() + v.getY());
}
```

To get this to work, we should inform the compiler that the functions getX() and getY() are not allowed to modify the class variables. To do this, we would add the modifier "const" to their function definitions.

Operator Overloading: One of the nicest features of C++ when dealing with geometric objects (points, vectors, rectangles, etc) is the ability to redefine standard operators, such as addition ("+"), subtraction ("-"), multiplication ("*"), and division ("/") and other operations, such as input ("¿¿") and output ("¡¡"). Suppose we want to define an addition operator that adds two vectors. We could do it as follows:

```
Vector2d operator+(const Vector2d& u, const Vector2d& v) {
  return Vector2d(u.getX() + v.getX(), u.getY() + v.getY());
}
```

This would allow us to perform arithmetic on vectors, as in Vector2d w = u + v.

Here is an example how to define an output operator for Vectors.

```
ostream& operator<<(ostream& out, const Vector2d& p) {
 out << "(" << p.getX() << ", " << p.getY() << ")";
 return out;
}
```

To output a vertex, we could then write cout ¡¡ "The vector v is " ¡¡ v. If $v = (2, 1)$, then this would generate the output "The vector v is (2, 1)".

While operator overloading is cool, it should be used sparingly, and only when the meaning of an operation is clear.

More information: There are a number of resources on the web, which can be found by searching for "C++ for Java programmers". Here are some examples:

```
http://www.cs.williams.edu/~lenhart/courses/cs371/c++forjava.html
http://www.icce.rug.nl/documents/cplusplus/
```

For a brief survey of the new features of C++11, see

```
http://www.cprogramming.com/c++11/what-is-c++0x.html
```

Lecture 8: Geometric Data Structures for Games: Meshes and Manifolds

Sources: This material is based on Chapt 10 of *Game Engine Architecture* by J. Gregory and Chapt 12 of *Fundamentals of Computer Graphics* (3rd edition) by P. Shirley and S. Marschner. The material on DCELs is covered in *Computational Geometry: Algorithms and Applications* (3rd Edition) by M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars.

Geometric Data Structures: In modern games, it is often desirable to give the player the feeling of being emmersed in an interesting environment. Managing large geometric models can be computationally very challenging. The objects being modeled can be of various forms ranging from rigid structures like buildings, articulated entities like people and animals, and amorphous objects like water and smoke. There are many ways that a game may need to interact with such models:

Rendering: The objects visible to the camera need to be rendered. This involves lighting and texturing, hidden-surface removal, efficiently computing which objects are at least partially visible, computation of shadows, determining the effect of atmospheric effects such as smoke.

Navigation and motion: Non-player entities often need to plan an obstacle-free path through a complex and dynamically changing environment. Often groups of objects need to plan coordinated motion, like a group of soldiers, pedestrians walking on the street, or a moving herd of animals. Auxiliary data structures may need to be maintained to facilitate computing these paths.

Collision detection: It is necessary to determining collisions to forbid the player (or other moving entities) from violating basic physical laws. Some collisions, such as weapons hits, are significant to the internal state of the game. Some games involve interactions of movable physical structures (such sling-shooting a bird into a towers of blocks), it may be necessary to simulate how they topple and collapse.

In this lecture, we will discuss fundamental geometric data structures from a fairly general perspective. In future lectures, we will see how these basic data structure can be adapted and applied to tasks like those mentioned above. Examples include:

Triangle Meshes: Are used for the storage and manipulation of geometric models. Such meshes can also be structured hierarchically in order to provide level-of-detail approximations.

Geometric graphs and subdivisions: Geometric graphs provide an invisible structure which can guide the navigation of non-player characters (NPCs). Geometric subdivisions are often used for interpolation and providing a structure for computing the motion of fluids and gasses.

Scene graphs: These are used for representing hierarchical assemblies of objects and their relationships.

Grids: These are among the simplest structures for storing geometric objects and are often used in collision detection and fluid dynamics.

Hierarchical Spatial Subdivisions: Data structures like quadtrees, kd-trees, and BSP-trees are also used in collision detection, but they are much more flexible and adaptable than grids. They are also used in visibility culling, and ray-shooting.

Triangle Meshes: While there are a number of different methods for representing 3-dimensional solid objects, the most common method used in computer games is based on representing the surface of the object as a mesh of triangles. These are also known as *triangular meshes*, *triangulated meshes*, and *triangular irregular networks*[2] (TINs). (An example of a triangle mesh is shown in Fig. 21(a), and an example of a quadrilateral mesh is shown in Fig. 21(b).)

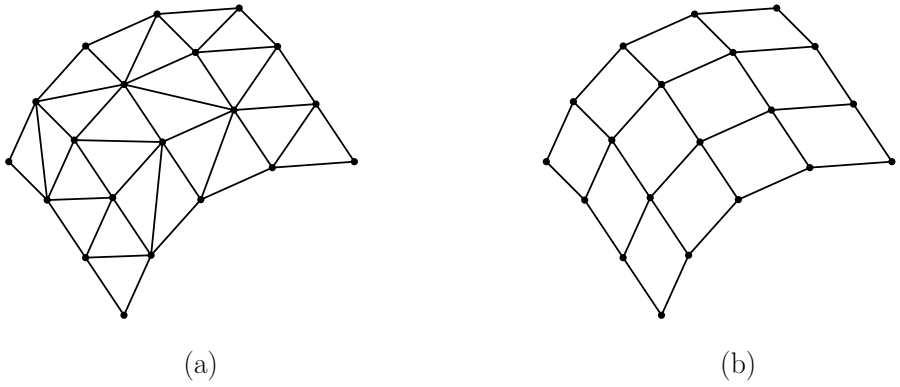


Fig. 21: A triangle mesh (a) and a quadrilateral mesh (b).

Why use triangles? They have a number of nice properties. First, they are the simplest polygon, unlike k-sided polygons for $k \geq 4$, triangles are always convex and (in 3-dimensional space) they are always planar. Finally, because they are so ubiquitous, graphics hardware has been optimized to efficiently render triangle meshes. The most common alternative to the triangle is the quadrilateral. One reason that quadrilaterals are popular is that if they are sufficiently regular, they can be stored implicitly in a 2-dimensional array of vertices.

Why connect the triangles into a mesh, as opposed to just considering a set of triangles? By putting triangles in a mesh, it is possible to perform global optimizations that would not be easy given an arbitrary set of triangles. For example, to compute the shadow cast by a mesh, it suffices to visit just the border edges of the mesh, which is typically many fewer than the total number. Also, because many triangles may share the same vertex, we can perform optimizations such as computing the illumination for each vertex once, and then reuse this information for each triangle that shares this vertex.

[2]You might wonder, what is "irregular" about triangular meshes. In the early days of meshing, most meshes were based on a regular 2-dimensional array of vertices, which were linked together in a grid of quadrilaterals, like a large fish net. Except along the boundary, each vertex in such a grid has exactly four neighbors, and so the grid is *regular*. Triangle meshes are not so constrained, and hence are *irregular*.

Graphics Representation: What is the best way to represent a 3-dimensional triangle mesh for the sake of rendering? Perhaps the simplest method would be to create a class called *Triangle*, which would consist of an array of three vertices, where each vertex would be represented by a vector of three floats or doubles. This means that every triangle would involve nine floating point numbers. (By the way, this is just what is needed to represent the geometry. We should also store information for lighting such as surface normal vectors at each of the vertices and coordinates for texture mapping.) Clearly, this naive solution is not very space efficient.

A better approach would be to generate two arrays. The first array holds the *vertices*. Each entry consists of three vertex objects, where a vertex object is a vector, consisting say of three floating point coordinates (see Fig. 22(b)). An alternative approach (which is used by OpenGL) is to store all the coordinates in a single 1-dimensional array, where each three consecutive entries ($[0, 1, 2]$, $[3, 4, 5]$, and so on) are the coordinates of a single vertex.

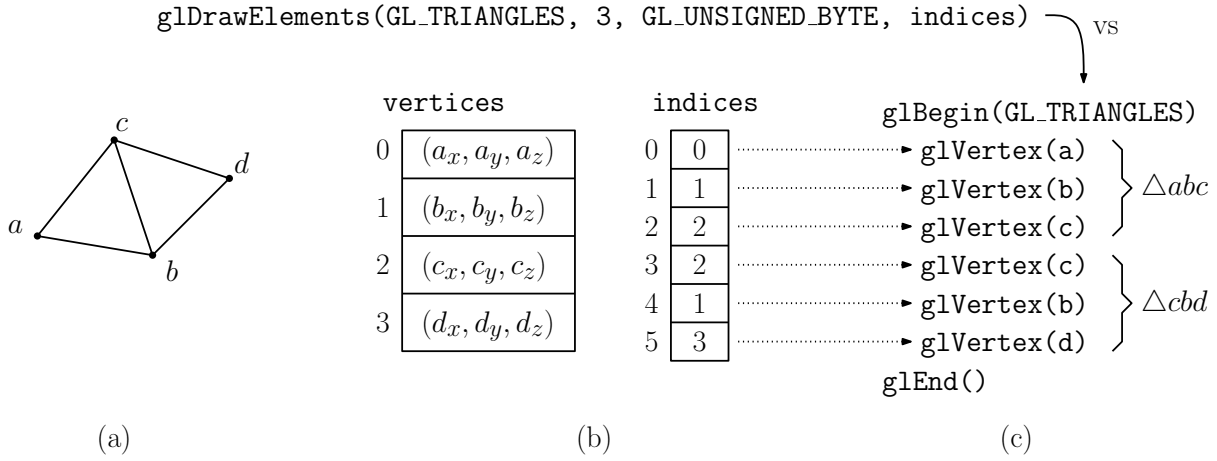


Fig. 22: Drawing a triangle mesh using an OpenGL index array.

In order to represent the triangles, we generate an array of *indices*, where the first three entries specify the vertices of the first triangle, the next three entries specify the vertices of the second triangle, and so on (see Fig. 22(b) for the mesh given in Fig. 22(a)). Note that each vertex is stored only *once*, no matter how many triangles contain it. Once the vertex array has been set up, we only need one index to reference an individual vertex. The straightforward method of doing this would involve transmitting 18 floating point values and making 8 OpenGL calls (see Fig. 22(c).

We have seen how to draw a sequence of triangles in OpenGL using glBegin(GL_TRIANGLES). This involves transmitting nine coordinates for each triangle that is drawn. OpenGL supports a more efficient mechanism, which involves setting an array of vertices and array of indices. Through the use of the OpenGL function glDrawElements it is possible to render the entire mesh by passing in the arrays vertices and the indices (see the code block below). (I will not explain the various arguments, but instead refer you to the OpenGL documentation. There are, by the way, more sophisticated ways of maintaining multiple buffers of vertices and other information, such as colors.)

Mesh Toplogy: A mesh is characterized by two important features, (1) where in space are the vertices that make up its triangles and (2) how are these triangles connected together? The answer to question (1) defines the *geometry* of the mesh. The answer to (2) defines the *topology* of the mesh.

When defining how triangles can be joined to make a mesh, there are usually certain requirements that are laid down. The first requirement is that the mesh be a *cell complex*. Saying that a mesh is a cell complex means that the triangular elements of the mesh are "properly joined" to each other. What does this mean? For example, when two triangles intersect, they either share an entire edge in common or they share just a vertex in common. There are many illegal ways that triangles might intersect (see Fig. 23), but these cannot occur in a cell complex.

```
GLfloat vertices[] = { ax, ay, az,            // define vertices
                       bx, by, bz,
                       cx, cy, cz,
                       dx, dy, dz };
GLubyte indices[] = { 0, 1, 2, 2, 1, 3 };    // define indices

glEnableClientState(GL_VERTEX_ARRAY);        // enable vertex array
glEnableClientState(GL_INDEX_ARRAY);         // enable index array

glIndexPointer(GL_UNSIGNED_BYTE, 0, indices);// location of indices
glVertexPointer(3, GL_FLOAT, 0, vertices);   // location of vertices
                                          // draw the mesh
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices);
```

Cell complexes (that represent surfaces) are composed of three types of elements: 0-dimensional elements called *vertices*, 1-dimensional elements called *edges*, and 2-dimensional elements called *faces*.

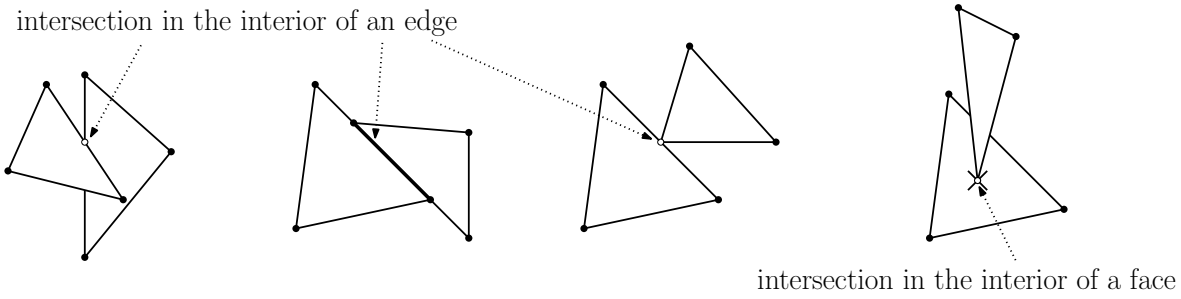


Fig. 23: Examples of triangle intersections that cannot occur in a cell complex.

Let us assume from her on that our meshes are cell complexes. The second condition that one would like to have satisfied by a mesh is that it defines 2-dimensional surface. In topology terms, this is called a *2-manifold*. Formally, this means that, if you consider a small neighborhood around any point (which might be a vertex, in the interior of an edge, or in the interior of a triangle face) the region around this point forms a 2-dimensional topological disk. Why might this fail to happen? Consider the neighborhoods shown in Fig. 24(a), (b), and (c). In all three cases, the neighborhood of the point is topologically equivalent to a 2-dimensional disk. However, in Fig. 24(d) and (e), the neighborhood of the point is definitely not a disk.

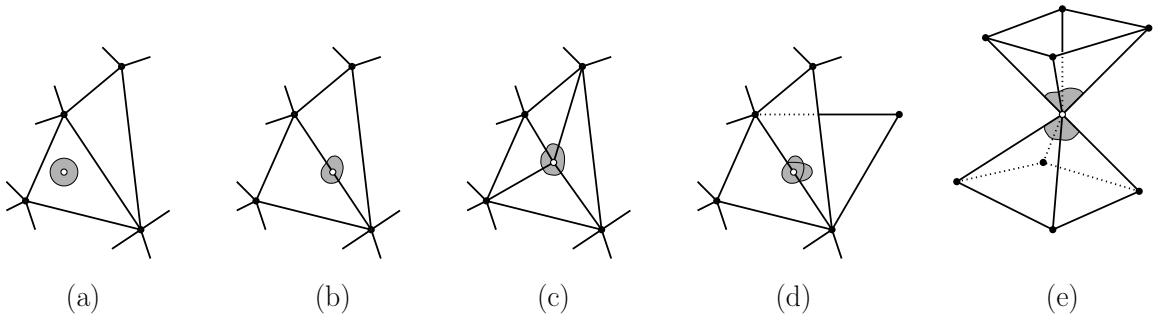


Fig. 24: Examples of cell complexes that are 2-manifolds (a)–(c), and those that are *not* 2-manifolds (d)–(e).

An equivalent characterization of a 2-manifold for cell complexes is that each edge should be incident to exactly

two triangles and each vertex should have a single loop of triangles about it.

Unfortunately, pure 2-manifolds do not allow for models that have a boundary, since the neighborhood surrounding a boundary point is essentially a topological half-disk. We say that a surface is a *2-manifold with boundary* if every interior point of the mesh satisfies the above definition for 2-manifolds, and each boundary point has a single semi-disk as its neighborhood.

Although there are a few applications of non-manifold surfaces, it is common to assume that all the triangular meshes that we will deal with are 2-manifold cell complexes (possibly with a boundary).

Doubly-connected Edge List: What sort of data structure can be used for storing triangle meshes? As far as OpenGL is concerned, a simple index array is sufficient. But your game program may require more structure. For example, suppose that you are using a 2-manifold to represent a terrain, and bug is walking across this terrain. As the bug walks leaves one triangle, we would like to be able to determine efficiently which new triangle it is entering. One way to do this would be "walk" around the edges of triangles of the mesh that the bug visits (see Fig. 25). In order to do this, we need to know which edges are adjacent to each triangle, and for each edge, we need to know what triangle lies on the other side of this edge.

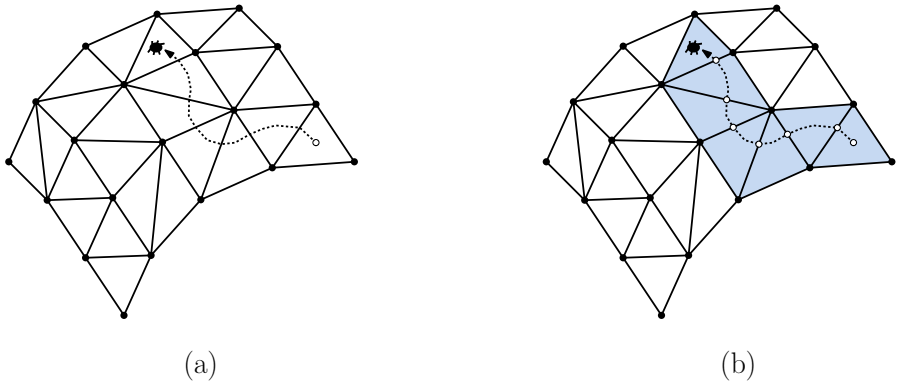


Fig. 25: Walking a bug on a mesh.

There are a number of different data structures for doing this. These include the *winged-edge data structure*, the *half-edge data structure*, and *doubly-connected edge list*, and the *quad-edge data structure*. All of these structures are equivalent, in the sense that given one, it is an easy matter to convert it into any of the others. We will discuss the *doubly-connected edge list* (or DCEL).

In the DCEL of a mesh, there are three sets of records one for each element in the cell complex: *vertex records*, a *edge records*, and *face records*. For the purposes of unambiguously defining left and right, each undirected edge is represented by two directed *half-edges*.

Vertex: Each vertex stores its coordinates, along with a reference to any incident directed edge that has this vertex as its origin, v.inc_edge.

Edge: Each undirected edge is represented as two directed edges. Each edge has a reference to the oppositely directed edge, called its *twin*. Each directed edge is implicitly associated with two vertices, its *origin* and *destination*. Each directed edge is also implicitly associate with two faces, the one to its left and the one to its right.

Each edge stores a reference to the origin vertex e.org. (Note that we do not need to store the destination vertex, since it may be computed as e.twin.org.) Each edge also stores a reference to the face to the left of the edge e.left. (Again, we do not need to store the face to the right, since it can computed as e.twin.left). We also store the next and previous directed edges in counterclockwise order about the incident face, called e.next and e.prev, respectively.

Face: Each face f stores a reference to an arbitrary directed edge such that this face lies to the left of the edge, called f.inc_edge.

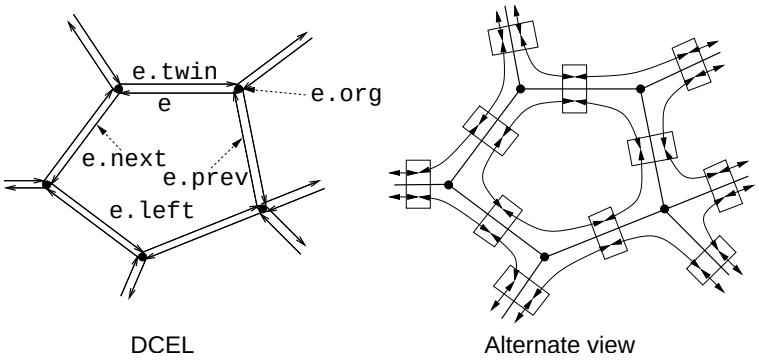


Fig. 26: Doubly-connected edge list.

Fig. 26 shows two ways of visualizing the DCEL. One is in terms of a collection of doubled-up directed edges. An alternate 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 reference provide links around each face of the polygon. The next references are directed counterclockwise around each face and the prev references are directed clockwise.

Of course, in addition the data structure may be enhanced with whatever application data is relevant. In some applications, it is not necessary to know either the face or vertex information (or both) at all, and if so these records may be deleted. As an example of how to use the DCEL, suppose that we wanted to enumerate the vertices that lie on some face f in counterclockwise order. The code block below shows how to do this.

Vertex enumeration about a face using DCEL

```
verticesOnFaceCCW(Face f) {
    Edge e = f.inc_edge;          // any edge such that f is to the left
    do {
        output e.org;             // output e's origin vertex
        e = e.next;               // next edge about f in CCW order
    } while (e != f.inc_edge);    // done when we return to start
}
```

Let's try a slightly trickier one. Suppose that you at a vertex v, and you wish to list all the vertices that are adjacent to v in counterclockwise order. First, we would access v.inc_edge to find any edge e that has v as its origin. The vertex on the other side of this edge is given as e.twin.org. Next, how would we determine the next adjacent vertex in counterclockwise order? First, observe that e.prev is the reverse of the next edge in counterclockwise order emanating from v. We then reverse this edge to obtain the desired next edge, e.prev.twin. Here is the code:

(As an exercise to see whether you understand this, you might try repeating these two enumerations, but this time do it in clockwise order.)

Lecture 9: Geometric Data Structures for Games: Geometric Graphs

Sources: Today's materials is presented in part in *Computational Geometry: Algorithms and Applications* (3rd Edition) by M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars.

```
adjacentVerticesCCW(Vertex v) {
    Edge e = v.inc_edge;          // find starting edge
    do {
        output e.twin.org;        // output vertex on opposite end of e
        e = e.prev.twin;          // jump to the next edge in CCW order
    } while (e != v.inc_edge);    // repeat until looping back
}
```

Geometric Graphs and Subdivisions: We continue our discussion of fundamental geometric data structures. Today we will discuss geometric graphs and subdivisions. In computer games, it is often desirable to construct a framework to help plan the movement of various non-player entities. There are two particular structures of interest that we will consider.

Geometric graphs: This is a graph structure consisting of nodes and edges, where the nodes are points in space (for us, typically lying on a 2-dimensional surface) and the edges are straight lines (see Fig. 27(a)). Such graphs can be used for planning navigation and motion within an otherwise unstructured environment.

Subdivisions: A subdivision is a partitioning of some region of space into simple regions. A cell complex that subdivides a surface into a collection of cells is an example, but there are other types of subdivisions (see Fig. 27(b)). Subdivisions have many uses. They provide a framework onto which to anchor textures. They provide a method for describing points on a complex surface (by specifying the cell containing the point and location of the point within the cell). They can be used for interpolation, by specifying values at the vertices of the complex and interpolating values in between.

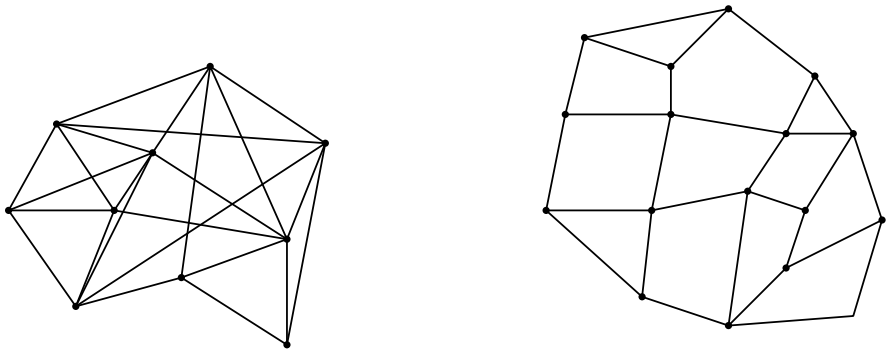


Fig. 27: A geometric graph (a), and a planar subdivision (b).

Although geometric graphs and subdivisions can be defined in obstacle-free environments, when used in games they are often constrained by the presence of constraining obstacles. In our discussions below we will often introduce each structure in an unconstrained setting, and then we will present strategies for restricting them depending on environmental constraints.

Visibility Graphs: Our first structure is motivated by the following simple navigation problem. Suppose that you want to compute the shortest path from one point s to another point t in the plane while avoiding a collection of polygonal obstacles (see Fig. 28(a)). The shortest path between two points is a straight line, but such a line is only of use to us if it does not intersect the interior of any obstacles. What we would like to do is to compute a graph such that the shortest path in this graph is the shortest path between s and t.

Observation: The shortest path between any two points that avoids a set of polygonal obstacles in the plane is a polygonal curve (a sequence of line segments joined end to end), whose vertices are either vertices of the obstacles or the points s and t, such that no segment intersects the interior of any of the obstacles.

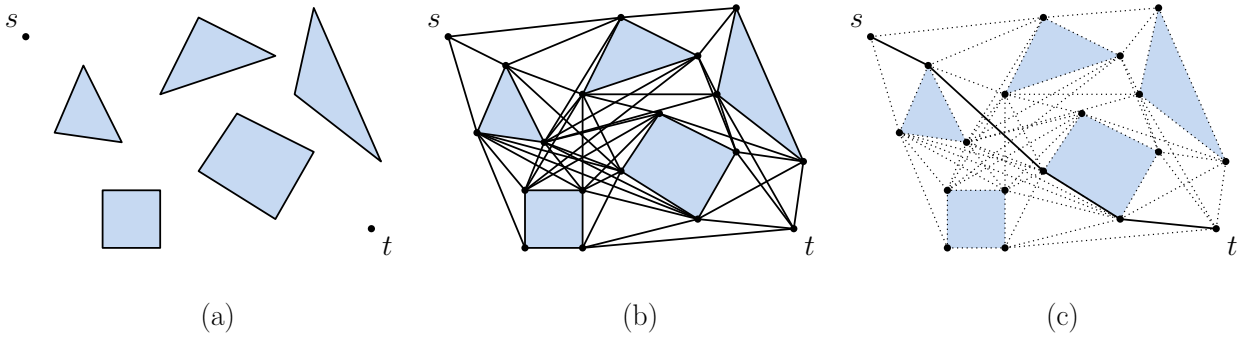


Fig. 28: A set of polygonal objects (a), the associated visibility graph (b), and a shortest path (c).

We say that two points u and v are *visible* to each other if the line segment $\overline{uv}$ does not intersect the interior of any obstacle. (Note that it is allowed to look right along the "wall" of any obstacle.) Basically, this observation tells us that that we should follow only visible segments, and it is never in our interest to "bend" the path unless we are at a corner point of some obstacle. This motivates the following definition:

Definition: The *visibility graph* of s and t and the obstacle set is a graph whose vertices are s and t the obstacle vertices, and vertices v and w are joined by an edge if v and w are either mutually visible or if (v, w) is an edge of some obstacle (see Fig. 28(b)).

It follows directly from the above observation that we can compute the shortest path between any two points by computing the visibility graph (including these points) and then running any shortest path algorithm, such as Dijkstra's algorithm on the result (see Fig. 28(c)).

If n denotes the total number vertices in our obstacles, then clearly the visibility graph can have as many as $O(n^2)$ edges. This is unfortunately rather large, but we will discuss a more efficient approach, which saves on space but sacrifices accuracy.

Can we compute the visibility graph efficiently? A naive approach would be to generate all $O(n^2)$ pairs of vertices (u, v), and then test whether the line segment $\overline{uv}$ intersects any obstacles. Testing a segment against the obstacles will take $O(n)$ time, which would lead to an $O(n^3)$ time algorithm. A more efficient approach is to perform an *angular sweep* around each vertex. Imagine that you want to simulate a rotating spot light centered at a vertex p (see Fig. 29). As you swing the spot-light beam around, you maintain a sorted list of edges that are intersected by the current spot-light beam. (The beam is actually an x-ray beam, since it penetrates through the obstacles.) Whenever the beam encounters a new vertex v in the sweep, it updates the edge list by either inserting or deleting edges that are incident to this vertex. If the change affects the *closest edge*, then we have discovered a new visibility edge from p to v, which we add to our visibility graph.

How long does this angular-sweep algorithm take? There are n object vertices, and visiting them in angular sorted order can be done in $O(n \log n)$ time. (I am omitting lots of details, but trust me.) Since this sweep needs to be done for all n vertices, the total running time is $O(n^2 \log n)$, which is much faster than the $O(n^3)$ naive solution.

Yao Graphs, Θ-Graphs, and Spanners: The principal shortcoming of the visibility graph is its size. Can we avoid the quadratic size complexity? Here is an idea. Rather than storing an edge to *every* visible vertex, how about if we store only a subset of the "most relevant" edges. But, how do we know which edges are most relevant? If a vertex has has a very high degree in the visibility graph, we know that there must be many edges that are

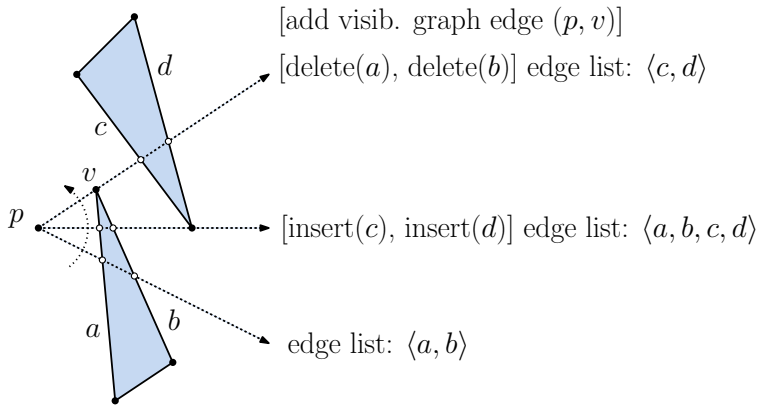


Fig. 29: Computing visibility edges by angular sweep.

traveling in essentially the same direction. Do we need all of these edges? What if we were to store just a single edge to the closest of these vertices.

This motivates the following idea. First, fix an integer $m \geq 6$. For each vertex u of your obstacle set (including s and t), subdivide the angular space surrounding u into a collection m infinite cones, each subtending an angle of $\theta_m = 2\pi/m$. For each such cone, add an undirected edge between u and the closest visible vertex (if one exists) within this cone (see Fig. 30(a)). In contrast to the visibility graph, which might have $O(n^2)$ edges, this graph has only $O(nm)$ edges. If m is a small constant, this is much more space efficient. This is sometimes called the *Yao graph* (named after Andrew Yao, a famous computer scientist, who first used this construction).

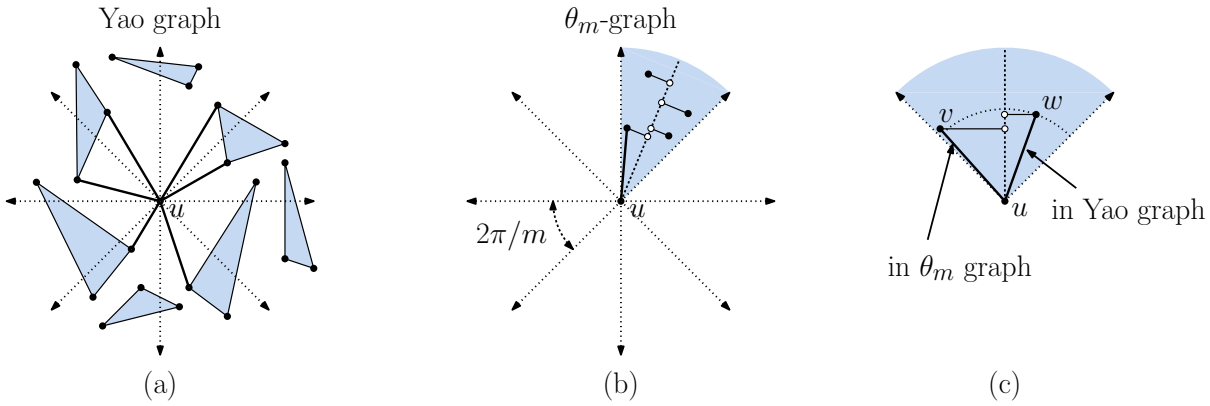


Fig. 30: The Yao graph (a), the θ_8-graph (b), and the difference between them (c).

For technical reasons, this definition is not the one that is usually preferred. (This is for two reasons, one practical and one theoretical. Unfortunately, it will take us too far afield to explain why.) The alternate definition that is preferred is the following. Again, consider any point u and all the visible vertices that lie within one of its cones. Project these vertices orthogonally onto the central axes of the cone. Add an undirected edge between u and the vertex whose projection is closest to u (see Fig. 30(a)). The resulting graph is called the *θ_m-graph*.

You might wonder whether the Yao graph (with m cones) and θ_m-graph are at all different. It would seem that the closest vertex to u will be the same as the vertex in the cone that has the closest projection onto the central axis. With a bit of work, it can be shown that they are indeed different, but you need a pretty large cone angle or you need to be quite careful in placing the points to generate a difference between the two (see Fig. 30(c)).

Both the Yao graph and the θ_m-graph are subgraphs of the visibility graph. Therefore, the shortest path from s

to t in either of these graphs cannot be any shorter that the true shortest path. The interesting question is just how bad might the resulting path be. We will not prove it here, but the following theorem shows that, as m grows, the θ_m-graph produces ever more accurate approximations to the shortest path length.

Theorem: Consider any obstacle set and any two points s and t. Let ℓ denote the length of the shortest obstacle avoiding path from s to t (that is, the shortest path in the visibility graph) and ℓ_m is the length of the shortest path in the θ_m-graph, for $m \geq 7$. Then

$$\frac{\ell_m}{\ell} \leq \frac{1}{1 - 2\sin(\theta_m/2)},$$

where $\theta_m = 2\pi/m$ (the cone angle).

There is a similar bound for Yao graphs, where the ratio is $1/(\cos\theta - \sin\theta)$ assuming you have at least 9 cones. You might ask, then how large do I need to set m to get a reasonably good approximation? To guarantee that ratio is at most 2 (a 100% error), you would need to set $m = 13$. To get the ratio down to 1.1 (a 10% error) you would need to set $m = 70$. These bounds are terribly pessimistic, however. In practice, I would conjecture that setting $m = 12$, would probably yield paths that are nearly indistinguishable from the optimum.

By the way, a graph that has the property that the shortest path in the graph is an approximation to the true shortest path is called a *spanner*. The theorem above states that the θ_m-graph is a ρ-spanner, where $\rho = 1/(1 - 2\sin(\theta_m/2)$ spanner.

In case you are interested, θ_m-graphs can be computed quite efficiently, in roughly $O(nm \log n)$ time. Since the graph has $O(nm)$ edges, this is pretty close to optimal. Indeed, the principal reason that people prefer the θ_m-graph over the Yao graph is that θ_m graphs can be computed more efficiently. In practice, the two data structures are virtually identical in all other respects.

Voronoi Diagrams and Delaunay Triangulations: Next, let us consider some examples of important geometric subdivisions. Suppose you are give a collection of points P in the plane, and you would like to define the notion of a *sphere of influence* for each point. For example, each point might represent a guard tower. Each guard tower is responsible for guarding the points that are closer to it than some other guard tower. This implicitly subdivides the plane into regions according to which of the guard towers is closest to it. The resulting subdivision of the plane is called the *Voronoi diagram*, and it is geometric structure of fundamental importance. An example of such a diagram is shown in Fig. 31.

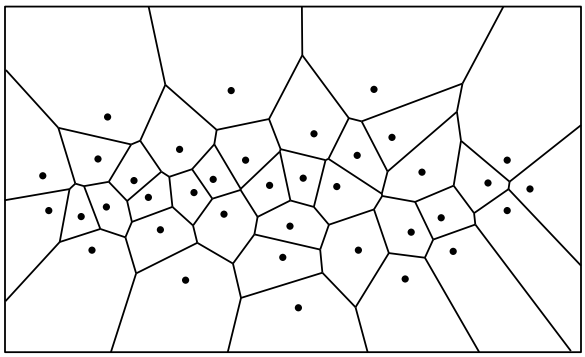


Fig. 31: The Voronoi diagram.

We will say much about the Voronoi diagram here. (If you want to learn more, take a course on Computational Geometry.) Given a set of n points, the Voronoi diagram can be computed in $O(n \log n)$ time. It can be represented as a DCEL (doubly-connected edge list).

The Voronoi diagram of a set of points in the plane is a planar subdivision, that is, a cell complex. The *dual* of such subdivision is another subdivision that is defined as follows. For each face of the Voronoi diagram, we create a vertex (corresponding to the point). For each edge of the Voronoi diagram lying between two sites p_i and p_j, we create an edge in the dual connecting these two vertices. Finally, each vertex of the Voronoi diagram corresponds to a face of the dual (see Fig. 32).

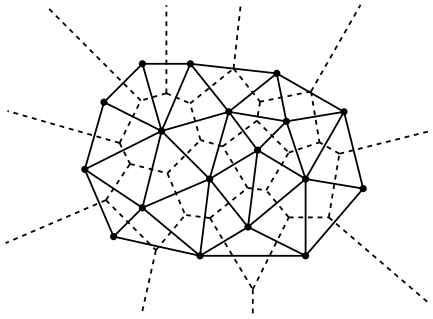


Fig. 32: The Delaunay triangulation (solid) and Voronoi diagram (dashed).

The resulting dual graph is also planar subdivision. Observe that (assuming that the points are not in some highly improbable configuration) the vertices of the Voronoi diagram all have degree three, that is, they are each incident to the three edges. It follows that the faces of the resulting dual graph (excluding the exterior face) are all triangles. Thus, the resulting dual graph is a triangulation of the sites, called the *Delaunay triangulation*.

Why are Delaunay triangulations of interest? If you are given a set of points and you want to determine the "best" way to connect them to form a cell complex, it is hard to beat the Delaunay triangulation. It has a number of interesting properties. First, it tends to produce nice "fat" triangles. For example, among all possible ways of triangulating a point set, the Delaunay triangulation maximizes the minimum angle appearing in any of its triangles. (It hates skinny triangles.)

Another interesting property is that, if you take any triangle, and consider the circumcircle passing through these three points, this circle cannot contain any of the other points inside it. This is called the *empty circumcircle property*. This might seem to be a property that many triangulations might possess, but this not the case. Indeed, if every triangle of a triangulation of a set of points satisfies the empty circumcircle property, then this is the Delaunay triangulation.

The Relative Neighborhood Graph and Gabriel Graph: We said that the Delaunay triangulation is a cell complex, but it can be thought of as a geometric graph. It is but one example of a straight-line planar graph defined on a set P of n points in the plane. Two other popular examples include the *Gabriel graph* and the *relative neighborhood graph* (or RNG). Let P be any set of points in the plane. As with the Delaunay triangulation, the points of P will be the vertices of graph.

In the Gabriel graph, denoted $\mathrm{GG}(P)$, two points p_i and p_j are joined by an edge if the disk whose diameter is p_ip_j contains no other points of P (see Fig. 33(b)).

In the RNG, denoted $\mathrm{RNG}(P)$, there is an edge joining p_i and p_j if there is no point $p_k \in P$ that is simultaneously closer to p_i and p_j than they are to each other. That is, there is no p_k such that

$$\max(\mathrm{dist}(p_i, p_k), \mathrm{dist}(p_j, p_k)) \;<\; \mathrm{dist}(p_i, p_j).$$

Given two points p_i and p_j, define their *lune* to be the "football shaped' region where the circle centered at p_i and passing through p_j intersects the circle centered at p_j and passing through p_i. The RNG edge condition is equivalent to saying that p_i and p_j are adjacent if and only if $\mathrm{lune}(p_i, p_j)$ contains no other points of P (see Fig. 33(c)).

We will leave it as an exercise to prove that these three geometric graphs form a nested hierarchy, that is, for any point set P, we have $\mathrm{RNG}(P) \subseteq \mathrm{GG}(P) \subseteq \mathrm{DT}(P)$.

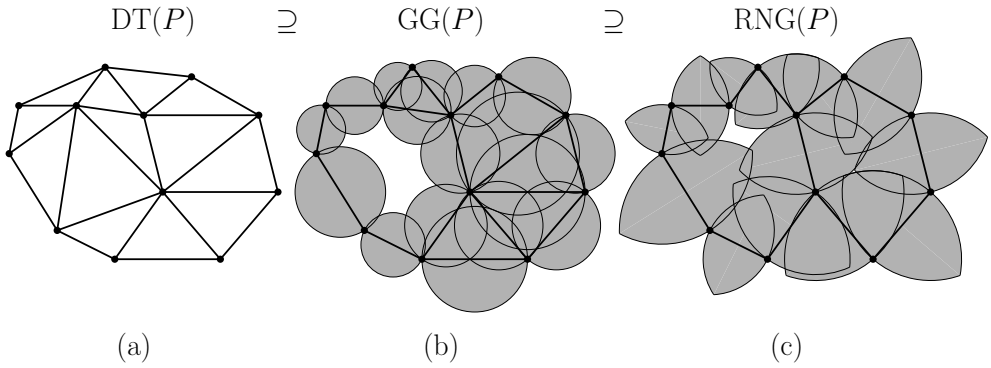


Fig. 33: The Delaunay triangulation (a), Gabriel graph (b), and the relative neighborhood graph (c).

Lecture 10: Geometric Data Structures for Games: Index Structures

Sources: Some of today's materials can be found in "Foundations of Multidimensional and Metric Data Structures," by H. Samet, 2006.

Index Structures: We continue our discussion of useful geometric data structure for computer games. So far we have discussed data structures for representing meshes (and more generally, cell complexes), geometric graphs (such as the visibility graph, θ-graph), and spatial subdivisions (such as the Delaunay triangulation). Usually, when we think of data structures for 1-dimensional data (such as hash maps, binary trees, and skip lists), the data structure is a means for efficiently accessing data based on a *key value*. A data structure that locates an object based on a key value is called an *index structure*.

In this lecture, we consider some useful index structures for storing spatial data sets. The topic is amazingly vast, and we will just provide a very short description of a few of the simplest spatial index structures. (See Samet's book above for more topics.)

Geometric Objects and Queries: What sorts of objects might be stored in a geometric data structure? The simplest object to store is a point, and it is common to introduce data structures under the assumption that the objects themselves are points. As needed, the data structure and associated algorithms can be enhanced to deal with more general types of 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. 34(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.) They are also very easy to intersect. (Just determine whether the intervals of x-coordinates intersect and the intervals of the y-coordinates intersect.)

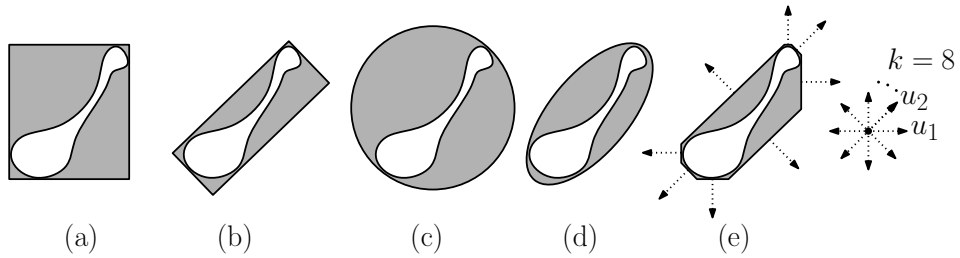


Fig. 34: Examples of common enclosures: (a) AABB, (b) general BB, (c) sphere, (d) ellipsoid, (e) 8-DOP.

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. 34(b)). Unfortunately, computing the minimum area or minimum volume bounding box is not a trivial problem. Determining whether two such boxes intersect is a more complex procedure, but it is likely to be much more efficient than computing the intersection of the two objects.

Bounding spheres: Given that arbitrary bounding boxes are harder to intersect, an alternative is to use bounding spheres (see Fig. 34(b)). Spheres are invariant under both translation and rotation. It is very easy to determine whether two spheres intersect one another. Just compute the distance between their centers, and check that this is smaller than the sum of their radii.

Bounding ellipsoids: The main problem with spheres (and problem that also exists with axis-parallel bounding boxes) is that skinny objects are no well approximated by a sphere. A ellipse (or generally, an ellipsoid in higher dimensions) is just the image of a sphere under a linear transformations (see Fig. 34(b)). 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.

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. 34(e)).

Hierarchies of bounding boxes: A natural generalization of the concept of a bounding box is that of constructing a hierarchy consisting of multiple levels of bounding boxes, where the boxes at a given level enclose a constant number of boxes at the next lower level. A common approach is to specify that the number of boxes contained within a given node lies between some minimum and maximum value. The result is called an *R-tree*. In Fig. 35 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).

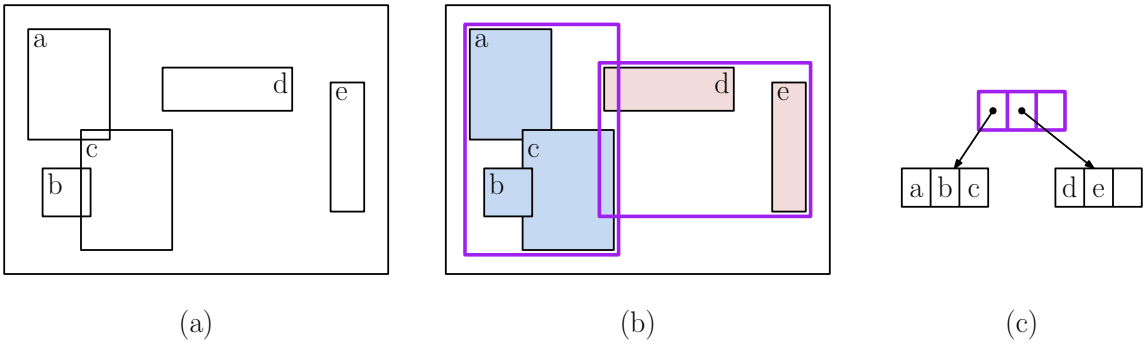


Fig. 35: A hierarchy of bounding boxes: (a) the input boxes, (b) the hierarchy of boxes, (c) the associated R-tree structure.

There are a number of messy 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 and how do you minimize the overlap between boxes at a given level of the tree.

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 B-tree data structure for 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. 36(a)). Compute the cells of the grid that contain these points (see Fig. 36(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. 36(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 compromose the data structure's correctness.

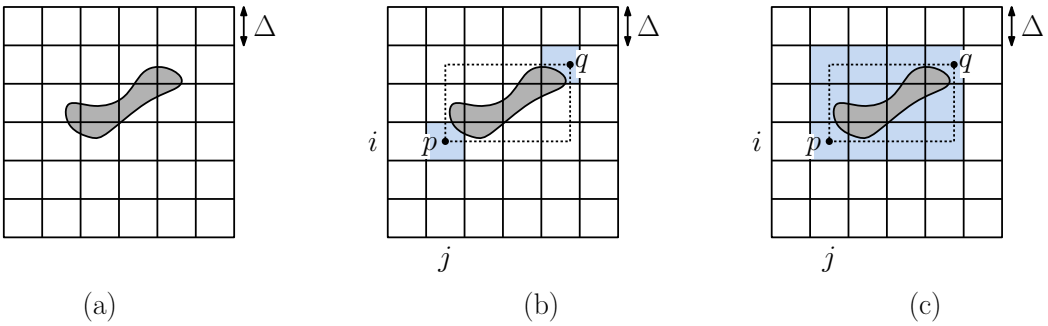


Fig. 36: Storing an object in a grid.

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, and 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. 37(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. 37(b) and (c). The first is called the *Hilbert curve* and the second is known as the *Morton order* (also called the *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. 37(c). Expressing i and j as 3-element bit vectors

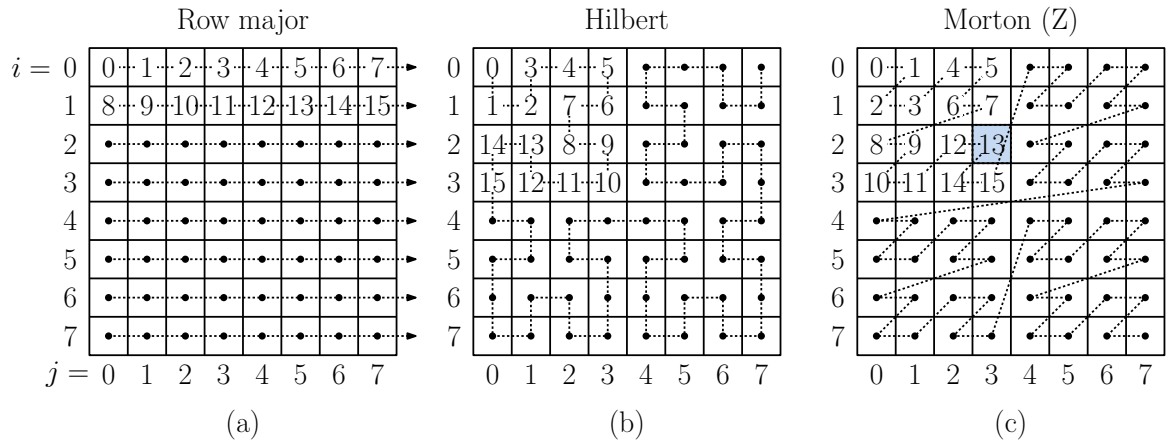


Fig. 37: Linear allocations to improve reference locality of neighboring cells: (a) row-major order, (b) the Hilbert curve, (c) the Morton order (or Z-order).

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.

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. 38). 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 $1/2^k$.

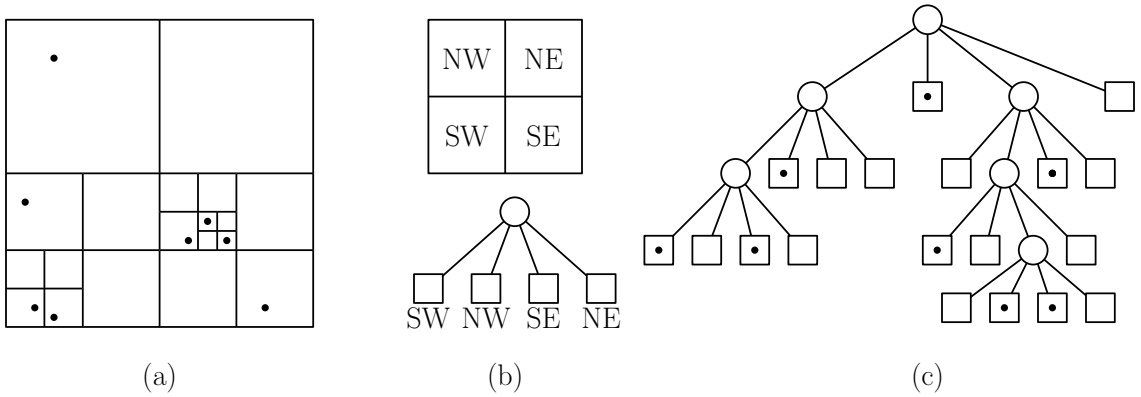


Fig. 38: A quadtree decomposition and the associated tree.

Here are a few tips to making quadtrees somewhat more manageable.

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. 39).

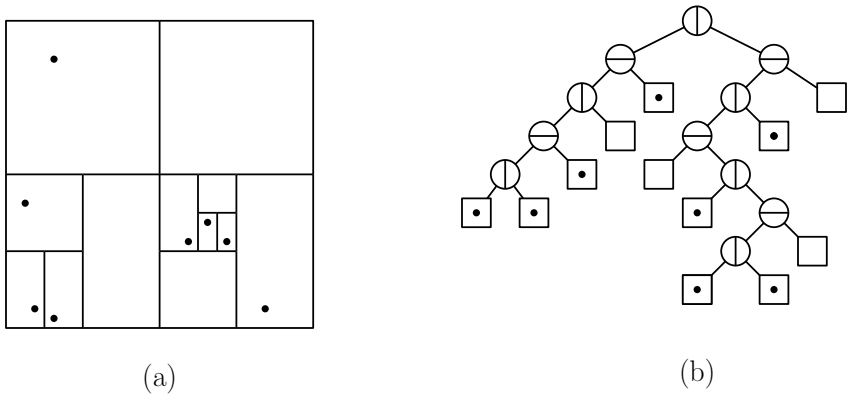


Fig. 39: A binary quadtree.

Compressed Quadtree: If the objects are very highly clustered, it is possible that the quadtree may actually much larger than the number of objects. Consider the case of two very close points in Fig. 40. Separating these two points would involve a long string of splits, most of which are *trivial* in the sense that each node has only one nonempty child. One way to deal with this is to compress out such trivial paths by a single edge that jumps, in one hop, to the first node that has two or more nonempty children. For example, you could introduce a new type of node, called a *compression node*, that has a single child, which points to the last node of the trivial path. The result is called a *compressed quadtree*. A nice feature of the compressed quadtree is that a compressed quadtree with n points has $O(n)$ nodes.

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

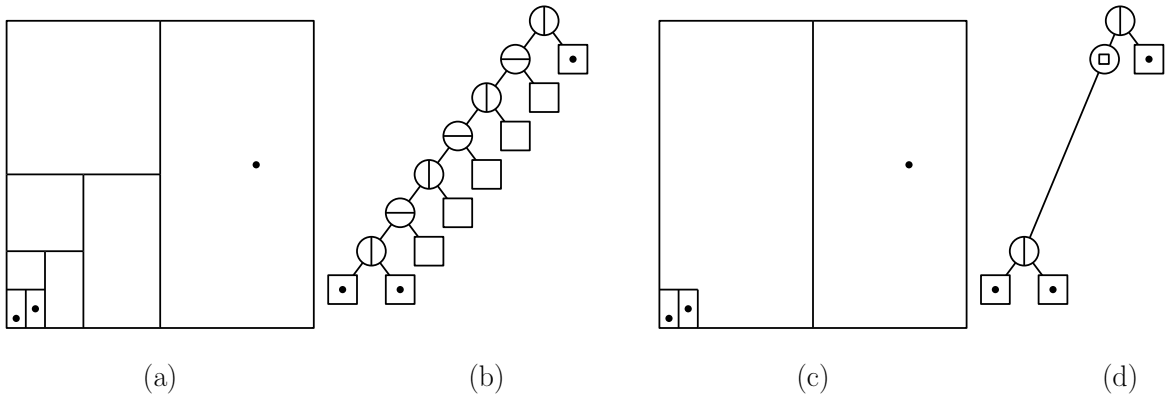


Fig. 40: A compressed quadtree.

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. 41).

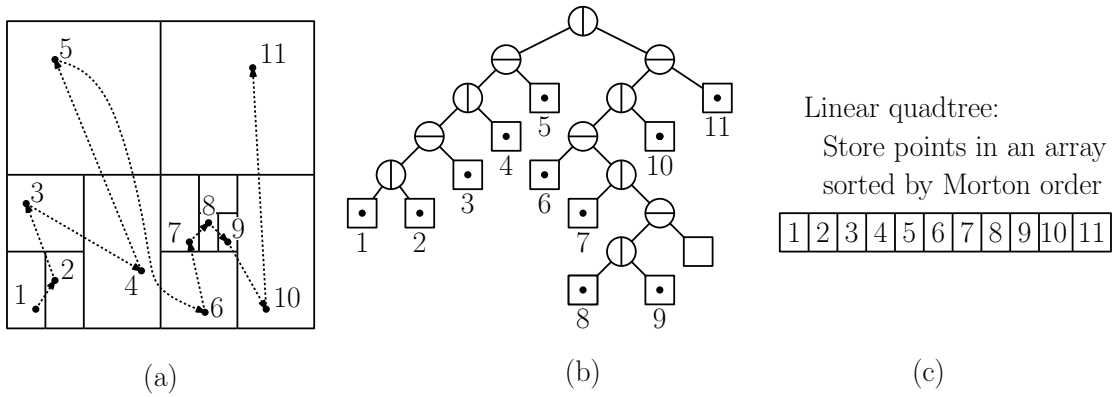


Fig. 41: A linear quadtree.

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.

Lecture 11: Light Modeling for Games

Sources: Chapt 10 of Gregory, *Game Engine Architecture*.

Lighting in Games: In order to produce a realistic rendering of 3-dimensional scene, we need to specify how the various elements of the scene are to be illuminated or shaded. Lighting and shading serve a number of roles in a computer game:

Illumination: Accurate illuminate can enhance realism and help encourage a feeling of immersion.

Revelation of form: Good lighting can be used to compensate for the lack of accuracy in a geometric models. (For example, adding a bumpy lighting pattern to a flat surface can suggest a stone road.)

Focus: Highlighting can be used to direct the viewer's attention to a desired region of focus.

Mood: Lighting can be used to set the tone of a scene (e.g., safe, threatening, foreboding, pastoral), which may be useful in establishing the narrative elements of a game. One of the simplest applications of this idea is the use of low-contrast versus high-contrast lighting (see Fig. 42):

High-Key Lighting: Bright, warm, soft, and high-set lights tend to provide a feeling of safety

Low-Key Lighting: Dim, cool, harsh, and low-set lights produce a feeling of danger

Fig. 42: High-key versus low-key lighting.

Local/Global Light Models: Lighting is a very complex phenomenon that involves the interaction of physical elements (photons, reflection and refraction, chromatic dispersion), physiological elements (color perception), and psychological elements (e.g., the feelings engendered by certain colors).

Basic OpenGL supports a very simple lighting and shading model, and hence can achieve only limited realism. This was done primarily because speed is of the essence in interactive graphics. OpenGL assumes a *local illumination model*, which means that the shading of a point depends *only* on its relationship to the light sources, without considering the other objects in the scene.

This is in contrast to a *global illumination model*, in which light reflected or passing through one object might affects the illumination of other objects. Global illumination models deal with many affects, such as shadows, indirect illumination, color bleeding (colors from one object reflecting and altering the color of a nearby object), caustics (which result when light passes through a lens and is focused on another surface).

For example, OpenGL's lighting model does not model shadows, it does not handle indirect reflection from other objects (where light bounces off of one object and illuminates another), it does not handle objects that reflect or refract light (like metal spheres and glass balls). OpenGL's light and shading model was designed to be very efficient. Although it is not physically realistic, the OpenGL designers provided many ways to "fake" realistic illumination models. Modern GPUs support programmable shaders, which offer even greater realism.

The Phong Lighting Model: A detailed discussion of light and its properties would take us more deeply into physics than we care to go. For our purposes, we can imagine a simple model of light consisting of a large number of photons being emitted continuously from each light source.

Each photon has an associated energy, which (when aggregated over millions of different reflected photons) we perceive as *color*. Although color is complex phenomenon, for our purposes it is sufficient to consider color to be a modeled as a triple of red, green, and blue components. On hitting a surface, one of three things can happen (see Fig. 43):

Reflection: The photon can be reflected or scattered back into the atmosphere. At a microscopic level, surfaces tend to be smooth (as with polished metals) or irregular (as with cloth). The former surfaces reflect light in a *specular* (or shiny) manner and surfaces of the latter type reflect light in a *diffuse* (or uniform) manner. We thus distinguish three different varieties of reflection:

Pure reflection: Perfect mirror-like reflectors

Specular reflection: Imperfect reflectors like brushed metal and shiny plastics

Diffuse reflection: Uniformly scattering, like cloth

Absorption: The photon can be absorbed into the surface (and hence dissipates in the form of heat energy). We do not see this light.

Transmission: The photon can pass through the surface. This happens perfectly with *transparent* objects (like glass and polished gem stones) and with a significant amount of scattering with *translucent* objects (like human skin or a thin piece of tissue paper).

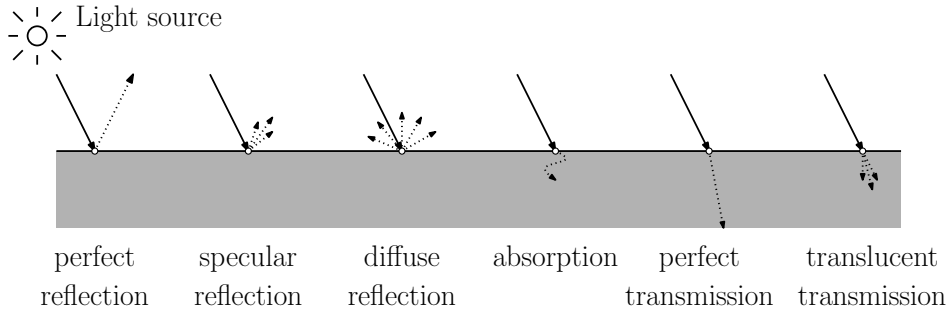


Fig. 43: The ways in which a photon of light can interact with a surface.

All of the above involve how incident light reacts with a surface. Another way that light may result from a surface is through emission, which will be discussed below.

Light Sources in OpenGL: OpenGL assumes that each light source is a *point*, and that the energy emitted can be modeled as an RGB triple, called a *luminance function*. This is described by a vector with three components $L = (L_r, L_g, L_b)$, which indicate the intensities of red, green, and blue light respectively. We will not concern ourselves with the exact units of measurement, since this is very simple model. Note that, although your display device will have an absolute upper limit on how much energy each color component of each pixel can generate (which is typically modeled as an 8-bit value in the range from 0 to 255), in theory there is no upper limit on the intensity of light.

Lighting in real environments usually involves a considerable amount of indirect reflection between objects of the scene. If we were to ignore this effect and simply consider a point to be illuminated only if it can see the light source, then the resulting image in which objects in the shadows are totally black. In indoor scenes we are accustomed to seeing much softer shading, so that even objects that are hidden from the light source are partially illuminated. In OpenGL (and most local illumination models) this scattering of light modeled by breaking the light source's intensity into two components: *ambient emission* and *point emission*.

Ambient emission: Refers to light that does not come from any particular location. Like heat, it is assumed to be scattered uniformly in all locations and directions. A point is illuminated by ambient emission even if it is not visible from the light source.

Point emission: Refers to light that originates from a single point. In theory, point emission only affects points that are directly visible to the light source. That is, a point p is illuminate by light source q if and only if the open line segment $\overline{pq}$ does not intersect any of the objects of the scene.

Unfortunately, determining whether a point is visible to a light source in a complex scene with thousands of objects can be computationally quite expensive. So OpenGL simply tests whether the surface is facing towards the light or away from the light. Surfaces in OpenGL are polygons, but let us consider this in a more general setting. Suppose that have a point p lying on some surface. Let n denote the normal vector at p, directed *outwards* from the object's interior, and let ℓ denote the directional vector from p to the light source ($\ell = q - p$), then p will be illuminated if and only if the angle between these vectors is acute. We can determine this by testing whether their dot produce is positive, that is, $n \cdot \ell > 0$. (The dot product btween two vectors is positive if and only if the angle between them is acute.)

For example, in the Fig. 44, the point p is illuminated. In spite of the obscuring triangle, point p' is also illuminated, because other objects in the scene are ignored by the local illumination model. The point p'' is clearly not illuminated, because its normal is directed away from the light.

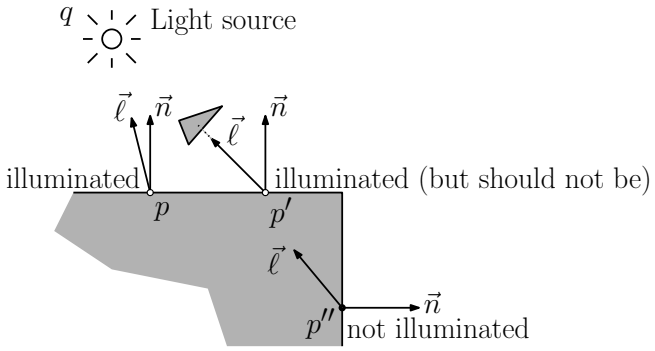


Fig. 44: Point light source visibility using a local illumination model. Note that p' is illuminated in spite of the obscuring triangle.

Attenuation: The light that is emitted from a point source is subject to *attenuation*, that is, the decrease in strength of illumination as the distance to the source increases. Physics tells us that the intensity of light falls off as the inverse square of the distance. This would imply that the intensity at some (unblocked) point p would be

$$I(p, q) = \frac{1}{\|p - q\|^2} I(q),$$

where $\|p - q\|$ denotes the Euclidean distance from p to q. However, in OpenGL, our various simplifying assumptions (ignoring indirect reflections, for example) will cause point sources to appear unnaturally dim using the exact physical model of attenuation. Consequently, OpenGL uses an attenuation function that has constant, linear, and quadratic components. The user specifies constants a, b and c. Let $d = \|p - q\|$ denote the distance to the point source. Then the attenuation function is

$$I(p, q) = \frac{1}{a + bd + cd^2} I(q).$$

In OpenGL, the default values are $a = 1$ and $b = c = 0$, so there is no attenuation by default.

Types of light reflection: The next issue needed to determine how objects appear is how this light is *reflected* off of the objects in the scene and reach the viewer. So the discussion shifts from the discussion of light sources to the discussion of object surface properties. We will assume that all objects are opaque. The simple model that we will use for describing the reflectance properties of objects is called the *Phong model*. The model is over 20

years old, and is based on modeling surface reflection as a combination of emission (glowing) and the ambient, diffuse, and specular reflection from the light sources in the scene.

Let $L = (L_r, L_g, L_b)$ denote the illumination intensity of the light source. OpenGL allows us to break this light's emitted intensity into three components: *ambient* L_a, *diffuse* L_d, and *specular* L_s. Each type of light component consists of the three color components, so, for example, $L_d = (L_{dr}, L_{dg}, L_{db})$, denotes the RGB vector (or more generally the RGBA components) of the diffuse component of light. As we have seen, modeling the ambient component separately is merely a convenience for modeling indirect reflection. The diffuse and specular intensities of a light source are usually set equal to each other.

An object's color determines how much of a given intensity is reflected. Let $C = (C_r, C_g, C_b)$ denote the object's color. These are assumed to be normalized to the interval $[0, 1]$. Thus we can think of C_r as the fraction of red light that is reflected from an object. Thus, if $C_r = 0$, then no red light is reflected. When light of intensity L hits an object of color C, the amount of reflected light is given by the *component-wise product* of the L and C vectors. Let us define $L \odot C$ to be this product, that is,

$$L \odot C = (L_r \cdot C_r, L_g \cdot C_g, L_b \cdot C_b).$$

For example, if the light is white $L = (1, 1, 1)$ and the color is red $C = (0.25, 0, 0)$ then the reflection is $L \odot C = (0.5, 0, 0)$ which is a dark shade of red. However if the light is blue $L = (0, 0, 1)$, then $L \odot C = (0, 0, 0)$, and hence the object appears to be black.

In OpenGL, rather than specifying a single color for an object (which indicates how much light is reflected for each component) you instead specify the amount of reflection for each type of illumination: C_a, C_d, and C_s. Each of these is an RGBA vector. This seems to be a rather extreme bit of generality, because, for example, it allows you to specify that an object can reflect only red light ambient light and only blue diffuse light. Although they can each be set separately, OpenGL provides a single command that sets both the ambient and diffuse components of reflection for an object.

What about the specular color? It is interesting to note that specular color is typically the color of the light source, not the color of the object. (Consider the shiny white spot on a black billiard ball.) For this reason, the components of the specular color of an object are typically set to $(1, 1, 1)$, meaning that the light's color is reflected perfectly.

Lighting and Shading in OpenGL: To describe lighting in OpenGL there are three major steps that need to be performed: setting the general parameters of the lighting and shading model, defining the lights (their positions, colors, and properties), and finally defining objects and specifying their material properties.

Lighting/Shading model: There are a number of global lighting parameters and options that can be set through the command glLightModel*(). It has two forms, one for scalar-valued parameters and one for vector-valued parameters.

```
glLightModelf(GLenum pname, GLfloat param);
glLightModelfv(GLenum pname, const GLfloat* params);
```

Perhaps the most important parameter is the global intensity of ambient light (independent of any light sources). Its pname is GL_LIGHT_MODEL_AMBIENT and params is a pointer to an RGBA vector.

One important issue is whether polygons are to be drawn using *flat shading*, in which every point in the polygon has the same shading, or *smooth shading*, in which shading varies across the surface by interpolating the vertex shading. This is set by the following command, whose argument is either GL_SMOOTH (the default) or GL_FLAT.

```
glShadeModel(GL_SMOOTH);    --OR--    glShadeModel(GL_FLAT);
```

In theory, shading interplation can be handled in one of two ways. In the classical *Gouraud interpolation* the illumination is computed exactly at the vertices (using the above formula) and the values are interpolated across the polygon. In *Phong interpolation*, the normal vectors are given at each vertex, and the system interpolates these vectors in the interior of the polygon. Then this interpolated normal vector is used in the above lighting equation. This produces more realistic images, but takes considerably more time. OpenGL uses Gouraud shading. Just before a vertex is given (with glVertex*()), you should specify its normal vertex (with glNormal*()).

The commands glLightModel and glShadeModel are usually invoked in your initializations.

Create/Enable lights: To use lighting in OpenGL, it must first be enabled. This is done by glEnable(GL_LIGHTING). OpenGL allows the user to create up to 8 light sources, named GL_LIGHT0 through GL_LIGHT7. Each light source may either be enabled (turned on) or disabled (turned off). By default they are all disabled. Again, this is done using glEnable() (and glDisable()). The properties of each light source is set by the command glLight*(). This command takes three arguments, the name of the light, the property of the light to set, and the value of this property.

Let us consider a light source 0, whose position is $(2, 4, 5)$. This would be presented to OpenGL in homogeneous coordinates. Recall that this means that we set the last component to 1. Thus, it would be $(2, 4, 5, 1)$ in homogeneous coordinates. Suppose we want this to generate a bright red ambient intensity, given as the RGB triple $(0.9, 0, 0)$, and white diffuse and specular intensities, given as the RGB triple $(1.2, 1.2, 1.2)$. (Normally all the intensities will be of the same color, albeit of different strengths. We have made them different just to emphasize that it is possible.) There are no real units of measurement involved here. Usually the values are adjusted manually by a designer until the image "looks good."

Light intensities are actually expressed in OpenGL as RGBA, rather than just RGB triples. The 'A' component can be used for various special effects, but for now, let us just assume the default situation by setting 'A' to 1. Here is an example of how to set up such a light in OpenGL. The procedure glLight*() can also be used for setting other light properties, such as attenuation.

Setting up a simple lighting situation

```
glClearColor(0.0, 1.0, 0.0, 1.0);           // intentionally background
glEnable(GL_NORMALIZE);                     // normalize normal vectors
glShadeModel(GL_SMOOTH);                    // do smooth shading
glEnable(GL_LIGHTING);                      // enable lighting
                                            // ambient light (red)
GLfloat ambientIntensity[4] = {0.9, 0.0, 0.0, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientIntensity);

                                            // set up light 0 properties
GLfloat lt0Intensity[4] = {1.5, 1.5, 1.5, 1.0};  // white
glLightfv(GL_LIGHT0, GL_DIFFUSE, lt0Intensity);
glLightfv(GL_LIGHT0, GL_SPECULAR, lt0Intensity);

GLfloat lt0Position[4] = {2.0, 4.0, 5.0, 1.0};   // location
glLightfv(GL_LIGHT0, GL_POSITION, lt0Position);
                                           // attenuation params (a,b,c)
glLightf (GL_LIGHT0, GL_CONSTANT_ATTENUATION,  0.0);
glLightf (GL_LIGHT0, GL_LINEAR_ATTENUATION,    0.0);
glLightf (GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.1);
glEnable(GL_LIGHT0);
```

Defining Surface Materials (Colors): When lighting is in effect, rather than specifying colors using glColor() you do so by setting the material properties of the objects to be rendered. OpenGL computes the color based on the lights and these properties. Surface properties are assigned to vertices (and not assigned to faces as you might

think). In smooth shading, this vertex information (for colors and normals) are interpolated across the entire face. In flat shading the information for the first vertex determines the color of the entire face.

Every object in OpenGL is a polygon, and in general every face can be colored in two different ways. In most graphic scenes, polygons are used to bound the faces of solid polyhedra objects and hence are only to be seen from one side, called the *front face*. This is the side from which the vertices are given in counterclockwise order. By default OpenGL, only applies lighting equations to the front side of each polygon and the back side is drawn in exactly the same way. If in your application you want to be able to view polygons from both sides, it is possible to change this default (using glLightModel() so that each side of each face is colored and shaded independently of the other. We will assume the default situation.

Surface material properties are specified by glMaterialf() and glMaterialfv().

```
glMaterialf(GLenum face, GLenum pname, GLfloat param);
glMaterialfv(GLenum face, GLenum pname, const GLfloat *params);
```

It is possible to color the front and back faces separately. The first argument indicates which face we are coloring (GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK). The second argument indicates the parameter name (GL_EMISSION, GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_SHININESS). The last parameter is the value (either scalar or vector). See the OpenGL documentation for more information.

Typical drawing with lighting

```
GLfloat color[]  = {0.0, 0.0, 1.0, 1.0};    // blue
GLfloat white[]  = {1.0, 1.0, 1.0, 1.0};    // white
                                            // set object colors
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, color);
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 100);

glPushMatrix();
    glTranslatef(...);                      // your transformations
    glRotatef(...);
    glBegin(GL_POLYGON);                    // draw your shape
        glNormal3f(...); glVertex(...);     // remember to add normals
        glNormal3f(...); glVertex(...);
        glNormal3f(...); glVertex(...);
    glEnd();
glPopMatrix();
```

Recall from the Phong model that each surface is associated with a single color and various coefficients are provided to determine the strength of each type of reflection: emission, ambient, diffuse, and specular. In OpenGL, these two elements are combined into a single vector given as an RGB or RGBA value. For example, in the traditional Phong model, a red object might have a RGB color of $(1, 0, 0)$ and a diffuse coefficient of 0.5. In OpenGL, you would just set the diffuse material to $(0.5, 0, 0)$. This allows objects to reflect different colors of ambient and diffuse light (although I know of no physical situation in which this arises).

Other options: You may want to enable a number of GL options using glEnable(). This procedure takes a single argument, which is the name of the option. To turn each option off, you can use glDisable(). These optional include:

GL_CULL_FACE: Recall that each polygon has two sides, and typically you know that for your scene, it is impossible that a polygon can only be seen from its back side. For example, if you draw a cube with six square faces, and you know that the viewer is outside the cube, then the viewer will never see the back sides of the walls of the cube. There is no need for OpenGL to attempt to draw them. This can often save a

factor of 2 in rendering time, since (on average) one expects about half as many polygons to face towards the viewer as to face away.

Backface culling is the process by which faces which face away from the viewer (the dot product of the normal and view vector is negative) are not drawn.

By the way, OpenGL actually allows you to specify which face (back or front) that you would like to have culled. This is done with glCullFace() where the argument is either GL_FRONT or GL_BACK (the latter being the default).

GL_NORMALIZE: Recall that normal vectors are used in shading computations. You supply these normal to OpenGL. These are assumed to be normalized to unit length in the Phong model. Enabling this option causes all normal vectors to be normalized to unit length automatically. If you know that your normal vectors are of unit length, then you will not need this. It is provided as a convenience, to save you from having to do this extra work.

Tips: Until you have finished debugging your program, I would suggest *disabling* backface culling (in case you accidentally generate the elements of your mesh in an improper CW/CCW orientation). I would also suggest *enabling* normalization of normal vectors.

Lecture 12: Texture Mapping

Surface Detail: We have discussed the use of lighting as a method of producing more realistic images. This is fine for smooth surfaces of uniform color (plaster walls, plastic cups, metallic objects), but many of the objects that we want to render have some complex surface finish that we would like to model. In theory, it is possible to try to model objects with complex surface finishes through extremely detailed models (e.g. modeling the cover of a book on a character by character basis) or to define some sort of regular mathematical texture function (e.g. a checkerboard or modeling bricks in a wall). But this may be infeasible for very complex unpredictable textures.

Textures and Texture Space: Although originally designed for textured surfaces, the process of *texture mapping* can be used to map (or "wrap") any digitized image onto a surface. For example, suppose that we want to render a picture of the Mona Lisa, or wrap an image of the earth around a sphere, or draw a grassy texture on a soccer field. We could download a digitized photograph of the texture, and then map this image onto surface as part of the rendering process.

There are a number of common image formats which we might use. We will not discuss these formats. Instead, we will think of an image simply as a 2-dimensional array of RGB values. Let us assume for simplicity that the image is square, of dimensions $n \times n$ (OpenGL requires that n is a power of 2 for its internal representation. If you image is not of this size, you can pad it out with unused additional rows and columns.) Images are typically indexed row by row with the upper left corner as the origin. The individual RGB pixel values of the texture image are often called *texels*, short for *texture elements*.

Rather than thinking of the image as being stored in an array, it will be a little more elegant to think of the image as function that maps a point (s, t) in 2-dimensional *texture space* to an RGB value (see Fig. 45). That is, given any pair (s, t), $0 \le s, t < 1$, the texture image defines the value of $T(s, t)$ is an RGB value. Note that the interval $[0, 1)$ does not depend on the size of the images. This has the advantage an image of a different size can be substituted without the need of modifying the wrapping process.

For example, if we assume that our image array $I[n][n]$ is indexed by row and column from 0 to $n - 1$ with (as is common with images) the origin in the upper left corner. Our texture space $T(s, t)$ has two axes, labeled s (horizontal) and t (vertical), such that the origin is in the lower-left corner. We could then apply the following function to round a point in image space to the corresponding array element:

$$T(s,t) = I\big[\,\lfloor (1-t)n \rfloor\,\big]\big[\,\lfloor sn \rfloor\,\big], \quad \text{for } 0 \le s, t < 1.$$

In many cases, it is convenient to think of the texture is an infinite function. We do this by imagining that the texture image is repeated cyclically throughout the plane. This is useful when applying a small repeating texture

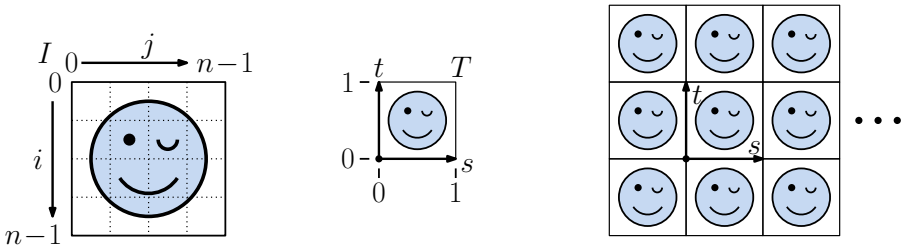


Fig. 45: Texture space.

to a very large surface (like a brick pattern on a wall). This is sometimes called a *repeated texture*. In this case we can modify the above function to be

$$T(s,t) = I\big[\,\lfloor(1-t)n\rfloor \bmod n\big]\big[\,\lfloor sn\rfloor \bmod n\big], \quad \text{for any } s,t.$$

Inverse Wrapping Function and Parametrization: Suppose that we wish to "wrap" a 2-dimensional texture image onto the surface of a 3-dimensional ball of unit radius, that is, a unit sphere. We need to define a wrapping function that achieves this. The surface resides in 3-dimensional space, so the wrapping function would need to map a point (s,t) in texture space to the corresponding point (x,y,z) in 3-space. That is, the wrapping function can be thought of as a function $W(s,t)$ that maps a point in 2-dimensional texture space to a point (x,y,z) in three dimensional space.

Later we will see that it is not the wrapping function that we need to compute, but rather its inverse. So, let us instead consider the problem of computing a function W^{-1} that maps a point (x,y,z) on the sphere to a point (s,t) in parameter space. This is called the *inverse wrapping function*.

This is typically done by first computing a 2-dimensional *parametrization* of the surface. This means that we associate each point on the object surface with two coordinates (u,v) in *surface space*. Implicitly, we can think of this as three functions, $x(u,v)$, $y(u,v)$ and $z(u,v)$, which map the parameter pair (u,v) to the x,y,z-coordinates of the corresponding surface point.

Our approach to solving the inverse wrapping problem will be to map a point (x,y,z) to the corresponding parameter pair (u,v), and then map this parameter pair to the desired point (s,t) in texture space. At the end of this lecture, we present an example of how to derive a parametrization for a sphere.

Texture Mapping in OpenGL: Recall that all objects in OpenGL are rendered as polygons or generally meshes of polygons. This simplifies the texture mapping process because it means that we need only provide the inverse wrapping function for the vertices, and we can rely on simple interpolation to fill in the polygon's interior. For example, suppose that a triangle is being drawn. When the vertices of the polygon are given, the user also specifies the corresponding (s,t) coordinates of these points in texture space. These are called the vertices' *texture coordinates*. This implicitly defines the inverse wrapping function from the surface of the polygon to a point in texture space.

As with surface normals (which were used for lighting computations) texture vertices are specified *before* each vertex is drawn. For example, a texture-mapped object in 3-space with shading might be drawn using the following general form, where $\vec{n} = (n_x, n_y, n_z)$ is the surface normal, (s,t) are the texture coordinates, and $p = (p_x, p_y, p_z)$ is the vertex position.

```
glBegin(GL_POLYGON);
  glNormal3f(nx, ny, nz); glTexCoord2f(s, t); glVertex3f(px, py, pz);
  // ...
glEnd();
```

Interpolating Texture Coordinates: Given the texture coordinates, the next question is how to interpolate the texture coordinates for the points in the interior of the polygon. An obvious approach is to first project the vertices of the triangle onto the viewport. This gives us three points p_0, p_1, and p_2 for the vertices of the triangle in 2-space. Let q_0, q_1 and q_2 denote the three texture coordinates, corresponding to these points. The simplest approach is to apply a simple linear interpolation procedure, mapping each point of the $\triangle p_0 p_1 p_2$ to its corresponding point in $\triangle q_0 q_1 q_2$.

What is wrong with this simple approach? The first problem has to do with perspective. The problem is that projective transformations are not linear transformations, but a linear interpolation is. An illustration of what can go wrong is shown in Fig. 46(c).

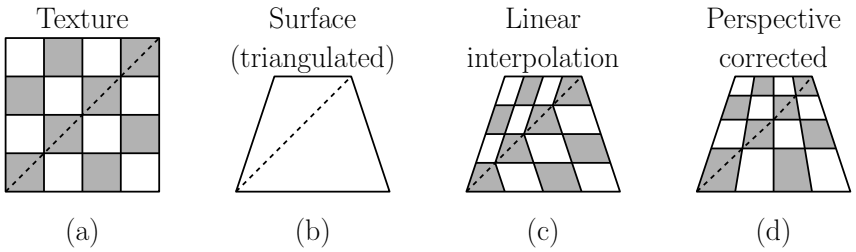


Fig. 46: Perspective correction.

There are a number of ways to fix this problem. One approach is to use a more complex formula for interpolation, which corrects for perspective distortion. (See the text for details. An example of the result is shown in Fig. 46(d)). This can be activated using the following OpenGL command:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

(The other possible choice is GL_FASTEST, which does simple linear interpolation.)

The other method involves slicing the polygon up into sufficiently small polygonal pieces, such that within each piece the amount of distortion due to perspective is small. Recall that with Gauroud shading, it was often necessary to subdivide large polygons into small pieces for accurate lighting computation. If you have already done this, then the perspective distortion due to texture mapping may not be a significant issue for you.

Aliasing and mipmapping: A second problem with the aforementioned simple approach to interpolating texture coordinates has to do with something called *aliasing*. After determining the region of texture space onto which the pixel projects, we should blend the colors of the texels in this region to determine the color of the pixel. The problem with the above procedure does no blending, it determines the color on the basis of just a single texel. In situations where the pixel corresponds to a point in the distance and hence covers a large region in texture space, this may produce very strange looking results, because the color of the entire pixel is determined entirely by a single point in texture space that happens to correspond to the pixel's center coordinates (see Fig. 47(a)).

Dealing with aliasing in general is a deep topic, which is a main issue in the general field of signal processing. The process of blending samples together to smooth over the effects of aliasing is called *filtering*. In the field of signal processing there are many techniques (some quite sophisticated) for filtering in general. OpenGL applies a very simple method for dealing with the aliasing of this sort. The method is called *mipmapping*. (The acronym "mip" comes from the Latin phrase *multum in parvo*, meaning "much in little.")

The idea behind mipmapping is to generate a series of texture images at decreasing levels of resolution. For example, if you originally started with a 128×128 image, a mipmap would consist of this image along with a 64×64 image, a 32×32 image, a 16×16 image, etc. All of these are scaled copies of the original image. Each pixel of the 64×64 image represents the average of a 2×2 block of the original. Each pixel of the 32×32 image represents the average of a 4×4 block of the original, and so on (see Fig. 48).

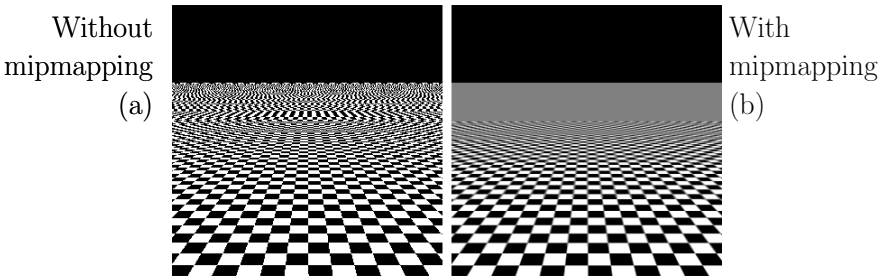


Fig. 47: Aliasing and mipmapping.

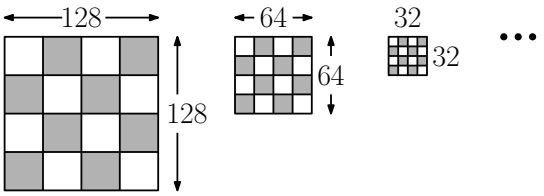


Fig. 48: Mipmapping.

Now, when OpenGL needs to apply texture mapping to a screen pixel that overlaps many texture pixels (that is, when "minimizing" the texture), it determines the mipmap in the hierarchy that is at the closest level of resolution, and uses the corresponding averaged pixel value from that mipmapped image. This results in more nicely blended images (see Fig. 48(b)).

Warning: If you use mipmapping in OpenGL (and this is not only a good idea, but the default for texture minimization), you *must* construct mipmaps for your texture. This can be done automatically for you by the command gluBuild2DMipmaps. (We will not discuss this command explicitly, but an example is given below.) If you fail to build mipmaps when they are needed, OpenGL will not produce an error message, it will simply disable texture mapping.

Texture mapping in OpenGL: OpenGL supports a fairly general mechanism for texture mapping. The process involves a bewildering number of different options. You are referred to the OpenGL documentation for more detailed information. By default, objects are not texture mapped. If you want your objects to be colored using texture mapping, you need to enable texture mapping before you draw them. This is done with the following command.

```
glEnable(GL_TEXTURE_2D);
```

After drawing textured objects, you can disable texture mapping.

If you plan to use more than one texture, then you will need to request that OpenGL generate texture objects. This is done with the following command:

```
glGenTextures(GLsizei n, GLuint* textureIDs);
```

This requests that n new texture objects be created. The n new texture id's are stored as unsigned integers in the array $textureIDs$. Each texture ID is an integer greater than 0. (Typically, these are just integers 1 through n, but OpenGL does not require this.) If you want to generate just one new texture, set $n = 1$ and pass it the address of the unsigned int that will hold the texture id.

By default, most of the texture commands apply to the "active" texture. How do we specify which texture object is the active one? This is done by a process called *binding*, and is defined by the following OpenGL command:

```
glBindTexture(GLenum target, GLuint textureID);
```

where $target$ is one of GL_TEXTURE_1D, GL_TEXTURE_2D, or GL_TEXTURE_3D, and $textureID$ is one of the texture IDs returned by glGenTextures. The $target$ will be GL_TEXTURE_2D for the sorts of 2-dimensional image textures we have been discussing so far. The $textureID$ parameter indicates which of the texture IDs will become the active texture. If this texture is being bound for the first time, a new texture object is created and assigned the given texture ID. If $textureID$ has been used before, the texture object with this ID becomes the active texture.

Presenting your Texture to OpenGL: The next thing that you need to do is to input your texture and present it to OpenGL in a format that it can access efficiently. It would be nice if you could just point OpenGL to an image file and have it convert it into its own internal format, but OpenGL does not provide this capability. You need to input your image file into an array of RGB (or possibly RGBA) values, one byte per color component (e.g. three bytes per pixel), stored row by row, from upper left to lower right. By the way, OpenGL requires images whose height and widths are powers of 2.

Once the image array has been input, you need to present the texture array to OpenGL, so it can be converted to its internal format. This is done by the following procedure. There are many different options, which are partially explained in below.

```
glTexImage2d(GL_TEXTURE_2D, level, internalFormat, width, height,
  border, format, type, image);
```

The procedure has an incredible array of options. Here is a simple example to present OpenGL an RGB image stored in the array myPixelArray. The image is to be stored with an internal format involving three components (RGB) per pixel. It is of width $nCols$ and height $nRows$. It has no border ($border = 0$), and we are storing the highest level resolution. (Other levels of resolution are used to implement the averaging process, through a method called *mipmaps*.) Typically, the level parameter will be 0 ($level = 0$). The format of the data that we will provide is RGB (GL_RGB) and the type of each element is an unsigned byte (GL_UNSIGNED_BYE). So the final call might look like the following:

```
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

In this instance, your array $myPixelArray$ is an array of size $256 \times 512 \times 3 = 393,216$ whose elements are the RGB values, expressed as unsigned bytes, for the 256×512 texture array. An example of a typical texture initialization is shown in the code block below. (The call to gluBuild2DMipmaps is needed only if mipmapping is to be used.)

Initialization for a Single Texture

```
GLuint textureID;                     // the ID of this texture
glGenTextures(1, &textureID);         // assign texture ID
glBindTexture(GL_TEXTURE_2D, textureID); // make this the active texture
//
// ... input image nRows x nCols into RGB array myPixelArray
//
glTexImage2d(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
                                      // generate mipmaps - very important!
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, nCols, nRows, GL_RGB,
  GL_UNSIGNED_BYTE, myPixelArray);
```

Texturing Options: Once the image has been input and presented to OpenGL, we need to tell OpenGL how it is to be mapped onto the surface. Again, OpenGL provides a large number of different methods to map a surface. These parameters are set using the following function:

```
glTexParameteri(target, param_name, param_value);
glTexParameterf(target, param_name, param_value);
```

The first form is for integer parameter values and the second is for float values. For most options, the argument $target$ will be set to GL_TEXTURE_2D. There are two common parameters to set. First, in order to specify that a texture should be repeated or not (clamped), the following options are useful.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

These options determine what happens of the s parameter of the texture coordinate is less than 0 or greater than 1. (If this never happens, then you don't need to worry about this option.) The first causes the texture to be displayed only once. In particular, values of s that are negative are treated as if they are 0, and values of s exceeding 1 are treated as if they are 1. The second causes the texture to be wrapped-around repeatedly, by taking the value of s modulo 1. Thus, $s = 5.234$ amd $s = 76.234$ are both equivalent to $s = 0.234$. This can independently be set for he t parameter of the texture coordinate, by setting GL_TEXTURE_WRAP_T.

Filtering and Mipmapping: Another useful parameter determines how rounding is performed during *magnification* (when a screen pixel is smaller than the corresponding texture pixel) and "*minification*" (when a screen pixel is larger than the corresponding texture pixel). The simplest, but not the best looking, option in each case is to just use the *nearest pixel* in the texture:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

A better approach is to use linear filtering when magnifying and mipmaps when minifying. An example is given below.

Combining Texture with Lighting: How are texture colors combined with object colors? The two most common options are GL_REPLACE, which simply makes the color of the pixel equal to the color of the texture, and GL_MODULATE (the default), which makes the colors of the pixel the product of the color of the pixel (without texture mapping) times the color of the texture. The former is for painting textures that are already *prelit*, meaning that lighting has already been applied. Examples include skyboxes and precomputed lighting for the ceiling and walls of a room. The latter is used when texturing objects to which lighting is to be applied, such as the clothing of a moving character.

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

Drawing a Texture Mapped Object: Once the initializations are complete, you are ready to start drawing. First, bind the desired texture (that is, make it the active texture), set the texture parameters, and enable texturing. Then start drawing your textured objects. For each vertex drawn, be sure to specify the texture coordinates associated with this vertex, prior to issuing the glVertex command. If lighting is enabled, you should also provide the surface normal. A generic example is shown in the code block below.

```
glEnable(GL_TEXTURE_2D);              // enable texturing
glBindTexture(GL_TEXTURE_2D, textureID); // select the active texture
                                      // (use GL_REPLACE below for skyboxes)
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
                                      // repeat texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
                                      // reasonable filter choices
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
glBegin(GL_POLYGON);                  // draw the object(s)
  glNormal3f( ... );                  // set the surface normal
  glTexCoord2f( ... );                // set texture coords
  glVertex3f( ... );                  // draw vertex
  // ... (repeat for other vertices)
glEnd()
glDisable(GL_TEXTURE_2D);             // disable texturing
```

Parametrization of a Sphere (Optional): Let's make this more concrete with an example. Our shape is a unit sphere centered at the origin. We want to find the inverse wrapping function W^{-1} that maps any point (x, y, z) on the of the sphere to a point (s, t) in texture space.

We first need to come up with a surface parametrization for the sphere. We can represent any point on the sphere with two angles, representing the point's latitude and longitude. We will use a slightly different approach. Any point on the sphere can be expressed by two angles, φ and θ, which are sometimes called *spherical coordinates*. (These will take the roles of the parameters u and v mentioned above.)

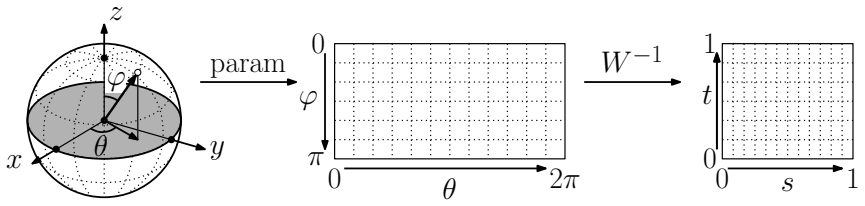


Fig. 49: Parametrization of a sphere.

Consider a vector from the origin to the desired point on the sphere. Let φ denote the angle in radians between this vector and the z-axis (north pole). So φ is related to, but not equal to, the latitude. We have $0 \leq \varphi \leq \pi$. Let θ denote the counterclockwise angle of the projection of this vector onto the xy-plane. Thus $0 \leq \theta < 2\pi$. (This is illustrated in Fig. 49.)

Our next task is to determine how to convert a point (x, y, z) on the sphere to a pair (θ, φ). It will be a bit easier to approach this problem in the reverse direction, by determining the (x, y, z) value that corresponds to a given parameter pair (θ, φ).

The z-coordinate is just $\cos\varphi$, and clearly this ranges from 1 to -1 as φ increases from 0 to π. To determine the value of θ, let us consider the projection of this vector onto the x, y-plane. Since the vertical component is of length $\cos\varphi$, and the overall length is 1 (since it's a unit sphere), by the Pythagorean theorem the horizontal length is $\ell = \sqrt{1 - \cos^2\varphi} = \sin\varphi$. The lengths of the projections onto the x and y coordinate axes are $x = \ell\cos\theta$ and $y = \ell\sin\theta$. Putting this all together, it follows that the (x, y, z) coordinates corresponding to

the spherical coordinates (θ, φ) are

$$z(\varphi, \theta) \;=\; \cos\varphi, \qquad x(\varphi, \theta) \;=\; \sin\varphi \cdot \cos\theta, \qquad y(\varphi, \theta) \;=\; \sin\varphi \cdot \sin\theta.$$

But what we wanted to know was how to map (x, y, z) to (θ, φ). To do this, observe first that $\varphi = \arccos z$. It appears at first that θ will be much messier, but there is an easy way to get its value. Observe that $y/x = \sin\theta/\cos\theta = \tan\theta$. Therefore, $\theta = \arctan(y/x)$. In summary:

$$\varphi \;=\; \arccos z \qquad \theta \;=\; \arctan(y/x),$$

(Remember that this can be computed accurately as $\mathsf{atan2}(y, x)$.)

The final step is to map the parameter pair (θ, φ) to a point in (s, t) space. To get the s coordinate, we just scale θ from the range $[0, 2\pi]$ to $[0, 1]$. Thus, $s = \theta/(2\pi)$.

The value of t is trickier. The value of φ increases from 0 at the north pole π at the south pole, but the value of t decreases from 1 at the north pole to 0 at the south pole. After a bit of playing around with the scale factors, we find that $t = 1 - (\varphi/\pi)$. Thus, as φ goes from 0 to π, this function goes from 1 down to 0, which is just what we want. In summary, the desired inverse wrapping function is $W^{-1}(x, y, z) = (s, t)$, where:

$$\begin{aligned} s &= \frac{\theta}{2\pi}, \quad \text{where} \quad \theta = \arctan\frac{y}{x} \\ t &= 1 - \frac{\varphi}{\pi}, \quad \text{where} \quad \varphi = \arccos z. \end{aligned}$$

Note that at the north and south poles there is a singularity in the sense that we cannot derive a unique value for θ. This is phenomenon is well known to cartographers. (What is the longitude of the north or south pole?)

To summarize, the *inverse wrapping* function $W^{-1}(x, y, z)$ maps a point on the surface to a point (s, t) in texture space. This is often done through a two step process, first determining the parameter values (u, v) associated with this point, and then mapping (u, v) to texture-space coordinates (s, t).This "unwrapping" function, maps the surface back to the texture. For this simple example, let's just set this function to the identity, that is, $W^{-1}(u, v) = (u, v)$. In general, we may want to stretch, translate, or rotate the texture to achieve the exact placement we desire.

Lecture 13: Animation for Games: Basics

Sources: Chapt 11 of Gregory, *Game Engine Architecture*.

Game Animation: Most computer games revolve around *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 (how to jump onto a platform), athletic technique (how a basketball player performs a lay-up shot or a warrior hurls a spear), stylistic choices (how a dancer moves), or simply the arbitrary conventions of everyday life (how a person steps out of a car or how a dog wags its tail). Producing natural looking animation is the topic of our next few lectures.

Image-based Animation: Most character animation arising in high-end 3-dimensional computer games is based on skeletal animation. Prior to this, most animation was based on two-dimensional methods.

Cel Animation: Before the advent of computers, the standard method of producing animation in films was by a process where an artist would paint each individual frame of a characters animation on a transparent piece of celluloid, which would then be placed over a static background. The term celluloid was shortened to *cel*.

Given the need to generate 30 or 60 frames per second, this is a very labor-intensive process. Often a master animator would sketch a sparse set of *key drawings* or *key frames* showing the most important transitions, and then less experienced artists would add color and fill in intermediate frames. The process of inserting intermediate drawings in between the key frames acquired the name of *tweening*. This approach has been extended to computer animation, where the modeling software generates a complete representation of certain key frames and some type of *interpolation* is then applied to generate the intermediate frames.

Sprites: An early adaptation of the cel approach to computer games is called *sprite animation*, where a series of bit-map images is generated (see Fig. 50(a)). Sprites were typically generated to form a cycle so that the sequence could be played repeatedly to generate the illusion of walking or running.

(a) (b)

Fig. 50: Image-based animation: (a) sprites and (b) a control mesh for morphing.

Animated Textures: A simple extension of sprites, used even in modern games for low resolution models, is to model a moving object by a sequence of images, which are played back in a loop. This could be used, for example, to show trees swaying in the distance or background crowd motion in a sports arena.

Texture Morphing: Storing a large number of large animated textures can be wasteful in space. One way of saving space is to store a small number of key images and transition smoothly between them. In computer graphics, *morphing* refers to the process of continuously mapping one image into another through a process of gradual distortion. This is typically done by selecting a small set of corresponding *control points* on each of the two images. For example, to morph between two faces these might consist of the corners of the mouth, nose, eyes, ears, chin, and hairline. Next, a pair of compatible *control meshes* is generated, one for each image (see Fig. 50(b)). (A good choice would be a Delaunay triangulation, as discussed in Lecture 9.) Then, the animation process involves linearly interpolating between these two images.

Skeletal Models: As mentioned above, 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. 51). The animation is performed by modifying the relationships between pairs of adjacent joints, for example, by altering joint angles. We will discuss this process extensively later.

A skeletal model is based on a hierarchical representation where peripheral elements are linked as children of more central elements. Here is an example:

```
Root (whole model)
.   Pelvis
.   .   .   Spine
.   .   .   .   .   Neck
.   .   .   .   .   .   Head
.   .   .   .   .   Right shoulder
.   .   .   .   .   .   Right elbow
.   .   .   .   .   .   .   .   Right hand
.   .   .   .   .   Left shoulder
.   .   .   .   .   .   Left elbow
```

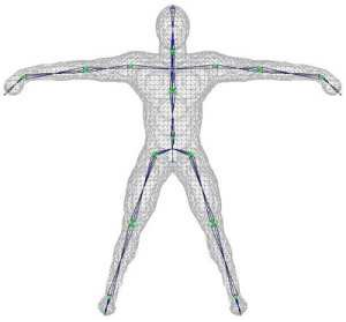


Fig. 51: Skeletal model.

. . . . Left hand
. . . . Right thigh
. . . . . Right knee
. . . . . . Right ankle
. . . . Left thigh
. . . . . Left knee
. . . . . . Left ankle

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.

Joint representation: At a minimum, we need to be able to perform the following operations on any skeleton tree:

enumerate$()$**:** Iterate through all the joints in the skeleton. In some implementations this might need to be done according to a particular tree traversal (preorder or postorder). For the algorithms we will discuss later, the order of enumeration will not be significant.

isRoot(j)**:** Return true if j is the root node and false otherwise.

parent(j)**:** For a given non-root node j, return its parent, which we denote by $p(j)$.

Depending on the details of the algorithms involved, it may also be useful to access a joint's children, but we will not need this. Given the above requirements, the tree can be represented very simply as an array of triples, where each triple consists of:

- a *name* for the joint (which will be useful for debugging),
- a structure holding the joint's *internal information* (which we describe below), and
- the *index* of the joint's parent in this array.

We can make the convention of storing the root joint in index 0 of this array, and therefore we can detect the root node easily. Of course, this is just one possible representation.

What does the joint's "internal information" consist of? Each joint can be thought of as defining its own *joint coordinate frame* (see Fig. 52(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). These coordinate frames are organized hierarchically according to the structure of the skeleton (see Fig. 52(b)). Rotating a joint can be achieved by rotating 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 descendent joints (elbow, hand, fingers, etc.) also move as a result (see Fig. 52(c)).

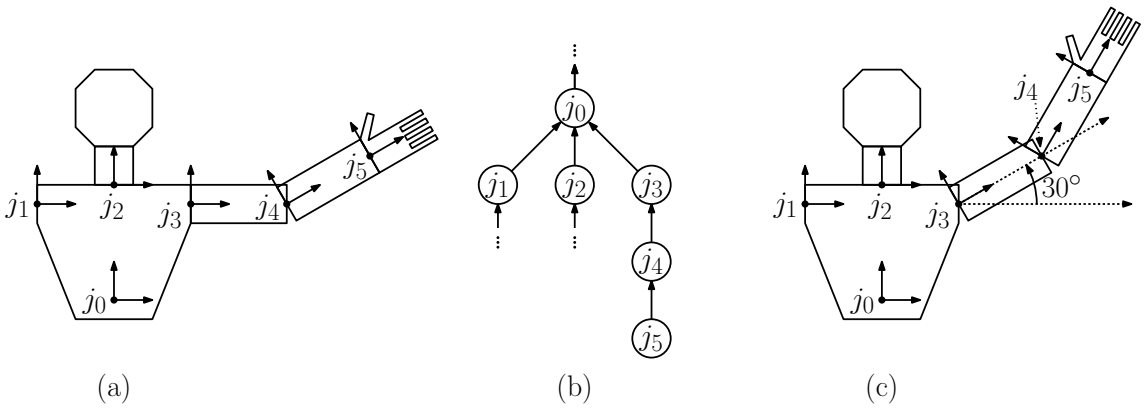


Fig. 52: Skeletal model.

Clearly, in order to achieve this effect, we need to know the relationships between the various joints of the system. A basic fact from affine geometry is that, given any two coordinate frames in d-dimensional space, it is possible to convert a point 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 a $(d+1)\times(d+1)$ matrix. Let's imagine that we are working in 3-dimensional space. For any two joints j and k, let $T_{[k\leftarrow j]}$ denote the change of coordinates transformation that maps a point in joint j's coordinate system to its representation in k's coordinate system. 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 v's representation relative to k's coordinate frame.

Given any non-root joint j, recall that $p(j)$ denotes its parent joint. Let M denote the root of the tree, which is associated with the entire skeletal model. We are interested in the following transformations:

Local-pose transformation: $T_{[p(j)\leftarrow j]}$ converts a point in j's coordinate system to its representation in its parent's coordinate system.

Inverse local-pose transformation: $T_{[j\leftarrow p(j)]}$ converts a point in j's parent's coordinate system to j's coordinate system. Clearly, $T_{[j\leftarrow p(j)]} = (T_{[p(j)\leftarrow j]})^{-1}$.

Which of these transformation matrices is associated with the joint? It turns out to be neither. We shall see that there is yet another matrix that is more important. (Well, there is another matrix more important at least for the purposes of the skin rendering algorithm that we shall describe later. You may have your own reasons for storing one or both of these matrices, since they are certainly relevant to the local processing of the skeleton.)

Storing local-pose transformations: If you wanted to store these matrices, how would you do so? As mentioned above, they can be represented as matrices, either a 3×3 for 2-dimensional animations or a 4×4 for 3-dimensional animations. You may prefer to use an alternative representation, depending on your application. For example, if you working in 2-dimensional space, then the transformation $T_{[j\leftarrow p(j)]}$ most likely consists of two components, a rotation angle θ, that aligns the two x-axes, and a translation vector v_t from $p(j)$'s origin to j's origin. This is a more concise representation. Storing a 3×3 matrix involves storing 9 floats, whereas storing the pair (θ, v_t) involves only 3 floats. Given this concise representation, it is easy to derive the inverse as the pair $(-\theta, -v_t)$. Also, it is easy to derive the associated matrices from this representation. (We will skip this easy exercise in affine geometry, since we haven't discussed affine geometry in detail.)

Is there a similarly concise representation for the local-pose transformation in 3-dimensional space? The answer is yes, but the issue of rotation is more complicated. Again, in order to represent $T_{[j\leftarrow p(j)]}$, you need to store a 3-dimensional rotation and a translation. The translation vector v_t can be stored as a 3-dimensional vector. However, how do you represent rotations in three dimensional space? There are two common methods:

Euler angles: It turns out that *any* rotation in 3-dimensional space can be expressed as the composition of three rotations, one about the x-axis, one about the y-axis, and one about the z-axis. These are sometimes referred to as *pitch*, *roll*, and *yaw* (see Fig. 53).

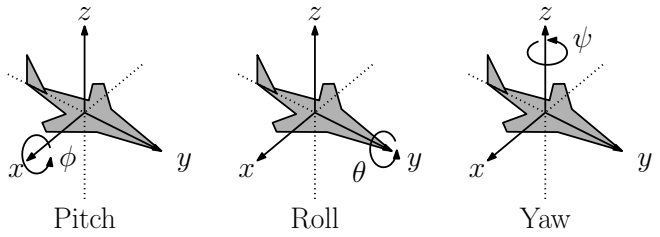


Fig. 53: Rotations by Euler angles.

Quaternions: While Euler angles are easy for humans to work with, they are not the most mathematically elegant manner of representing a rotation. Why this is so is rather hard to explain, but as an example of why, consider the question of how to interpolate between two Euler angle rotations. Suppose that one is represented by the rotations $(\phi_0, \theta_0, \psi_0)$ and the other by $(\phi_1, \theta_1, \psi_1)$. In order to generate a smooth interpolation between these two rotations, you would naturally consider performing a linear interpolation between them. For example, the midpoint rotation would be given by $\left(\frac{\phi_0+\phi_1}{2}, \frac{\theta_0+\theta_1}{2}, \frac{\psi_0+\psi_1}{2}\right)$. If you would do so (and trust me on this), you would find this midpoint rotation would not appear to look anything close to the initial or final rotations.

There is, however, a mathematically clean solution to the representation of 3-dimensional angles, which is called a *quaternion*. This is a mathematical object that was discovered around 150 years ago by the Irish mathematician Hamilton. It would take too long to explain how quaternions work, but in a nutshell, a quaternion represents a 3-dimensional rotation by a 4-element vector. (Note that this is less space efficient than Euler angles, which require only 3 numbers, but the 4-element vector is of unit length, so it is possible to derive the 4th component from the first 3.) There is a multiplication operator defined on quaternions, and it is possible to compose two rotations by multiplying the associated quaternions.

One nice feature of quaternions is that, unlike Euler angles, they interpolate nicely. Because a quaternion is a unit vector, you need to be sure to use spherical interpolation (that is, interpolating along great circle arcs) rather than linear (straight-line interpolation). The good news is that, without understanding the mathematical details of quaternions, it is possible to download software that will do all of this for you.

So, the upshot of this is that you can store the local pose transformation using either 6 or 7 floats (a rotation stored as 3 Euler angles or 4 quaternion coefficients) and a 3-element translation vector. As in the 2-dimensional case, there are easy procedures for computing the inverse transformations from either representation.

Bind Pose: Before discussing animation on skeletal structures, it is useful to first say a bit about the notion of a pose. In a typical skeleton, joints move by means of rotations. (It is rather interesting to think about how this happens for your own joints. For example, your shoulder joint as 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.)

Assigning angles to the various joints of a skeleton (and, more generally, specifying the local pose transformations for each joint) defines the skeleton's exact position, that is, its *pose*. When an 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*. 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. 51 above). There are two important change-of-coordinate transformations for this process.

Bind-pose transformation: (also called the *global pose transformation*) $T_{[M \leftarrow j]}$ converts a point in j's coordinate system to its representation in the model's global reference coordinate system M.

We can express the matrix associated with this transformation as the product of the matrices associated with the local pose transformations from j up to the root. In particular, suppose that the path from j to the root is $j = j_1 \rightarrow j_2 \rightarrow \ldots \rightarrow j_m = M$, then (assuming post-multiplication, as OpenGL does) the bind-pose transformation is

$$T_{[M \leftarrow j]} \;=\; T_{[j_m \leftarrow j_{m-1}]} \ldots T_{[j_3 \leftarrow j_2]} T_{[j_2 \leftarrow j_1]} \;=\; \prod_{i=1}^{m-1} T_{[j_{m-i-1} \leftarrow j_{m-i}]}.$$

This transformation is important since, if we know of a point's position relative to its nearest joint (for example, a point on the tip of the index finger), this transformation allows us to convert this point into its representation in the model's global coordinate frame. In spite of this obvious usefulness, we shall see that the inverse of this transformation, given next, is even more important.

Inverse bind-pose transformation: $T_{[j \leftarrow M]}$ converts a point in the model's global coordinate system to its representation relative to j's local coordinate system. Clearly, $T_{[j \leftarrow M]} = (T_{[M \leftarrow j]})^{-1}$. Equivalently,

$$T_{[j \leftarrow M]} \;=\; T_{[j_1 \leftarrow j_2]} T_{[j_2 \leftarrow j_3]} \ldots T_{[j_{m-1} \leftarrow j_m]} \;=\; \prod_{i=1}^{m-1} T_{[j_i \leftarrow j_{i+1}]}.$$

When an artist places the skin on an object, he/she is implicitly associating a triangulated mesh with the skeleton. The modeling software knows the coordinates of the vertices of this mesh with respect to the model's global coordinate system. However, in order to know how each vertex will respond to changes in joint angles, we will need to map each vertex of the mesh into its representation local to one or more nearby joints. Next time, we will discuss the process of how to render skin on an animated skeleton. We shall see that the most relevant of all the transformations associated with any joint is the *inverse bind-pose transformation*, which indicates how to map a vertex from the global frame down to a local frame. So, we can now answer the question raised earlier, "What does the joint's internal information consist of?" The answer is the inverse bind-pose transformation. Next time, we'll see exactly why.

Lecture 14: Animation for Games: Skeletal Animation

Sources: Chapt 11 of Gregory, *Game Engine Architecture*.

Recap: Last time we introduced the principal elements of skeletal models. Today we will discuss methods for animating these models. We will also discuss the issue of how to cover these models with "skin," and how to move the skin smoothly as part of the animation.

Recall that a skeletal model consists of a collection of joints, which have been joined into a rooted tree structure. Recall that a *pose* is defined by rotating each of the joints in a specific way. Rotations are defined relative to a default pose, called the *bind pose* or *reference pose*. The mesh defining the character's skin is defined relative to this pose. (Typically this is the character standing upright with arms outstretched to the sides.) Each node j of this tree is (either implicitly or explicitly) associated with a the following relevant transformations:

Local-pose transformation: $T_{[p(j) \leftarrow j]}$ converts a point in j's coordinate system to its representation in its parent's coordinate system. Its inverse, $T_{[j \leftarrow p(j)]}$, converts from the parent's frame to the joint's frame.

Bind-pose transformation: (also called the *global pose transformation*) $T_{[M \leftarrow j]}$ converts a point in j's coordinate system to its representation in the model's global reference coordinate system M. It can be computed as the product of the local-pose transformations from j up to the root. Its inverse, $T_{[j \leftarrow M]}$, converts from the model's frame to the joint's frame.

Animating the Model: There are two natural ways to specify a skeletal's model pose:

Forward kinematics: You specify the position of the model (both its location and rotation) and the angles by which the joints are to be rotated. The system then places the model at the desired position and applies these rotations, which fixes the positions of all the model's joints.

Inverse kinematics: The problem with forward kinematics is that forces the designer to determine which joint angles achieve the constraints that he/she wants to impose. For example, if we know that the object is to have both feet at certain positions on the ground and is reaching with its right hand for a door knob, shouldn't the system be able to determine the joint angles for us?

In *inverse kinematics*, the designer provides the system with a set of constraints that are to be satisfied, and the system's job is to determine a set of joint angles so that the resulting pose achieves these constraints. Typically, these constraints take the form of specifying where the endpoints (that is, leaves of the skeleton tree) are to be placed, and it determines the intermediate joint angles to achieve these constraints.

Inverse kinematics is clearly a more convenient option for a designer, and many high-end solid modelers provide the feature. In general, inverse kinematics is much more challenging to implement than forward kinematics. There are many reasons for this. First, the inverse kinematics problem is simply a more complicated mathematical problem to solve than forward kinematics. Forward kinematics systems require little more than the ability to perform matrix multiplications. The inverse involves solving a nonlinear constrained optimization problem. Another reason is that a typical system of constraints is *under-specified*, which means that there may be multiple (often infinitely many) valid solutions. An inverse kinematics system would either need to establish which of these solutions is most "natural" (whatever that means), or it will need to request further information from the designer to add additional constraints. For example, the system may assume that joint angles that are close to their reference-pose values are of lower energy than those that involve higher turning angles, and it will attempt to find a pose that satisfies all the constraints and is of the lowest possible energy.

By the way, there is a third way of obtaining joint angles in animation. This is through the process of *motion capture*. 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.)

Let us make the simplifying assumption that our poses and animations are specified using forward kinematics. In order to specify an animation, we need to specify how the joint angles change over time. At the most general abstract level, we can think of each local-pose transformation, $T_{[p(j)\leftarrow j]}$, as being a function of time. For any given time $t \geq 0$, this function would tell us how to transform a point from j's local coordinate frame to its parent's frame. If we knew these functions for all the joints of our system, at any time t we could apply matrix multiplication to compute the bind pose transformation, $T_{[M\leftarrow j]}$. From this, we know how any point defined in j's local frame is positioned relative to the model's frame.

Unfortunately, this abstract view is far to general lend itself to practical implementation. There are many approaches that can be applied to reduce the complexity to an acceptable level:

Sample: Rather than storing the animation in functional form, we will compute a collection of *discrete samples* at specific times. We can think of these as corresponding to the key frames of a key-frame animation. We then use some form of interpolation to smoothly fill the gapes between successive key frames.

Eliminate redundancy: Storing a full 4×4 linear transformation can be space inefficient. If we know that a joint can rotate only about a single fixed axis, then we need only store the angle of rotation.

Since different joints have different rotation characteristics, is there a general way to handle this? 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. 54). Between the two, the full spectrum of two-dimensional rotation can be covered.

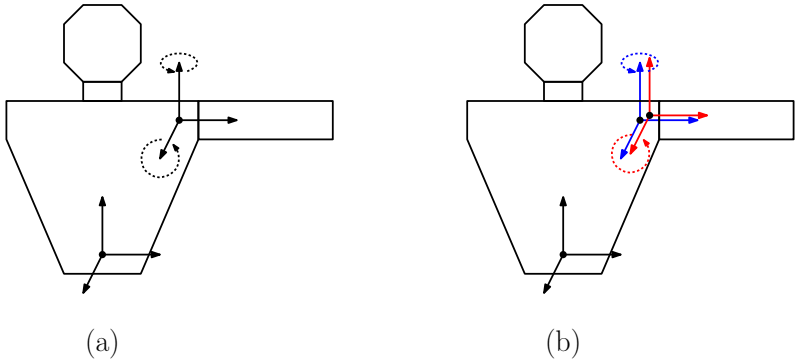


Fig. 54: Using two meta-joints (b) to simulate a single joint with two degrees of freedom.

Compress: If movements are small, the difference of joint angles over successive key frames may be very small. If so, it is not necessary to use all 32 bits of a floating-point value to store the rotation angle. Instead, it may suffice to store a low-precision integer that indicates the number of degrees of rotation. Expressed in terms of information theory, smoth motions have low information content, that is, *low entropy*. There are many data compression techniques that have been developed in the fields of signal processing and information theory for storing low-entropy streams efficiently.

Simple Animation Representation: There are a number of different file formats that are used for describing animations. These are based on various combinations of the above ideas. In Fig. 55 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) and a 4-element quaternion vector (s, t, u, v) to represent the rotation.

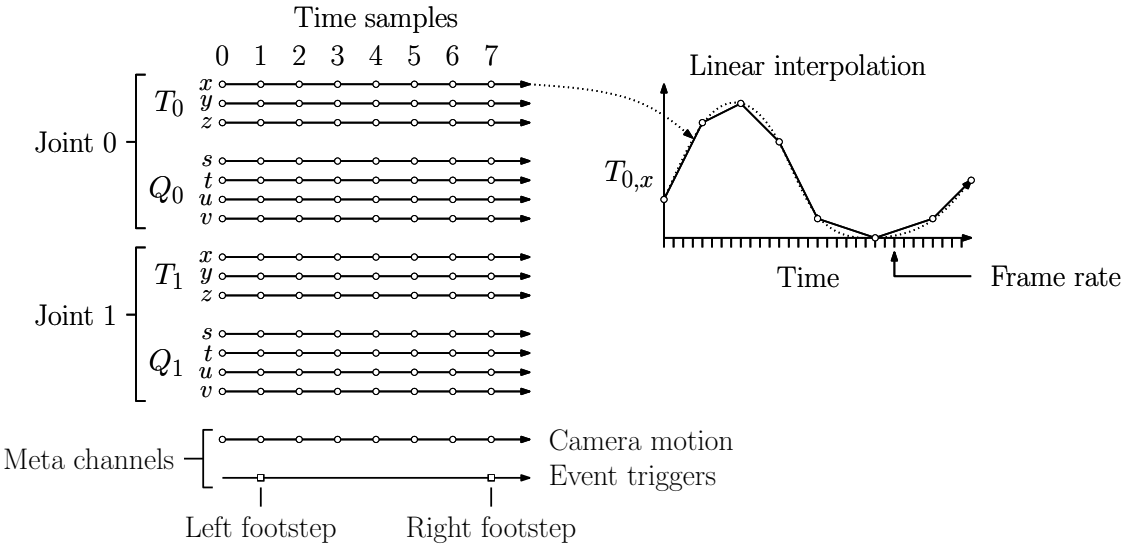


Fig. 55: An uncompressed animation stream.

The clip is discretized by taking samples at equally spaced intervals. (The interval size is generally longer than

the frame rate.) We may then interpolate linearly between consecutive samples at the frame rate of the display to obtain the transformation at any given point in time.

If the sampling is sufficiently dense, and the motion is sufficiently smooth, then linear interpolation works well enough. If you want to save more space by sampling more sparsely, you may find it useful to consider more sophisticated methods of sampling.

Spline reconstruction: There are methods in geometric modeling for extracting a smooth curve from a set of sample points. These systems go under various names, Bezier curves, B-splines, Hermite splines.

Spherical interpolation: Some quantities, such as quaternions, are by definition, vectors of unit length. When you interpolate linearly between two vectors of unit length, the result is a vector that is of less than unit length. (The linear interpolant travels along the straight line curve between the point, not along the great circle of the unit sphere as it should.) There are efficient computational methods for performing *spherical interpolation*. We will not discuss these, but just so you are aware of standard jargon, *lerp* means linear interpolation and *slerp* means spherical interpolation.

Auxiliary Information: It is sometimes useful to add further decorations to your animation, which are not necessarily related to the rendering of the moving character. Examples include:

Event triggers: 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. 55). This information will be interpreted by the game engine.

Lecture 15: Animation for Games: Animation and Skinning

Sources: Chapt 11 of Gregory, *Game Engine Architecture*.

Recap: In the last couple of lectures we introduced the principal elements of skeletal models and how to represent animations. Recall that a skeletal model consists of a collection of joints, which have been joined into a rooted tree structure. Recall that a *pose* is defined by rotating each of the joints in a specific way. Rotations are defined relative to a default pose, called the *bind pose* or *reference pose*. The mesh defining the character's skin is defined relative to this pose. (Typically this is the character standing upright with arms outstretched to the sides.) Each node j of this tree is (either implicitly or explicitly) associated with a the following relevant transformations:

Local-pose transformation: $T_{[p(j)\leftarrow j]}$ converts a point in j's coordinate system to its representation in its parent's coordinate system. Its inverse, $T_{[j\leftarrow p(j)]}$, converts from the parent's frame to the joint's frame.

Bind-pose transformation: (also called the *global pose transformation*) $T_{[M\leftarrow j]}$ converts a point in j's coordinate system to its representation in the model's global reference coordinate system M. It can be computed as the product of the local-pose transformations from j up to the root. Its inverse, $T_{[j\leftarrow M]}$, converts from the model's frame to the joint's frame.

Animation by forward kinematics: We assume that an animation is represented by a function that maps time to joint angles, or more generally, the transformations that relate one joint to another. This function is represented by means of a collection of *channels*, each of which stores a sequence of sampled transformations, one per joint. Through the use of various interpolation methods (linear or spherical, piecewise or smooth) we can determine the exact joint-to-joint transformation at any desired time instant.

Skinning: 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. 56).

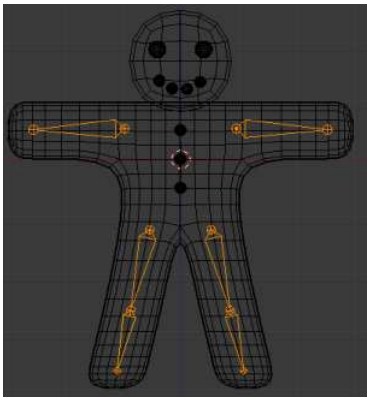


Fig. 56: Binding skin to a skeletal model in the reference pose.

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.)

As we mentioned earlier, if we were to bind each vertex to a single joint, then we would observe *cracks* appearing in our skin whenever neighboring vertices are bound to two different joints that are rotated apart from one another. To overcome this, we 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. Thus, 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.

For example, if you were to imagine the vertices of a forearm mesh, those that lie close to the elbow might be bound to the shoulder, elbow, and wrist joints, but with the elbow having the highest weight. As we move down the forearm towards the wrist, the vertex weights for the shoulder would vanish, the weights for the elbow would decrease, but the weights for the wrist would increase, and we would also see weight factors for the fingers to become positive as well. 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*, $R_j^{[t\leftarrow 0]}$, that specifies how joint j has rotated. For example, if the joint has undergone a rotation through an angle θ about some axis, then $R_j^{[t\leftarrow 0]}$ 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 $R_j^{[t\leftarrow 0]}$ 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 $v_j^{(t)} = R_j^{[t\leftarrow 0]} v_j^{(0)}$. (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 $T_{[M\leftarrow j]}^{[t\leftarrow 0]}$ 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 $T_{[M\leftarrow j]}^{[t\leftarrow 0]}$. 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 $R_j^{[t\leftarrow 0]}$ and then $T_{[p(j)\leftarrow j]}$. Let us define $T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]}$ to be the product $T_{[p(j)\leftarrow j]} R_j^{[t\leftarrow 0]}$. We now have

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

An Example: These matrix relations can be rather confusing, so let us consider a simple example to make this a bit more concrete. Consider the pair of joints shown in Fig. 57(a), where joint j lies 5 units to the right of its parent $p(j)$.

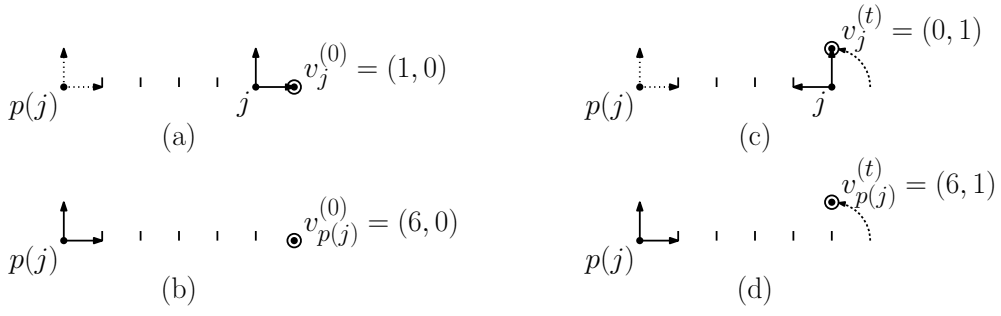


Fig. 57: Mapping a joint to its parent's frame. (a) and (b): before rotating the joint and (c) and (d): after rotating the joint.

It follows that $T_{[p(j)\leftarrow j]}(x,y) = (x+5, y)$. We can express this in the form of homogeneous matrices as follows.

Recall that the homogeneous representation of (x, y) is $[x, y, 1]$. Thus, we have

$$T_{[p(j)\leftarrow j]} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Let's consider the point lying at the tip of the x-axis in j's coordinate frame in the reference model. That is, $v_j^{(0)} = (1, 0)$. This point's representation relative to $p(j)$'s coordinate system is

$$v_{p(j)}^{(0)} = T_{[p(j)\leftarrow j]} v_j^{(0)} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 1 \end{bmatrix},$$

which is the point $v_{p(j)}^{(0)} = (6, 0)$, relative to $p(j)$'s coordinate system at time 0, as expected (see Fig. 57(b)).

Suppose that between time 0 and t we rotate the joint j by $90°$ counterclockwise (see Fig. 57(c)). The associated rotation matrix maps a point (x, y) to $(-y, x)$. This is represented by the following rotation matrix.

$$R_j^{[t\leftarrow 0]} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Since $v_j^{(0)} = (1, 0)$, after rotation it is mapped (relative to j's unrotated coordinate system) to

$$v_j^{(t)} = R_j^{[t\leftarrow 0]} v_j^{(0)} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix},$$

which is the point $(0, 1)$ at the tip of the y-axis as expected (see Fig. 57(c)).

Finally, let's consider the combination of these two steps. We want to know where the vertex v is mapped to at time t by in $p(j)$'s coordinate frame. To obtain this we compose the two transformations by multiplying the corresponding matrices

$$T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]} = T_{[p(j)\leftarrow j]} R_j^{[t\leftarrow 0]} = \begin{bmatrix} 1 & 0 & 5 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Therefore, we can obtain v's representation at time t relative to $p(j)$'s coordinate frame, that is, $v_{p(j)}^{(t)}$, by applying this transformation to $v_j^{(0)}$. That is,

$$v_{p(j)}^{(t)} = T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]} v_j^{(0)} = \begin{bmatrix} 0 & -1 & 5 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix},$$

which is the point $(5, 1)$, just as we would expect (see Fig. 57(d)).

The Current-Pose Transformation: Hopefully, you now believe that $T_{[p(j)\leftarrow j]}^{[t\leftarrow 0]}$ maps a vertex from j's frame at time 0 to $p(j)$'s frame at time t, accounting for the rotation of joint j. To obtain the position of a vertex associated with j's coordinate frame, we need only compose these matrices working back to the root of the tree. 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

$$T_{[M\leftarrow j]}^{[t\leftarrow 0]} = T_{[j_m\leftarrow j_{m-1}]}^{[t\leftarrow 0]} \cdots T_{[j_3\leftarrow j_2]}^{[t\leftarrow 0]} T_{[j_2\leftarrow j_1]}^{[t\leftarrow 0]} = \prod_{i=1}^{m-1} T_{[j_{m-i-1}\leftarrow j_{m-i}]}^{[t\leftarrow 0]}.$$

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. For the sake of making our notation more concise, we'll refer to it henceforth as $C_{[M \leftarrow j]}$. 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 $C_{[M \leftarrow j]}$ 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).

Blended Skinning: Next, we consider how to compute the positions of blended vertices. We assume that for the current-pose transformation $C_{[M \leftarrow j]}$ 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 2 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. Recall that this transformation is defined on the reference model, 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 $C_{[M \leftarrow j]}$ to map it to its current position relative to the model frame. If v was bound to a single joint j, we would have

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

Let us define $K_j = C_{[M \leftarrow j]} T_{[j \leftarrow M]}$. This is called the *skinning transformation* for joint j. (Note that this needs to be recomputed each time the rotation angles change.)

Now, we can generalize this to the general 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

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

A simple example of this is shown in Fig. 58. In Fig. 58(a) we show the reference pose. In Fig. 58(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. 58(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.

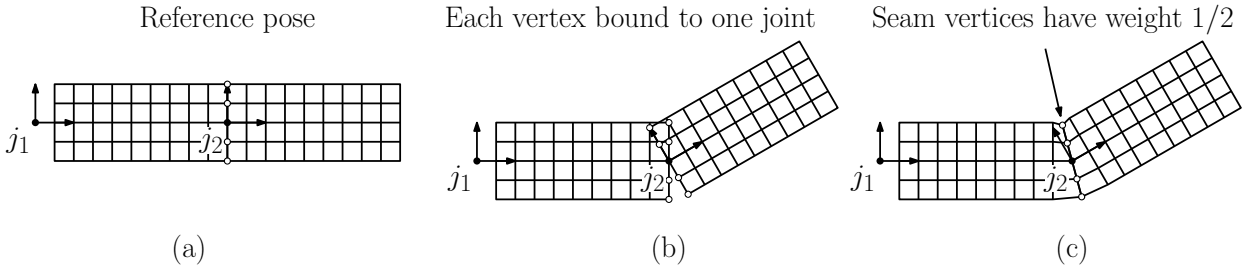


Fig. 58: A simple example of blended skinning.

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 vertex for each joint of the skeleton, namely the skinning matrices K_j. Recall that a skeleton will have perhaps tens of joints, while 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. 59).

Fig. 59: Shortcomings of vertex blending in skinning: (a) Collapsing due to bending and (b) collapsing due to twisting.

Lecture 16: Artificial Intelligence for Games: Basics

Sources: Some of the material from today's lecture comes from the book "Artificial Intelligence for Games" (2nd Edition) by I. Millington and J. Funge.

What is Artificial Intelligence? *Artificial intelligence* (AI) can be defined (circularly) as "the study of computational systems that exhibit intelligence." Unfortunately, it is not easy to define what we mean by "intelligence." In the context of games, an in particular in the design of non-player characters (NPCs) a working definition might be, "It is whatever a person would do." (Where, of course, the word "person" might be replaced by "ogre," "zombie," or "enchanted unicorn," whatever makes sense for the current context.)

At a basic level, game entities have *goals* that they are expected to achieve (e.g., staying out of danger, pursuing the enemy, fighting). This leads to a view of AI as planning strategies to achieve these goals. Computing optimal ways of achieving these goals may involve to *optimization algorithms* at a low level and lead to complex *planning strategies* at a high level. Often, in games, AI is most evident in games when it fails, that is, when other rationally behaving characters behave in an inexplicably unintelligent manner. (For example, it may fail to find a path around an obstacle, when such a path is visually obvious.)

Roles of Game AI: Generally, AI is used in games is to determine complex behaviors that not specified by the player. Examples include:

Nonplayer Opponents: For example, in an FPS, opponents should exhibit realistic attack behavior, which might include a decreased level or aggression or even retreating when suffering damage. This

Nonplayer Teammates: For example, given a squadron of soldiers, the group should move in a coordinated supportive manner. Such support NPCs are sometimes employed in multiplayer online games to assist inexperienced players. In some contexts, this might be scripted by the game designer. In others, the motion would be computed through the use of AI.

Support and Autonomous Characters: This includes generating realistic crowd behavior, where the characters may need to interact in a realistic manner when coming into contact with the player's character.

Commentary/Instruction: Again, this is typically scripted, but an example requiring AI might involve determining whether the player is stuck and in need of a hint on how to proceed.

Camera Control: Typically camera control is simply automatic, but in a complex environment it may require intelligence to compute a good viewpoint, from which it is possible to identify the important elements of the environment and is not obscured by obstacles.

The key element in all of these examples is the feature of *complexity*. Examples of things that are *not* AI include:

Determined by physical laws: Examples include the flight of a tennis ball or the reaction of a car that hits an obstacle.

Purely random: An example would include which block falls next in Tetris.

Direct response to game rules/user inputs: This includes events for which the response is predetermined by the game designer. This includes typical camera control, scripted animations, events that are triggered by the user's inputs, and events that are scheduled to occur at a particular time or after a particular time delay.

One notable gray area is where AI ends and animation begins. For example, a soccer player dribbling the ball must make decisions as to how to avoid opponents, which in turn affects the direction and speed with which he runs, which in turn affects joint angles. Typically, AI systems control the high-level decisions and the animation controls the lower level decisions:

Should I run with the ball or pass it? This is definitely an AI decision (unless it has been scripted)

If I run, what path should I take? This is getting into the gray area. If we wish to evaluate the likelihood of success of various options, based on hypotheses of how the player and other NPCs might respond, we are definitely in the realm of AI. If we simply wish to compute a shortest obstacle-avoiding path (say using Dijkstra's algorithm), this is in the realm of *algorithmics*[3]

How to move my legs to travel along this path? Now we are definitely outside of the realm of AI, and into the realm of animation.

Properties of a Good AI System: The following is a list of generally good properties of a game AI system.

Goal driven: The AI system should behave in a manner that is consistent with the (implicit) high-level goals of the entities involved.

Responsive: The AI system should respond rapidly to relevant changes in the state of the world. For example, if a path is blocked, the NPC should respond quickly by computing a new path.

Smart, but not omniscient: The AI system should behave as if it knows a good deal about the world (inanimate objects, other NPCs, and even the player) and select its behaviors accordingly. Of course, an NPC cannot act based on information that it could not reasonably have knowledge of.

Consistent: An NPC should behave in a consistent manner, to generate the impression that it embodies a believable character.

Efficient and Practical: Computational resources are limited, and the time needed to develop, program, and test the AI system must be considered within the economic constraints of the game.

[3]Some might claim that this is AI, since algorithmics is just an offshoot of AI. If you say this in earshot of an algorithms researcher, be prepared to get punched in the face.

Unfortunately, many of these goals conflict with each other, and many of the problems in game AI result from developers making compromises in quality for the sake of simplicity or efficiency.

The AI Loop: The fundamental view of a game from the perspective of AI involves continuously evaluating the current state of the (perceptible) world and determining what should the agent do in the very near future (that is, the next frame to be drawn). This generally may involve a number of layers of sensing and decision making. These are illustrated in Fig. 60.

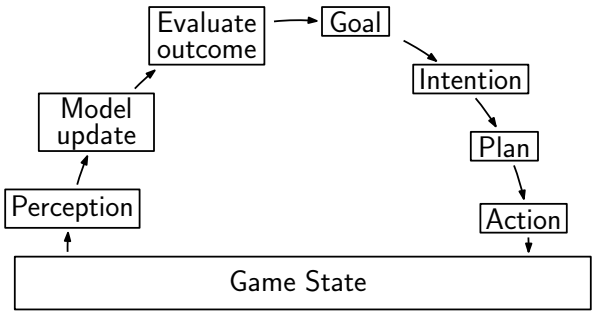


Fig. 60: The game AI loop.

Perception: The entity senses the elements of the game state that is within its scope of perception.

Model update and outcome: The perception of the state results in updates to the entities internal state model. The outcome of these state changes may be very simple ("I am injured and need to retreat") but could conceivably be quite sophisticated in the context of a game with complex narrative structure ("An ally has acted against my interests. Is he a spy?")

Goals and intentions: Based on a character's understanding of the world state, what are its motivations, and how are these motivations weighed to arrive at a set of goals? These goals need to be mapped these into intentions that are to be manifest through the character's future actions. ("That sniper is killing too many of us. How can we reduce our visibility? Hiding? Shoot out the lights? Smoke grenades?")

Plan and action: Given these intentions, the character then needs to develop a plan of actions in order to achieve the desired results. Such a plan consists of a sequence of tasks. Once a plan has been developed, the character needs to act in order to perform these tasks.

Agents: NPCs are often modeled in games through the use of an AI construct called an agent. An (autonomous intelligent) *agent* is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future.

At a high conceptual level, an agent is characterized by three basic components, which operate iteratively over time:

Sensing: Perceive features of the environment, and in particular, changes to the environment that are relevant to the agent's goals.

Thinking: Decide what action to take to achieve its goals, given the current situation and its knowledge. Of course, this is where all the complexity lies. Thinking may be simple reaction ("Danger. Flee!") or may involve complex responses based on past experiences and learning. If the objectives are complex, apply planning strategies to break them into well-defined actions.

Acting: Carry out these actions.

One can develop a taxonomy of agents, based on their sophistication.

Simple reflex agent: acts solely based of the current perception without regard to previous experiences. These are usually implemented by simple *rule-based systems*. "If X occurs then do Y." These are often encoded as a table, where input events are mapped to integer indices and the ith table entry is an encoding of the action to take on stimulus i.

Model-based agent: extends the simple reflex agent by storing its own internal state ("I'm healthy, hungry, injured, etc."), the state of the perceived world ("I hear a threat approaching from the east"), and some model of how the world works.

Such an agent chooses an action in the same way as the reflex agent, but the action will generally depend on the model's current state. These agents are often implemented by a *finite-state machine*. The machine's state corresponds to the agent's state, and current perception triggers an action and a transition to a possibly different state (For example, "If I am wandering and healthy (state) and see the player's avatar (current perception), I will start pursuing it (action).")

Goal-based agent: further extends on the capabilities of the model-based agents, by using *goal information*. Goal information describes situations that are desirable. This allows the agent a way to select among multiple possible intentions, selecting the one that will reach a goal state. Search and planning algorithms may be invoked to map goals into low-level actions.

Utility-based agent: further extends the goal-based agent by defining a measure of how desirable a particular state is. This measure is expressed through the use of a *utility function*, which measures how happy the agent would be with this state. The agent then chooses the action that maximizes the expected utility of the action.

Learning agent: takes all of this a step further by evaluating the results of past action, and uses this information to make (hopefully) better choices in the future. It has two important components, a *learning element*, which is responsible for making improvements by critically evaluating the benefit of the outcomes of prior actions, and *performance element*, which is responsible for selecting external actions. Note that future actions may not be chosen solely on the basis of expected utility, but there may also be exploration of unknown states to determine their utility.

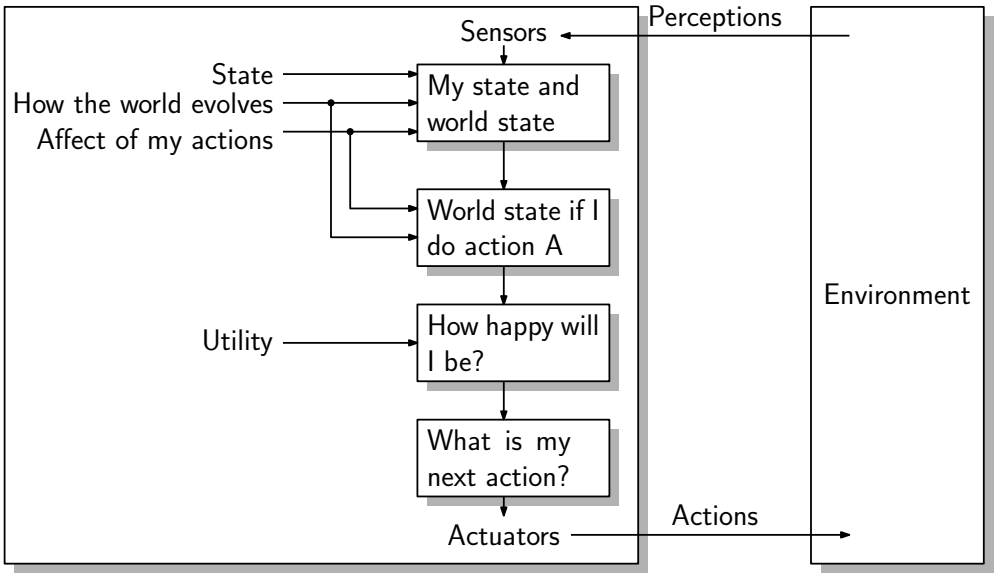


Fig. 61: A utility-based agents.

Looking ahead: In future lectures, we will explore two important aspects of AI in computer games. The first is planning motion to achieve various goals (e.g., pursue the enemy by the shortest path) subject to various constraints

(e.g., avoid obstacles). We will also discuss techniques for making decisions.

Lecture 17: Artificial Intelligence for Games: Planning Motion

Sources: Some of the material from today's lecture comes from the book "AI Game Programming Wisdom 2" by S. Rabin and "Planning Algorithms" by S. M. LaValle (Chapts. 4 and 5).

Motion For the next couple of lectures, we will discuss one of the major elements of the use of AI in games, planning motion for non-player characters (NPCs). Motion is a remarkably complex topic, which can range from trivial (computing a straight-line path between two points in the plane) to tremendously complex, 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 through a tight passageway
- planning the motion of a character from one location to another who is moving in a dense crowd of other moving people (who have their own destinations)
- planning complex ad hoc motion, like a mountain climber jumping over boulders or climbing up the side of walls

Game designers have some advantages in solving these problems, since the environment in which the NPCs move is under the control of the game designers. Nonetheless, the techniques that we will present for doing motion planning are broadly applicable, even though the may not need to applied in their full generality. (For example, we can place our obstacles farther apart to make it easier for characters to fit between them.)

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* to reduce the problem of moving a complex object (or assembly of objects) to one of moving a single point through space space

Discretization: methods such as *waypoints*, *roadmaps*, and *navigation meshes* to reduce the problem of moving a point in space to computing a path in a graph

Shortest paths: practical and efficient algorithms for computing and representing shortest paths

Multiple-object motion:

Flocking: methods such as *boids* for planning basic flocking behavior (as with birds and herding animals) and applications to simulating crowd motion

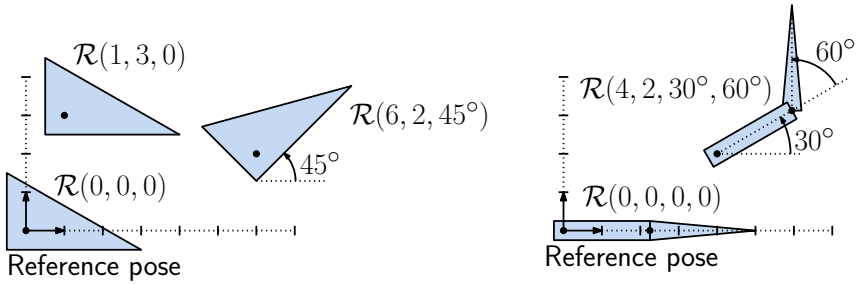
Purposeful crowd motion: techniques such as *velocity obstacles* for navigating a single agent from an initial start point to a desired goal point through an environment of moving agents

Configuration Spaces: Let us consider the problem of planning the motion of a single agent among a collection of obstacles. Since the technique that I will be presenting arises from the field of robotics, we'll refer to the agent henceforth 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 assume that the workspace is static, that is, the obstacles do not move. We also assume that a complete geometric description of the workspace is available to us.

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 values that fully specifies the position of the robot. Combined, these two elements fully define the robot's exact shape and position of the robot in space.

For example, suppose that the robot is a 2-dimensional polygon that can translate and rotate in the plane. 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. 62(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.



(a) (b)

Fig. 62: Configurations of: (a) translating and rotating robot and (b) a translating and rotating robot with a revolute joints.

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. 62(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 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

Such a configuration is called a *forbidden configuration*. 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. (Note that the regions do not need to be connected. That is, there may be two free configurations, but there is no path between them that lies entirely within the free space.)

C-Obstacles and Paths in Configuration Space: *Motion planning* is the following problem: Given a workspace S, a robot $\mathcal{R}$, an initial configuration s, and a final configuration t (both points in the robot's free configuration space), 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. 63(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. Thus, we have reduced the task of

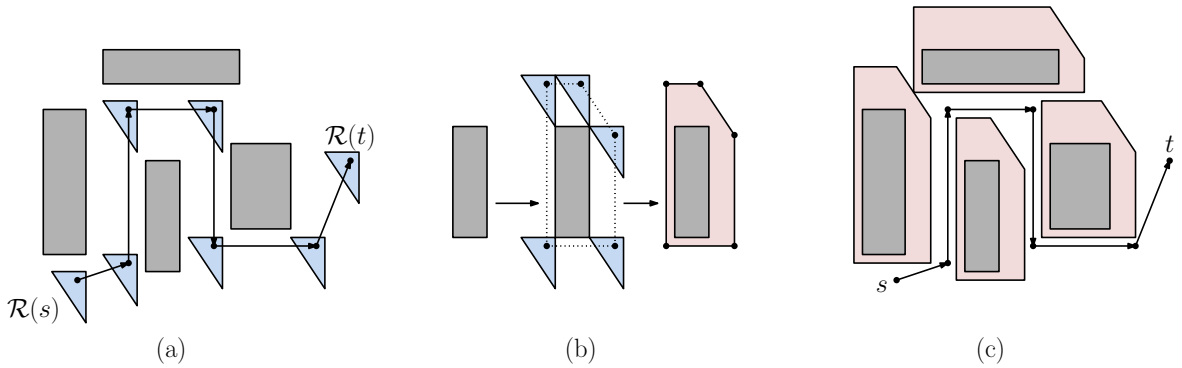


Fig. 63: Motion planning: (a) workspace with obstacles, (b) converting obstacles into C-obstacles, and (c) configuration space and C-obstacles.

planning the motion of a robot in its workspace to the problem of finding a path for a single point through free configuration space.

Since these are rather hard to illustrate, let's consider instead perhaps the simplest nontrivial case, which is 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.

In order to see how to define the C-obstacles in this simple example, consider the path traced out by the robot's reference point as we "scrape" it along the boundary of the obstacle (see Fig. 63(b)). The associated C-obstacle is simply the collection of points traced out by this path (or generally a surface in higher dimensions) (see Fig. 63(b)). Given the collection of C-obstacles in configuration space, the motion planning problem reduces to determining whether there exists a *path* from s to t that avoids all the C-obstacles. It is easy to prove that there exists a valid motion in the workspace if and only if there exists a path in the free space.

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 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.

Discretizing Configuration Space: As mentioned above, we can reduce the motion planning problem (for a single moving agent) to the problem of computing a path in the free configuration space. 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.)

There are two common techniques for computing such paths. The first is based on a sort of physical simulation of *repulsive potential fields* and the second is based on *graph search algorithms*. Let's consider each of these.

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. 64(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. 64(b)). Think of the obstacles like the plateaus you find out in the American deserts, where the cross section of the plateau is the obstacle.

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. 64(c)).

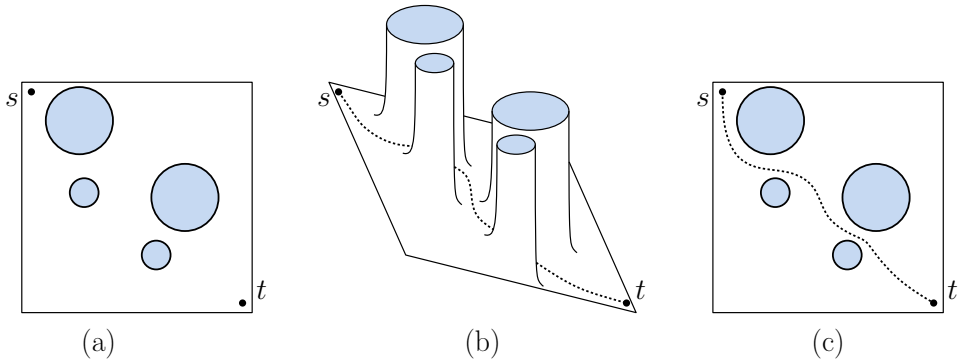


Fig. 64: Computing paths by potential-field navigation.

More formally, the process is modeled as follows. First, we set up a repulsive field around obstacles (and generally any forbidden regions of the configuration space). Next, we set up an attractive field centered at the desired destination point t. Finally, with the aid of a physics simulator, we let our robotic marble flow "downhill" along the line of steepest descent in the resulting potential field.

How do we do this? If we let $\Psi(x, y)$ denote the value of the potential field at any point direction (x, y), then the direction of steepest descent it given by the *gradient vector*, which can be computed from the partial derivatives of Ψ. More formally, the gradient is $\nabla\Psi = (\partial\Psi/\partial x, \partial\Psi/\partial y)$. Thus, if you define your potential fields to be differentiable functions, then the gradient is easy to compute at any point. Of course, the gradient changes from one point to the next. The simulator follows the gradient direction downhill for a short distance. It then recomputes the gradient at the new point, and keep continuing.

One of the features of potential-field navigation is that, if the physics is properly modeled, the movement point naturally follows a smooth energy-minimizing path. Thus, it results in smooth, natural motion.

While potential-field navigation has a certain intuitive charm to it, it has a significant downside, which is shared by all gradient-descent methods. Namely, it can get stuck in local minima of the potential field. Once your robotic marble roles into a depression surrounding by mountainous obstacles on all sides, it will come to rest there, without reaching the destination. This suggests the need for more *global* solutions, which will discuss next.

Graph-Based Navigation: The other widely used class of methods are based on first discretizing the configuration free space into a graph model (with nodes and edges), and then applying any standard shortest path algorithm, such as Dijkstra's algorithm or A* search to compute the shortest path. (We'll say more about these algorithms

later.) There are many different methods that are used for extracting this graph. We will consider a few common ones for the remainder of this lecture.

Waypoints and Road Maps: The simplest approach for generating a navigation graph, is to scatter a large number of points throughout the configuration free space, called *waypoints*, and then connect nearby waypoints to one another. The edges of this graph can be labeled with the distance between the associated points. The resulting graph is called a *road map*. (Note that these are distances in configuration space, so this is not simply Euclidean distance. For example, you may need to define what is meant by the distance of rotating a joint through an angle of θ.) Given the location of the start point s and the destination point t, we join these with edges to nearby waypoints. Finally, we invoke a shortest path algorithm. The result is a path in configuration space.

How do we construct these waypoints? There is no single preferred method. Here are a few ideas:

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 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, and generate waypoints along the vertices of the grid (see Fig. 65(b)). This has the advantage of simplicity, but note that it can result in the generation of a very large number of grid points, if the grid resolution is not properly set.

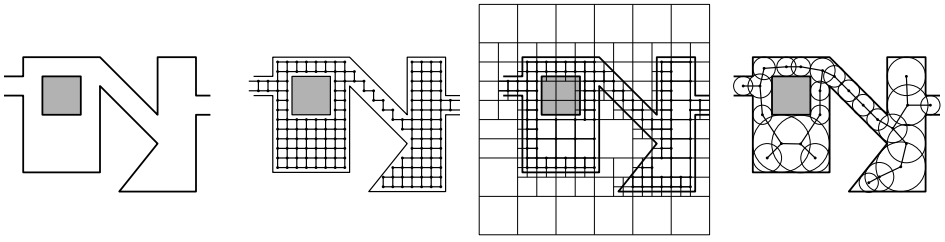


Fig. 65: A road map for a set of obstacles (a) based on waypoints generated on: (b) a grid, (c) a quadtree, (d) the medial axis.

Quadtree: Placing the waypoints at the vertices of a quadtree decomposition has the advantage that the resolution of the waypoints can be adjusted to fit the tightness of the environment (see Fig. 65(c)). Narrow cramped corridors will be decomposed into small regions, and wide open fields into large regions.

Medial-Axis Waypoints: Both grids and quadtrees suffer from the same problems. The first is that motion along the vertices of either structure tends to produce paths that are aligned with the coordinate axes. Thus, motion along a diagonal corridor will tend to zig-zag to the left and right. Additional steps are needed to smooth out such paths.

One idea for fixing this is to situate the waypoints so they correspond to the centers of obstacle-free disks of maximum size. We say that a circular disk D 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 your domain. It is an important object in geometry, called the *medial axis*.

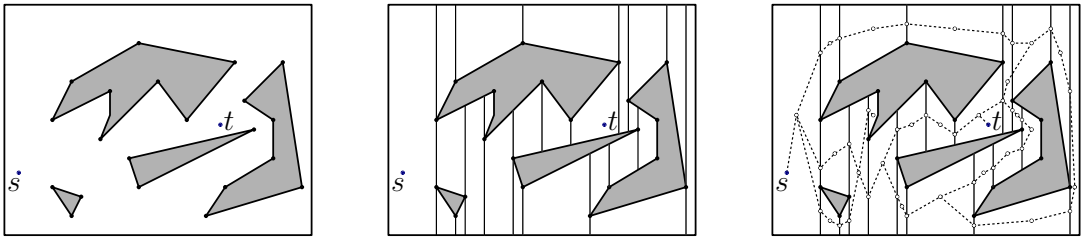
By sampling waypoints on or near the medial axis (see Fig. 65(d)), agents will naturally move along the centers of corridors. (Of course, you can add a bit of random variation, so they merely walk in a

more natural manner that is not too close to the walls.) 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) and where the domain consists mostly of corridor-like structures.

Navigation Meshes: One of the shortcomings of all the aforementioned approaches is that they neglect the essential feature that the free configuration space is a *region*, not a collection of discrete points. If you are a dynamic setting, and an obstacle is placed on a waypoint in the center of a narrow corridor, it may be impossible to determine how to navigate around the obstacle, since there are no alternate waypoints nearby.

The next idea is based on producing a decomposition of the free configuration space into a collection of convex regions. Such a decomposition is called a *navigation mesh*. There are many ways to build road maps based on such meshes. A very popular way is to shoot a bullet path both up and down from each obstacle vertex. These bullet paths subdivide space into a collection of trapezoid shaped regions (with parallel vertical sides). The decomposition is called a *trapezoidal map*, and it can be computed very efficiently (see Fig. 66(b)).

To generate a road map from a trapezoidal map we create a vertex in the center of each trapezoid and a vertex at the midpoint of each vertical edge. We create edges joining each center vertex to the vertices on its (at most four) edges (see Fig. 66(c)).



(a) (b) (c)

Fig. 66: Trapezoidal-map based navigation mesh: (a) the obstacles, (b) trapezoidal map, (c) road map with connecting links to s and t.

Now to answer the motion planning problem, we assume we are given the start point s and destination point t. We locate the trapezoids containing these two points, and connect them to the corresponding center vertices. We can join them by a straight line segment, because the cells of the subdivision are convex.

Lecture 18: Artificial Intelligence for Games: Finding Paths

Sources: Some of the material from today's lecture comes from the book "Artificial Intelligence for Games" by I. Millington and J. Funge. The material on A* search is derived from D. Nau's lecture notes.

Recap: In the previous lecture, we discussed a technique for reducing the motion of an general object amidst a set of obstacles to the motion of a single point (in configuration space) among a collection configuration-obstacles, or C-obstacles. Since computing paths in a general geometric domain is difficult, we also presented technique for mapping the problem into one involving graphs. This involved sampling points from the free space (called *waypoints*) and judiciously connecting nearby waypoints to form a graph. Following this, techniques for computing shortest paths in graphs can be applied or adapted. Today, we will discuss some of these algorithms and how to adapt them to the context of computing shortest paths in graphs.

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. 67(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 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.

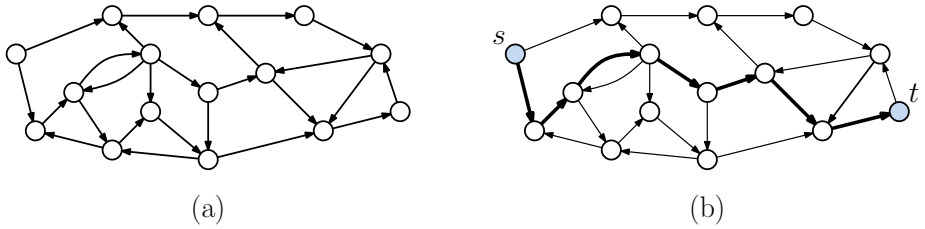


Fig. 67: A directed graph.

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. 67(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 best-known.

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.[4] Dijkstra's algorithm operates in a greedy manner. It associates each node u with a *distance estimate* $d[u]$, which in general is an upper bound on the distance from s to u. We maintain the nodes in two classes, those that are *unvisited* (that is, waiting to be processed, or sometimes called *open*) and those that have been *visited* (that is, already processed, or sometimes called *closed*). At each step, Dijkstra's algorithm selects the unvisited node u that has the lowest distance estimate. For each neighbor v of u, we update its distance estimate by setting $d[v] = \min(d[v], d[u] + w(u, v))$. This process is called *relaxation*.

For the sake of efficiency, the unvisited nodes are stored in a *priority queue*, which is sorted by the d-values. If the priority queue is implemented efficiently, say, using a heap data structure, then Dijkstra's algorithm runs in $O(m \log n)$ time.[5]

[4]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.

[5]If the priority queue is implemented using a much fancier data structure, called a Fibonacci heap, the running time is even better, $O(m + n \log n)$.

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. It runs in time $O(nm)$.

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 $O(n^3)$.

Note that the output size of Floyd-Warshall is $O(n^2)$. This may be unacceptably large. When used in games, all-pairs shortest paths are usually computed in a hierarchical context. For example, the environment can be broken up into regions or *neighborhoods*. Each neighborhood is associated with a small number of *representative waypoints* in your graph. Then shortest paths are computed between all pairs of these representatives. To compute a shortest path between an arbitrary pair of points, you can find the closest representatives to the source and destination, and simply look up the shortest path between the representatives. This described a two-tiered hierarchy, but of course you could extend this to any number of layers.

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. 68(a)). By following the predecessor pointer back to s, we can construct the shortest path, but in reverse (see Fig. 68(b)). Since this involves only a constant amount of information per node, this representation is quite efficient.

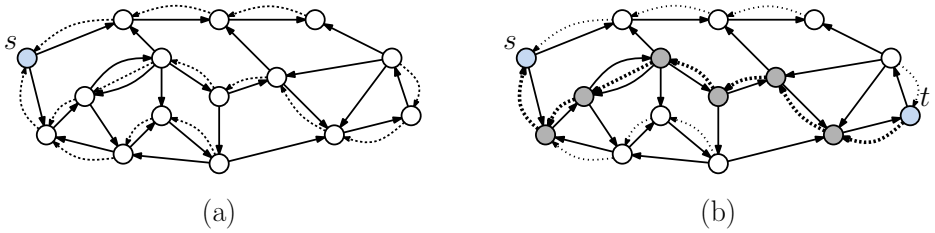


Fig. 68: Storing/reconstructing the shortest path.

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 special 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. 69(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. 69(b)). Note that this only requires *one* invocation of a shortest path algorithm, not k.

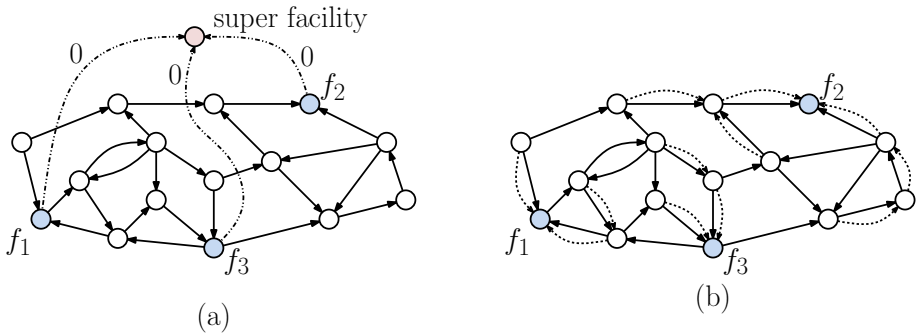


Fig. 69: Using shortest paths to compute the closest facility.

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. 70(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 *uniformed* 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?

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. 70(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). 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

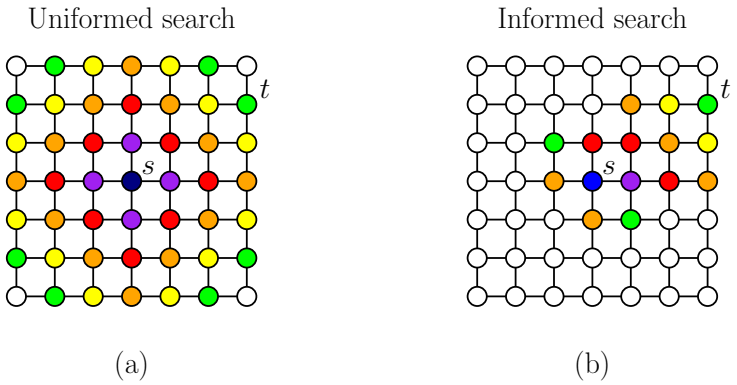


Fig. 70: Search algorithms where colors indicate the order in which nodes are visited by the algorithms: (a) uniformed search (such as Dijkstra) and (b) informed search (such as A*).

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) {
  for (each node u) {                    // initialize
    d[u] = +infinity;  mark u as undiscovered
  }
  d[s] = 0;  mark s as discovered        // distance to source is 0
  repeat forever {                       // 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 as discovered
      }
      mark u as 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

Best-First Search

```
BestFirst(G, s, t) {
  for (each node u) {                    // initialize
    d[u] = +infinity;  mark u as undiscovered
  }
  d[s] = 0;  mark s as 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 as discovered
      }
      mark u as finished                 // we're done with u
  }
}
```

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 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) {
  for (each node u) {                    // initialize
    d[u] = +infinity;  mark u as undiscovered
  }
  d[s] = 0;  mark s as 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 as discovered
      }
      mark u as 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. 71. 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, $h(f) = 9$. (For this graph it is easy to verify that $h(\cdot)$ is an admissible heuristic.)

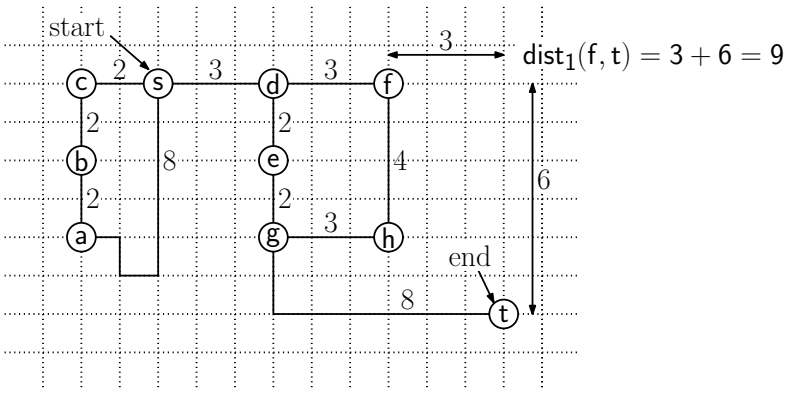


Fig. 71: The graph G used in the sample runs.

Dijkstra's Algorithm: The table below provides a trace of Dijkstra's algorithm. Each table entry indicates the d-values associated with each node. At each stage (each row of the table), the node with the lowest d-value

is selected next for processing. To indicate that a node is finished, we give its d-value as "–" since it will not change again.

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	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	–	8	∞	2	3	∞	∞	∞	∞	∞
2: c	–	8	4	–	3	∞	∞	∞	∞	∞
3: d	–	8	4	–	–	5	6	∞	∞	∞
4: b	–	6	–	–	–	5	6	∞	∞	∞
5: e	–	6	–	–	–	–	6	7	∞	∞
6: a	–	–	–	–	–	–	6	7	∞	∞
7: f	–	–	–	–	–	–	–	7	10	∞
8: g	–	–	–	–	–	–	–	–	10	15
9: h	–	–	–	–	–	–	–	–	–	15
10: t	–	–	–	–	–	–	–	–	–	–
Final	0	6	4	2	3	5	6	7	10	15

Best-First Search: The table below shows the trace of best-first search. Each entry contains the value $d[u]/h(u)$. (The $h(u)$-values are only shown for the discovered nodes.) At each stage the discovered node with the smallest h-value is chosen for processing. As above, to indicate that a node is finished, we give its d-value as "–" since it will not change again.

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	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	–	8/13	∞	2/17	3/12	∞	∞	∞	∞	∞
2: d	–	8/13	–	2/17	–	5/10	6/9	∞	∞	∞
3: f	–	8/13	4/15	2/17	–	5/10	–	∞	10/5	∞
4: h	–	8/13	4/15	2/17	–	5/10	–	13/8	–	∞
5: g	–	8/13	4/15	2/17	–	5/10	–	–	–	21/0
6: t	–	8/13	4/15	2/17	–	5/10	–	–	–	–
Final	0	8	4	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 A* search. Each entry contains the value $d[u]/f(u)$. (The $f(u)$-values are only shown for the discovered nodes.) At each stage the discovered node with the smallest f-value is chosen for processing. As above, to indicate that a node is finished, we give its d-value as "–" since it will not change again.

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	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	–	8/21	∞	2/19	3/13	∞	∞	∞	∞	∞
2: d	–	8/21	∞	2/19	–	5/15	6/15	∞	∞	∞
3: e	–	8/21	∞	2/19	–	–	–	7/15	–	∞
4: g	–	8/21	∞	2/19	–	–	–	–	–	15/15
5: t	–	8/21	∞	2/19	–	–	–	–	–	–
Final	0	8	∞	2	3	5	6	13	10	21

Note that the algorithm computes the correct result, but it terminates after 5 stages, not 10 as was the case for Dijkstra's algorithm.

A* with inadmissible heuristic: The table below shows the trace of A* search using the heuristic $h'(u) = 10 \cdot \text{dist}_1(u, t)$, which is not admissible for this input. Each entry contains the value $d[u]/f(u)$. (The $f(u)$-values are only shown for the discovered nodes. At each stage the discovered node with the smallest f-value is chosen for processing. As above, to indicate that a node is finished, we give its d-value as "–" since it will not change again.

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	∞	∞	∞	∞	∞	∞	∞	∞	∞
1: s	–	8/138	∞	2/172	3/123	∞	∞	∞	∞	∞
2: d	–	8/138	–	2/172	–	5/105	6/96	∞	∞	∞
3: f	–	8/138	4/154	2/172	–	5/105	–	∞	10/60	∞
4: h	–	8/138	4/154	2/172	–	5/105	–	13/83	–	∞
5: g	–	8/138	4/154	2/172	–	5/105	–	–	–	21/21
6: t	–	8/138	4/154	2/172	–	5/105	–	–	–	–
Final	0	8	4	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 19: Artificial Intelligence for Games: Multiple Agent Motion

Sources: Today's material is drawn from a variety of sources. Some of the material on particle systems is drawn from lecture notes by Alex Benton from the University of Cambridge and the classic paper "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects," by W. T. Reeves, *ACM Transactions on Graphics*, 2, 1983, 91–108. The material on flocking is based in part on C. W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics*, 21, 25–34, 1987. The material on reciprocal velocity obstacles is taken from J. van den Berg, S. Guy, M. C. Lin, D. Manocha, "Reciprocal n-body Collision Avoidance," *Proc. Int. Symposium of Robotics Research* (ISRR), 2009.

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.

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. 72).

Fig. 72: A particle system simulating the flow of a liquid.

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.

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.

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 v_0, and some fixed field force g (representing, for example, the affect of gravity) the position of the particle at time t can be expressed in closed form as:

$$p(t) \;=\; p_0 + v_0 t + \frac{1}{2} g t^2.$$

(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 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 v, and its *current acceleration*, which is given by a vector a. (Acceleration should be thought of as the accumulated effect of all the forces acting on the particle. For example, gravity tends to decrease the vertical component of the velocity and air resistance tends to decrease the particle's absolute velocity, without altering its direction.) We can then update the state over a small time interval, Δt (for example, the elapsed time between two consecutive frames). By the basic laws of kinematics (which we may discuss later this semester):

$$v' \;=\; v + a \cdot \Delta t \qquad \text{and} \qquad p' \;=\; p + v \cdot \Delta t.$$

where p' and v' denote the particle's new position and velocity, respectively. (Depending on your implementation, you may also need to update the forces acting on the particle and update acceleration as well.)

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. 73).

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. 73: An example of complex emergent behavior in flocking.

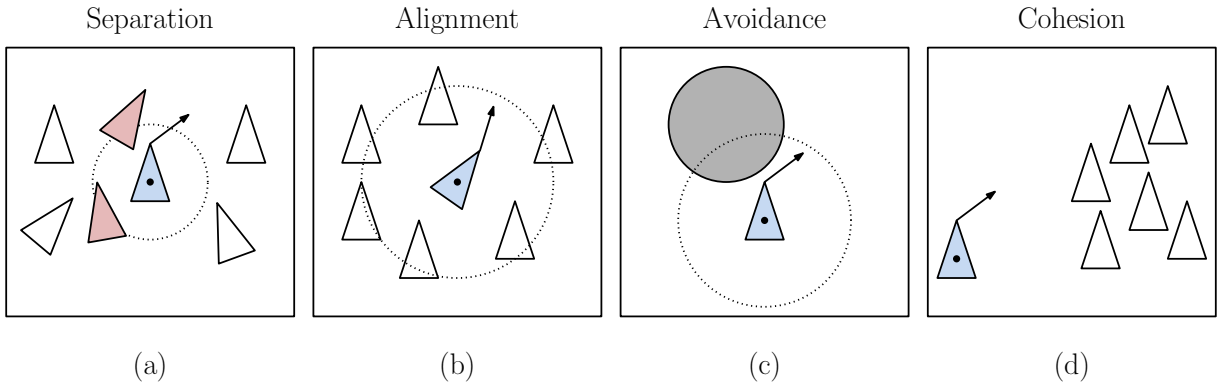


Fig. 74: Forces that control flocking behavior.

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, 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 set the boid's acceleration. Given this acceleration vector a caused by the boid forces, we apply the same update rules, $v' = v + a \cdot \Delta t$ and $p' = p + v \cdot \Delta t$, to obtain the new position p' and new velocity vector 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 way 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 determine what the laws of kinetics would do. 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 u pointing in the induced direction and the strength s given as a scalar, we can compute the updated acceleration vector as $a = c \cdot s \cdot u$, 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 numeric 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.

Crowd Simulation: The last, and most interesting, type of motion simulation is that of crowds of agents. Unlike flocking systems, in which it is assumed that the agents behave homogeneously, in a crowd it is assumed that every agent has its own agenda to pursue (see Fig. 75). 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.)

Fig. 75: Crowd simulation.

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.

In order to plan such motions, we will next introduce the handy concept of a velocity obstacles and reciprocal velocity obstacles.

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. 76(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

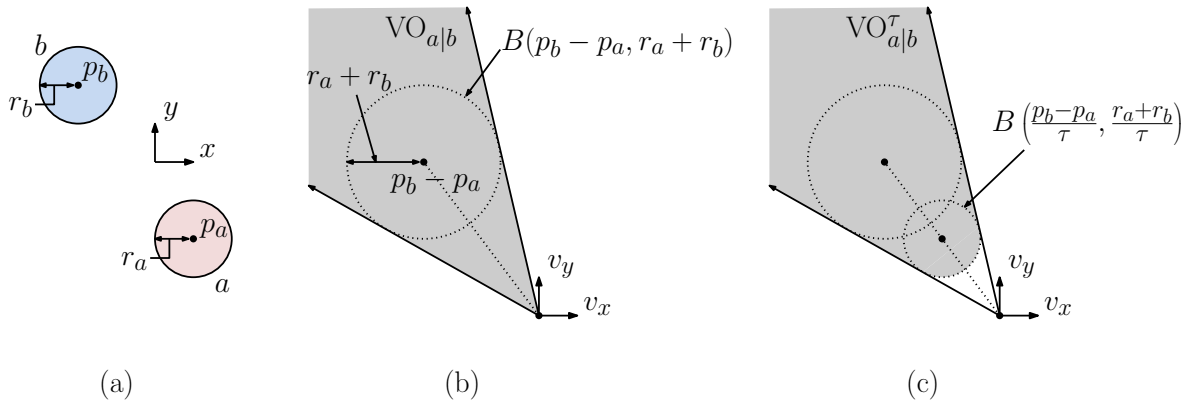


Fig. 76: Velocity obstacle for a induced by b (assuming b is stationary).

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

$$\|(p_a + t \cdot v_a) - p_b\| \ < \ r_a + r_b. \tag{1}$$

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

$$B(p, r) \ = \ \{q : \|q - p\| < r\}.$$

We can rewrite Eq. (1) 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. (1) 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. 76(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. 76(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

$$\text{VO}^{\tau}_{a|b} \;=\; -\text{VO}^{\tau}_{b|a}.$$

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, $v_a - v_b \in \text{VO}^{\tau}_{a|b}$. Equivalently, v_a will generate a collision if b if v_a lies in the offset velocity obstacle $\text{VO}^{\tau}_{a|b} + v_b$ (see Fig. 77(a)).

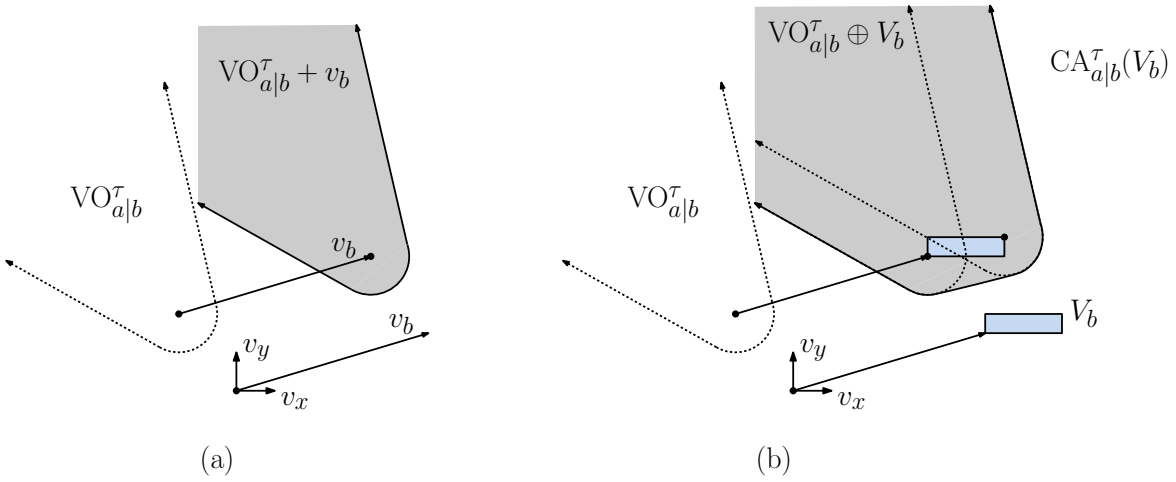


Fig. 77: Velocity obstacles where (a) object b is moving at velocity v_b and (b) object b is moving at any velocity in the set V_b.

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 $\text{VO}^{\tau}_{a|b} \oplus V_b$. 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

$$\text{CA}^{\tau}_{a|b}(V_b) \;=\; \{v : v \notin \text{VO}^{\tau}_{a|b} \oplus V_b\}$$

(see Fig. 77(b).)

Just to recap, if a selects its velocity vector from anywhere outside $\text{VO}^{\tau}_{a|b} \oplus V_b$ (that is, anywhere inside $\text{CA}^{\tau}_{a|b}(V_b)$), 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

$$\text{CA}_a^\tau = \bigcap_b \text{CA}_{a|b}^\tau(V_b)$$

  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. 78. Two agents are moving to each other. They see that they are on a collision course, and so the divert from their ideal 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. 78(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. 78(c)). So, both agents return to their original velocity, and they are now right back on a collision course (see Fig. 78(d)) and so again they divert (see Fig. 78(e)). Clearly, this vicious cycle of zig-zagging motion will repeat until they manage to make it around one another.

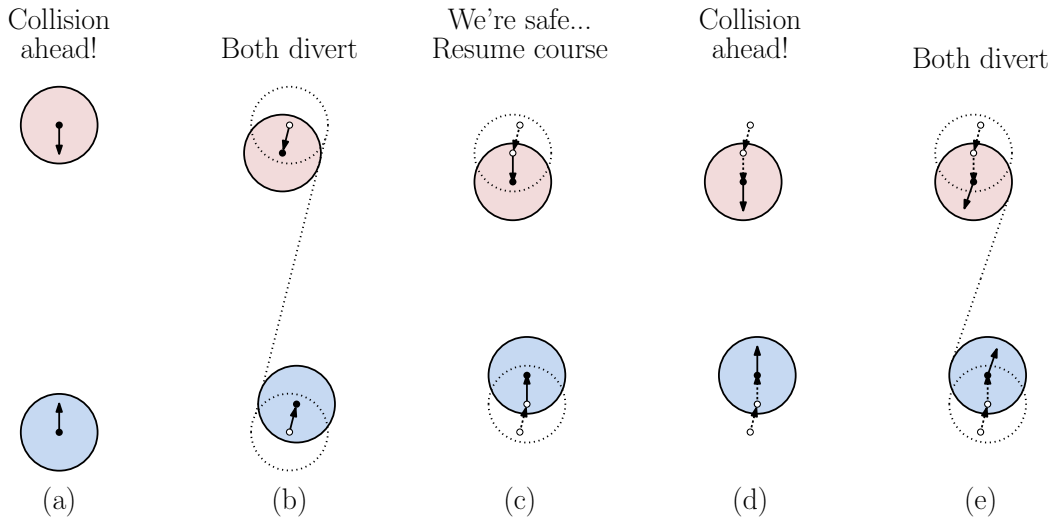


Fig. 78: Oscillation that can result from standard velocity-obstacle motion planning.

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 $\text{CA}^{\tau}_{a|b}(V_b)$, the collision-avoiding velocities for a assuming that b selects its velocity from V_b, and $\text{CA}^{\tau}_{b|a}(V_a)$, 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

$$V_a \subseteq \text{CA}^{\tau}_{a|b}(V_b) \qquad \text{and} \qquad V_b \subseteq \text{CA}^{\tau}_{b|a}(V_a).$$

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

$$V_a = \text{CA}^{\tau}_{a|b}(V_b) \qquad \text{and} \qquad V_b = \text{CA}^{\tau}_{b|a}(V_a).$$

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 $\text{area}(V_a \cap B(v_a^*, r))$ 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 $\text{area}(V_a \cap B(v_a^*, r))$ and $\text{area}(V_b \cap B(v_b^*, r))$ 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 $\text{area}(V_a \cap B(v_a^*, r)) = \text{area}(V_b \cap B(v_b^*, r))$. 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: $\text{area}(V_a \cap B(v_a^*, r)) = \text{area}(V_b \cap B(v_b^*, r))$

Maximal: For any other reciprocal collision avoiding set (V_a', V_b'),

$$\min(\text{area}(V_a \cap B(v_a^*, r)), \text{area}(V_b \cap B(v_b^*, r))) \geq \min(\text{area}(V_a' \cap B(v_a^*, r)), \text{area}(V_b' \cap B(v_b^*, r))).$$

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 $(\text{ORCA}^\tau_{a|b}, \text{ORCA}^\tau_{b|a})$, 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, $v_a^* - v_b^* \in \text{VO}^\tau_{a|b}$. 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 $\text{VO}^\tau_{a|b}$ that lies closest to $v_a^* - v_b^*$ (see Fig. 79(a)). Since $\text{VO}^\tau_{a|b}$ 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.)

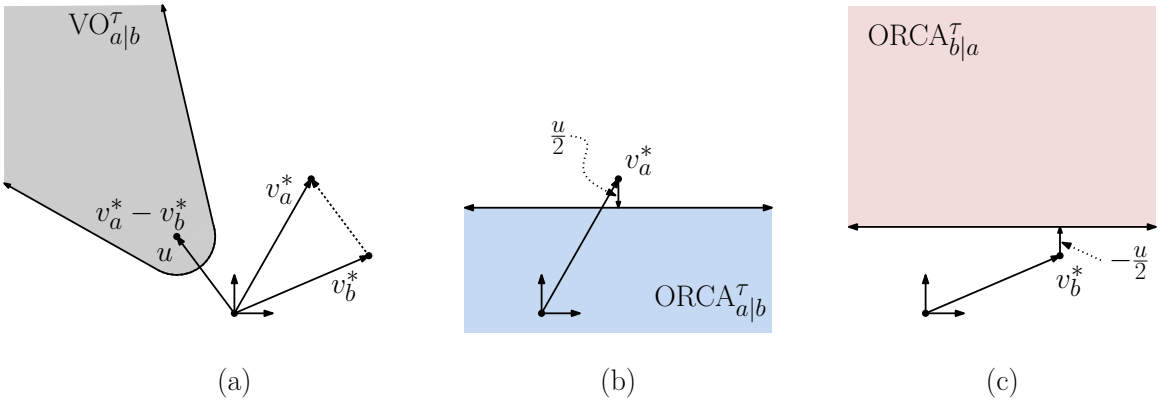


Fig. 79: Computing an optimal reciprocal collision avoiding pair of candidate velocities.

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:

$$\text{ORCA}^\tau_{a|b} \;=\; \left\{ v : \left(v - \left(v_a^* + \frac{u}{2} \right) \right) \cdot n \geq 0 \right\},$$

(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. 79(b)). Define $\text{ORCA}^\tau_{b|a}$, symmetrically, but using $-\frac{u}{2}$ instead (see Fig. 79(c)). In their paper, van den Berg, Lin, and Manocha claim that the resulting pair of sets $(\text{ORCA}^\tau_{a|b}, \text{ORCA}^\tau_{b|a})$ define an optimal reciprocally maximal pair of collision avoiding. In other words, if a selects *any* velocity from $\text{ORCA}^\tau_{a|b}$ and b selects any velocity from $\text{ORCA}^\tau_{b|a}$, 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, $\bigcap_{i=1}^n \text{ORCA}^\tau_{a|b_i}$. 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.

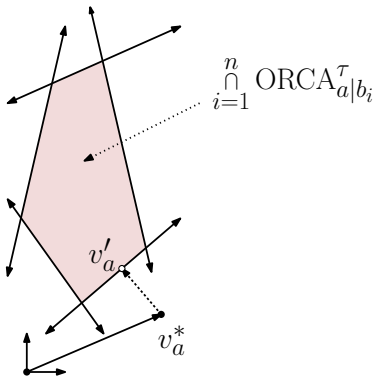


Fig. 80: Computing agent a's velocity.

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.

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 20: Artificial Intelligence for Games: Decision Making

Sources: The material on Behavior Trees has been taken from a nice lecture by Alex Champandard, "Behavior Trees for Next-Gen Game AI," which appears on aigamedev.com (visit: http://aigamedev.com/insider/presentations/behavior-trees/).

Decision Making: So far, we have discussed how to design AI systems for simple path planning. Path planning is a relatively well defined problem. Most application of AI in game programming is significantly more involved. In particular, we wish to consider ways to model interesting non-player characters (NPCs).

Designing general purpose AI systems that are capable of modeling interesting behaviors is a difficult task. On the one hand, we would 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 AI 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 bahaviors in intersting 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.

- Rule-based systems
- Finite state machines
- Behavior trees

Rule-based Systms: The very first computer games used rule-based systems to control the behaviors of the NPCS. A *rule-based system* is one that stores no (or very little) state information, and the behavior of an NPC is a simple function of the present conditions it finds itself in.

As an example, consider planning the motion of one of the ghosts in the game *Pac-Man*.[6] Let's ignore for now the fact that ghosts have two states, depending on whether they are chaising the Pac-Man or they are being chased. Even when they are chasing the Pac-Man, ghosts alternate between two states, depending on whether they are wandering or chasing. Let's consider a simple example of how to use a rule-based system to define wandering behavior (see Fig. 81). This simple system prefers to go ahead whenever possible, and if the way is blocked it selects (in order of decreasing preference) turning right, turning left, and reversing.

Ahead	**Right**	**Left**	**Action**
Open	–	–	Go ahead
Blocked	Open	–	Turn right
Blocked	Blocked	Open	Turn left
Blocked	Blocked	Blocked	Turn around

Fig. 81: A (ridiculously) simple wandering behavior for a ghost in Pac-Man.

Of course, this is too simplistic to be useable in a game, but it illustrates some of the features of a ruled-based system. It is easy to implement (just a look-up table), easy to modify, and easy even for non-programmers to understand. There a number of ways that one could enhance this simple idea. For example, rather than just having a single action for a given set of events, there may be a number of possibilities and randomization is used to select the next option (perhaps with weighted probabilities to favor some actions over others).

Finite State Machines: The next step up in complexity from a rule-based system is to add a notion of *state* to add complexity to a character's behavior. 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 enemy combatant "bot" NPC in a first-person shooter that seeks the player's character and engages it in combat. Suppose that as the designer you decide to implement the following type of behavior:

- If you dont see an enemy, wander randomly
- When you see enemy, attack him
- When hear an enemy, chase him
- On dying, re-spawn

An example of a possible implementation of this using an FSM is shown in Fig. 83(a). For example, suppose that we are currently in the *Wander* state. If we see our enemy (that is, the player) we transition to the *Attack* state. If we hear a sound, we transition to the *Chase* state. (This would presumably be followed by querying the game database to determine where the sound came from and the AI system to compute a path.) If we die, we jump to the *Spawn* state, where we presumably wait until we have been revived and exit this state. Note that the "!" indicates that the specified event has not occurred (or that the specified condition is not satisfied).

FSMs are a popular method of defining behavior in games. The principal reasons that they are liked is that they are easy to implement, easy to design (if they are not too big), and they are easy to interpret. For example, based

[6] Our description is not accurate. There are resources on the Web that provide detailed descriptions of the Pac-Man ghost behaviors. For example, see http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior and http://donhodges.com/pacman_pinky_explanation.htm.

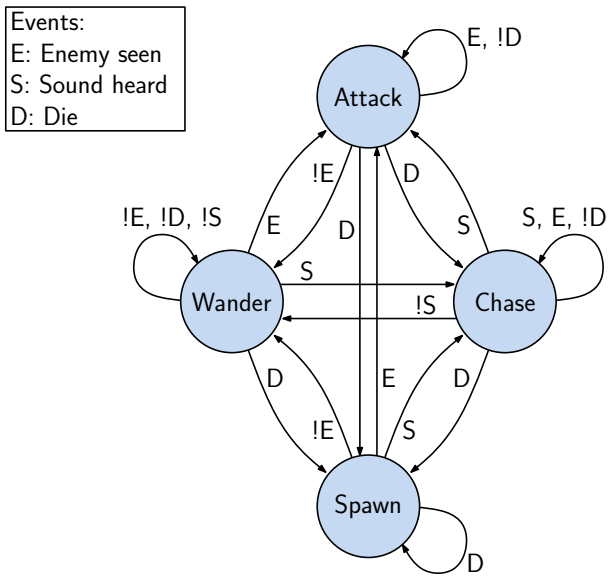


Fig. 82: Implementing an enemy combatant NPC for a FPS game.

on the graphical layout of our FSM we can observe one interesting anomaly, there is no transition from *Attack* to *Chase*? Was this intentional? To remedy this, we could create an additional state, called *Attack-with-sound-heard*. While we are attacking, if we hear a sound, we could transition to this state. When we are done with the attack, if we are in this new state, we transition immediately to the *Chase* state.

How are FSMs implemented? Basically, they can be implemented using a two-dimensional array, where the row index is the current state and the column index (or indices in general) correspond to the events that may trigger a transition.

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).

The principal problem 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 9 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.)

Hierarchical FSMs: 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). 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.

Returning to our fighting bot example, each of the major states, *Wander*, *Attack*, *Chase*, and *Spawn*, could be further subdivided into lower level states. For example, we could design different types of chasing behavior

(chase on foot, chase while riding a unicyle, chase while flying a hovercraft).

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.

Other Approaches to Hierarchical Decision Making: A good general, scalable approach is to design the AI decision system *hierarchically*. There are three classical methods for achieving a hierarchical AI structure:

Programming systems: Programming and scripting systems are very powerful (Turing complete). They are very good at describing sequential, conditional, and repetitive behaviors, since these constructs are common to all programming languages. The downside is that they are so general that it is hard to reuse components in other games or to compose components to form new behaviors.

Hierarchical Finite-State Machines (HFSM): As we have seen, these are easy to design for moderately sized systems. It is easy to generate a bunch of states and then consider the conditions under which one state leads to another. They have a nice modular structure, which makes it possible to copy FSMs from one game to another. However, FSMs and HFSMs can be clunky to deal with when designing "procedural" behaviors (do x, then y, then repeat z 20 times), which are more easily handled in a programming/scripting system.

Hierarchical Planners: AI planning systems are software systems that are used for searching among a complex state of possible actions to determine the best course of action. An example (which we will not discuss) is called a *hierarchical task network* (HTN). Planning-based systems are very powerful, and can be very useful when dealing with complex decision making. Because the planning process can take a lot of computational resources, HTN software systems tend to be rather slow. Also they do not adapt well to highly dynamic contexts, since any change in the system may require that the (time-consuming) planner be re-run from scratch. Planners are often overkill for many of the simple decisions that need to be made in typical games.

Behavior Trees: The question that this raises is whether there is a system that combines the strengths of these various systems. We would like a system that is more general the FSMs, more structured than programs, and lighter weight than 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.

Following Alex Champandard's example, 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. 83(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!).

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.

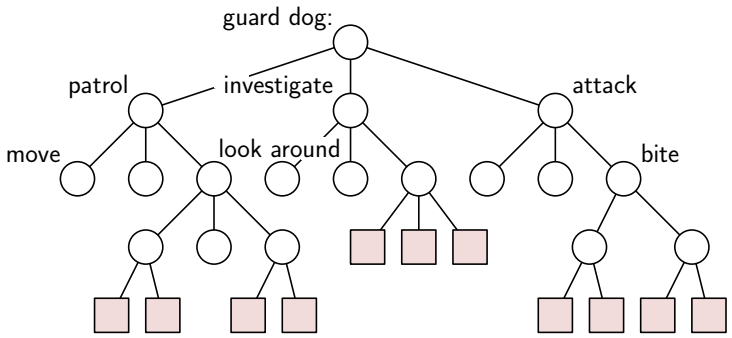


Fig. 83: Sample hierarchical structure of a guard-dog's behavior.

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. 84(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. 84(b)). If all succeed, the sequence returns success.

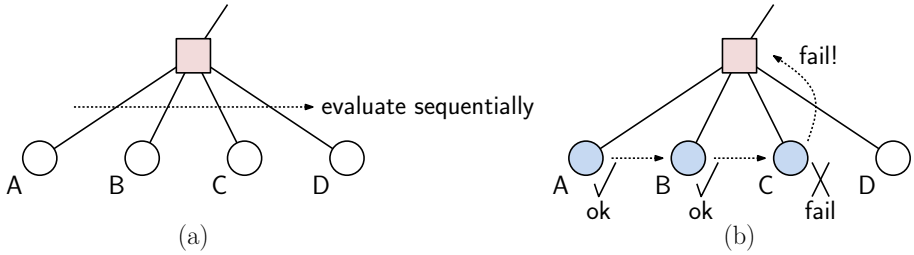


Fig. 84: Sequence: (a) structure and (b) semantics.

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. 85(b)). If none succeed, then the selector returns failure.

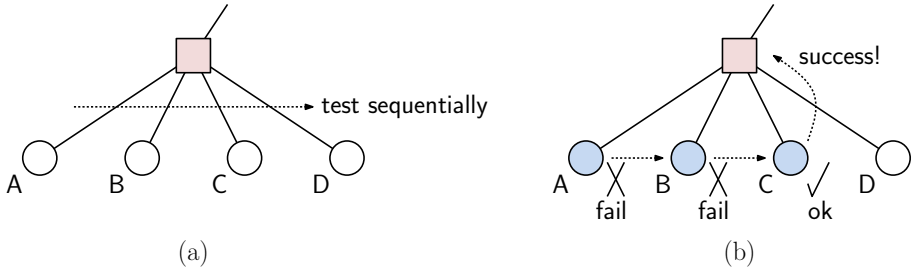


Fig. 85: Selector: (a) structure and (b) semantics.

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 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 21: Multiplayer Games and Networking

Sources: Today's lecture is from a number of sources, including lecture notes from University of Michigan by Sugih Jamin and John Laird, and the article "Network and Multiplayer," by Chuck Walters which appears as Chapter 5.6 in *Introduction to Game Development* by S. Rabin.

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. Examples include *massively multiplayer online games* (MMOGs), such as "World of Warcraft," 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., mail (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 22: Cheating in Multiplayer Games

Sources: This lecture is based on the following articles: M. Pritchard, "How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It," Gamasutra 2000; J. Yan and B. Randell, "A Systematic Classification of Cheating in Online Games," NetGames 2005, 1–9; S. D. Webb and S. Soh, "Cheating in Networked Computer Games: A Review," DIMEA 2007, 105–112

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-tamp 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.