cisc3660
game programming
fall 2012
lecture # IV.1

topics:

- game engines
- multiplayer games

references:

- `http://www.gamecareerguide.com/features/529/what_is_a_game.php`, "What is a Game Engine?", by Jeff Ward, GameCareerGuide.com, April 29, 2008.
- `http://www.evl.uic.edu/spiff/class/cs426/`, by Prof Jason Leigh, University of Illinois at Chicago (`http://www.evl.uic.edu/spiff/`) and Prof Robert Kooima, Louisiana State University (`http://csc.lsu.edu/~kooima/`)

game engines

- *notes from (Ward, 2008)*
- software to support common game programming tasks:
  - modeling and rendering
  - physics
  - user interaction
  - inclusion of assets
  - programming behaviors
- common tools:
  - loading, displaying, animating models
  - collision detection
  - graphical user interface development
  - artificial intelligence
- Ward uses a "car" analogy:
  - the chassis, seats, sound system are part of the *content* of the car

  - the *engine* is what makes the game *go*
- can include:
  - API (application programming interface)
  - SDK (software development kit)
- a little history:
  - in the 1980's and 1990's, companies created their own in-house game engines (e.g., SCUMM by LucasArts and SCI by Sierra)
  - then some in-house game engines were released for more general use (e.g., idTech, the Quake engine; and Unreal, the Unreal Tournament engine)
  - these are referred to as *middleware*—these companies don't actually make games, but they provide the engines to other companies who do make games
- types of game engines:
  - "roll-your-own"
    use publicly available APIs (e.g., OpenGL), but still create their own engines; provide the greatest flexibility to programmers, but are complex to build and maintain; require building a lot of tools from scratch

  - "mostly-ready"
    include rendering, input handling, GUI tools, physics (e.g., Unreal, idTech); but still require developers to create some tools of their own
  - "point-and-click"
    easiest to use, friendliest to (novice) programmers, require not much programming (e.g., Unity, Blender); but can be very limiting

multiplayer games

- *notes adapted from (Leigh, 2007)*
- multiplayer games cannot be designed as add-ons to existing games—must be designed as multiplayer from the beginning
- types of multiplayer games:
  - 2+ players on a single machine/console vs 2+ players on separate networked machines
  - cooperative vs competitive
  - asynchronous vs turn-taking
  - persistent (game continues even when you log off) vs not
- other considerations/challenges:
  - match-making for networked games—recommending good partners/competitors
  - number of players to support
  - role of non-player characters (NPC)—ally or enemy?
  - network latency
  - expansion of game play as players improve

- example: Halo

  - cooperation can be played through the regular storyline of the game
  - when one player dies, the player can be resurrected
  - if both players die, the game can restart from a checkpoint
  - split screen display

- example: Gauntlet

  - third-person view: all players on the same screen
  - cooperative play
  - camera zooms out (up to a limit) when players move in different directions

- example: Quake

  - first-person view: entire screen dedicated to one player (you)
  - cooperative (e.g., capture the flag) and competitive (e.g., free for all) play possible over network
  - cooperation can only be played in special multiplayer arenas
  - characters can be customized using "mods"

- example: Ultima

  - no single-user mode
  - entire screen dedicated to one player
  - all players are on the network
  - large-scale persistent role-playing game (RPG)
  - open-ended play
  - handles hundreds and thousands of players
  - large databases maintain constantly running simulation environment

- real world meets game world...

  Sony Online Entertainment found out in August that when you create a virtual, living world, real-world problems are sure to follow. In one of the year's most interesting stories, players of Star Wars Galaxies staged a demonstration in Theed, the capital city of Naboo, to protest a rash of account suspensions stemming from a "credit duping" operation. Certain users had discovered a way to illegally replicate large numbers of credits (Galaxies' currency), and as a result, the fake credits had permeated Galaxies' economy. When SOE discovered the scheme, the company suspended the accounts of all players who had come in contact with the fake credits. Unfortunately, due to the fact that many Galaxies players are employed as artisans (selling armor and other items) and the common practice of "tipping" other users, many of the suspended players had no idea they had been given duped credits.

  Angry at SOE, a large group of users gathered to protest the suspensions. As the protest greatly hindered the servers' performance, SOE customer service reps began to break up the crowd by instantly teleporting protesters into space. SOE justified its heavy handed tactics by releasing a statement regarding mass protests by releasing a statement on its forums: "Occasionally in Star Wars Galaxies, players feel the need to express their discontent with mass protests. With mass gatherings maliciously organized to disrupt other's play experience, we will restrict the number of players gathering in any one area to maintain the stability of the service."

  The uproar eventually subsided and players were given the chance to appeal their suspensions, but civil unrest is more common in Galaxies than one might expect. Other past incidents include a Wookie uprising in March and a 2002 demonstration on Tatooine in which imperial forces killed a handful of protestors.

  [Game Informer Magazine, issue 142, Feb 2005, p60.]

- fundamental networking issues
  - *protocols*
    (TCP/IP, Transfer Control Protocol / Internet Protocol)
    (from `http://tunnel.mrq3.com/explain/node2.html`, Pearson 2003)

    * TCP (Transmission Control Protocol)
      - connection-oriented protocol
      - a connection can be made from client to server
      - from then on, any data can be sent along that connection
      - reliable—when you send a message along a TCP socket, you know it will get there unless the connection fails completely. If it gets lost along the way, the server will re-request the lost part. This means complete integrity, things don't get corrupted.
      - ordered—if you send two messages along a connection, one after the other, you know the first message will get there first. You don't have to worry about data arriving in the wrong order.
      - heavyweight—when the low level parts of the TCP "stream" arrive in the wrong order, resend requests have to be sent, and all the out of sequence parts have to be put back together, so requires a bit of work to piece together.

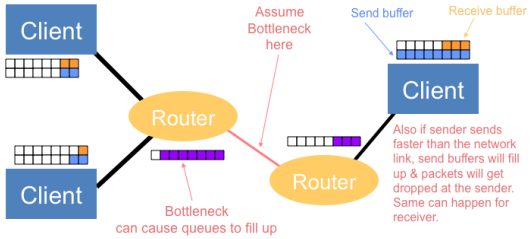
    * UDP (User Datagram Protocol)
      - a simpler message-based, connectionless protocol
      - messages are sent in packets (chunks) across the network
      - unreliable—when you send a message, you don't know if it'll get there, it could get lost on the way.
      - unordered—if you send two messages out, you don't know what order they'll arrive in.
      - lightweight—no ordering of messages, no tracking connections, etc. It's just fire and forget! This means it's a lot quicker, and the network card / OS have to do very little work to translate the data back from the packets.

  - *bandwidth*: measured in bits per second.

- *latency*:
  - big challenge in multi-player games
  - Usually due to: distance; router and end-system overflows; packet loss.
  - Roundtrip times from Chicago: TransUS: 50ms; TransAtlantic: 120ms; TransPacific: 200ms (Measured in milliseconds)
  - latency in TCP is higher than UDP because TCP needs to acknowledge that data stream is correct.
  - 200ms roundtrip latency is considered minimum acceptable latency for tightly coupled interactive applications.
- *jitter*:
  - variation of latency
  - more offensive than latency because it makes it difficult for the player to predict trajectory of an object
  - jitter in TCP is much higher than UDP
  - jitter is usually due to TCP recovering from packet loss
  - smooth out jitter by buffering or smoothing of avatar movements by averaging position values

- *unicast, broadcast, multicast*
  - *unicast* sends message to one specific computer
  - *broadcast* sends message to multiple receivers one at a time
  - *multicast* sends one message and allows router to replicate and send message (based on UDP).
  - The problem is that multicast is not enabled on most routers since it can flood the Internet.
- *firewalls*
  - usually networked applications operate over known port numbers.
  - firewalls prevent unintentional network traffic from entering computer.
  - but firewalls can prevent network traffic from entering you game application.
  - a game should provide information about the port number(s) used so that a firewall can be opened.
  - unfortunately, however, this creates a vulnerability for hackers...

- networking issues can ruin game experience
  - Routers and end-systems have network buffers (queues) which fill up during congestion.
  - Congestion is usually caused by an aggregate of network traffic entering the routers which far exceeds the rate at which the router is able to deliver it.



  - when buffers are full, packets will get dropped.
  - Effect on TCP: latency will increase because TCP will retransmit in order to guarantee reliable delivery and will reduce transmission rate by half to attempt to reduce congestion.
  - Effect on UDP: UDP will simply lose data.

- Is there an easy way to tell or determine what the best sending rate is? Unfortunately not. There is no Quality of Service over the Internet. The best you can do is to assume the core of the network will never slow down and you have to dynamically adjust your sending rate to accommodate the slowest player—down to a minimum acceptable threshold.
- Rate adjustment is commonly used in audio/video streaming.
- For all clients, as a rule, you should process any incoming packets as fast as possible to avoid congestion at the end points.
- But if your client spends all its time processing packets, it may take time away from making the game go!
- So a balance must be struck.

- connectivity models
  - *shared centralized*
    - clients connected to central server
    - simplest to implement and maintain consistency across all clients
    - but load at server can restrict number of participants that can be supported simultaneously
  - *replicated homogeneous*
    - clients with no centralized control (multicast)
    - most scalable solution for large number of clients
    - but multicast is not deployed across the whole Internet
    - used in experimental military VR simulations in the early 1990s
  - *shared distributed with peer-to-peer updates*
    - clients with full connectivity to each other
    - often a central mediator initiates the peer connections
  - *shared distributed using client-server subgrouping*
    - subgroups for areas of interest management
    - most scalable for very large worlds (like online persistent worlds, e.g., Ultima)

- extending game loop to handle networking
  - multi-threaded game loop:
    1. Input Loop Thread
    2. Network Receive Loop Thread
    3. Compute Loop Thread (put network send calls here)
    4. Draw Loop Thread
    5. Sound Loop Thread
- Network Receive Loop:
  - pertains to socket programming (applies to UNIX, Mac and Windows)
  - within a thread:

```
while( 1 ) {
  use blocking select() call to determine which incoming socket
       has data to be read
  read data from ready socket.
}
```

  - if you do not do a blocking call, your while loop will spin and eat CPU cycles...

- network game management issues
  - if a multiplayer game needs to retrieve large data files (like 3D models), then cache the model locally—in the same way a web browser caches images.
  - visual representation of networked entities:
    - "avatar"
      - body position and orientation
      - head orientation
      - pre-recorded gestures
      - body parts list to load at each client
    - share absolute state data, not key or button presses
    - use UDP to share data that is normally sent repeatedly, e.g., avatar's position.
    - With UDP, try to keep packets no larger than 1K bytes otherwise there is no guarantee of arrival.
    - Pack several variables together—don't send one at a time.
    - UDP does not guarantee the order of packets, so include a packet number in the message so that old messages can be filtered out.

  - estimating bandwidth requirements:
    - for every N cycles through the game loop, keep track of how many bytes are sent and received and how much time has elapsed
    - print `total_bytes/elapsed_time` to get an estimate of bandwidth requirements
    - multiply this by the maximum number of players supported, to get an upper bound estimate.
    - It is useful in your program to be able to print send and receive rate so that you can observe your game's network consumption during runtime.
  - reducing bandwidth requirements:
    - *dead reckoning*—send position, velocity and acceleration at every N loop cycles rather than every cycle
    - send only data that has changed

- example: implementing someone firing a bullet
  - when a bullet is fired, broadcast a message like:
    `BULLET position velocity bullet_type`
  - when clients receive the message, they add the bullet to a list of agents/entities they have to manage—like any other bullet in the game
  - each client computes how the bullet should fly
  - each client also figures out if a bullet has hit the client and reports this to everyone else
  - i.e., the client is the one that determines if it is dead

- handling "late joiners"
  - players who join the game while it is in session
  - the "late joiner" joins by first connecting to the main server.
  - server introduces the other clients to the late joiner by giving the other clients the IP address of the late joiner.
  - each client creates an entry in their data structure for the new joiner
  - each client connects to the late joiner
  - late joiner creates entries in its data structure for all the clients.
  - late joiner sends data about itself to everyone—in particular the body parts that make up the joiner's avatar
  - all clients send data about themselves to the late joiner.
  - this is why it is important to always send state information, not button presses.
  - either allow the late joiner to start playing in the game using incomplete information (e.g., if you don't have a head yet, use a temporary head) OR wait till all the clients and late joiner are fully synchronized and then allow the late joiner to start playing

- handling "zombies"
  - if someone disconnects, make sure to remove them from your list of avatars
  - since UDP is connectionless, it is not possible to tell if someone has "disconnected"
  - so for every peer, make sure to establish a TCP connection too—so that if it breaks, there is an easy way to know.