Description:
Class notes pertaining to the course *COMP521 – Modern Computer Games* at McGill University, taken January – April 2007.

Author:
Nicholas Rudzicz
nrudzicz at hotmail

COMP 521 – Modern Computer Games

January 4, 2007

- Games we play now
- Combinatorial games
  - Chess, checkers, etc.
  - Lots of research
  - 2-n player, *perfect information*
  - Applies to many board games
  - ...but these often start introducting imperfect information
  - Winnability – who can win, and how?
  - Complexity – NP or P space? Can show complexity of certain features. Solve sub-features, make approximate solutions.
  - Game tree – states representing different actions, arranged in a tree. Path finding, strategy, and other problems in game design can be seen as variation on same structure.
    - See J. Romain's solution of *Awari*
- Game Theory
  - Used in economics
  - Prisoner's dilemma (will kill one of you if you say, both if agree, etc.)
    - Both quiet -> Get reward.
    - One quiet, other confesses -> Confess gets big reward, quiet is punished
    - Both confess -> Both punished.
  - Optimal strategy? *Could* be applied to games (as opposed to econ.), but hasn't been done well yet.
  - Modern games have intractably large/random environments.

Modern Computer Games
- Not graphics.
- Limited AI.
  - Game AI slightly different from formal AI. Resource constraints are *key* in games. Attention paid to *appearance* of AI – imperfect simulations that still seem intelligent and are fun.
- Intersection of Ludology and Narratology
  - Ludology – no narrative (i.e. arcade games, card games, etc.)
  - Narratology – duh. Narratives and storylines.
- Intersection of: Graphics, Game AI, Combinatorics, Complexity and Algorithms, Distributed Systems, Networking, Simulation (don't worry too much about correctness), Databases, Security, Geometry, Testing/Software Engineering, Formal Verification.

The Games Industry
- *Industry*.
- Large industry at that. Enjoy stable income.
- Subscription model. Desirable. Keep people paying more, instead of one easy payment.

January 8, 2007

Industry Problems
- Application complexity
  - Artists, developers, testers, designers, etc. How to keep them all working at the same time; have

    some sort of SE pipeline?
  - What is optimal workflow?
- Implementation
  - Games are LARGE projects! Size, features, etc.
- Software Development
  - Rapid development! Race against other companies.
  - Lots of design changes! Prototyping!
- Very little software reuse
  - Often motivated by latest technology
- Improvements to visuals
- Performance is critical
  - Graphics suck up CPU, everything else gets small CPU budget
  - Can we come up with shortcuts?

<u>What is a game?</u>
- In combinatorial games:
  - 2 player
  - Perfect information
  - Cannot be ties (always a winnar)
  - Specific rules
  - Alternating gameplay
  - Etc.
- Modern computer games:
  - Sid Meier -- "A game is a series of interesting choices." Whoa man. Whoa.

<u>Elements of computer games</u>
- From Chris Crawford (see *Art of Computer Game Design*):
- A closed system/representation – self-contained set of rules/boundaries. Little need for external references.
- Interaction! Must respond to user.
- Conflict. Some obstacles to goal.
- Safety. No RL consequences.

<u>Genres of computer games</u>
- Again from Crawford
- Skill & Action game
  - Combat: Asteroid, Missle Command, Space Invaders...
  - Maze: Pac-Man
  - Sports: duh.
  - Paddle: PONG. Arkanoid.
  - Racing.
  - Misc.: Donkey Kong, Frogger
- Strategy
  - Adventure games: Scott Adams games, Colossal Cave, Interactive Fiction
  - D&D
  - War games: Avalon Hill-style games
  - Games of chance: (?)
  - Educational
  - Interpersonal games
- From Mark J. P. Wolf (2002)
  - 42 genres. Isn't point of genres to come up with a limited set of subsuming categories?

  - More of an arts perspective
- Abstract: Rez, etc.
- Adventure
- Capturing – capture things that are evading
- Catching – capture NOT evading things
- Collecting – wtf?
- Dodging games
- Escaping games
- Demo - ??
- Lots of criticism – 42 is too many, somewhat arbitrary divisions.
- Note (again) that genres are often organized around a particular technology
- From [Rollings & Adams] (available online through McGill)
- Action
  - FPSs – 3D world, 1st person viewpoint, ...
- Strategy
  - Boardgames as an origin.
  - RTS vs. TBS
  - 4X games – Expand, explore, exploit, exterminate!
- RPGs
  - Very much like adventure games
  - Difference? RPGs have character advancement. Paper doll, whut.
- Sports games
- Vehicle simulations
- Construction/management systems
  - Interesting. No concrete goal (Quake vs. Sim-City/Railroad Tycoon). Games vs. toys?!?
- Craig Lindley [2003 Game Taxonomies] (See Gamasutra)
- More abstract look at game types (why have categories as above, when they'll be obsolete in 10 years?)
- Comes up with certain dimensions/scales
- Triangle – 3 points: Ludology (arcade-like games, no story; Chess, Pac-Man); Narratology (IF – duh); Simulation (Sim-City, etc.)
- Games can vary according to these 3 points. RPG might be halfway between all, etc.
- This triangle makes the base of a 3D, 3-sided pyramid. The top of this pyramid is: chance/gambling (dice, lotteries, etc.).
- Virtual economies are b/w chance and simulation, e.g.
- Two more dimensions (however you'd wanna draw them):
  - Fiction vs. Non-fiction
  - Virtual vs. Physical

<u>Fun in games</u>
- Several studies in arts
- Nebulous. Duh.
- Play can be divided:
  - Agon: competitive play
  - Alea: chance-based
  - Mimicry: simulation-based
  - Ilinx: Physically-based (sports, etc.)
- Consider whether your game fits one or more categories. Who will it (will not) appeal to.
- “Flow”
  - psychological term from Csikszentnihalyi [90s]
  - “Getting into” an activity.

- 1) Task that can be completed.
- 2) Ability to concentrate; captivate attention
- 3) Clear goals
- 4) Immediate feedback
- 5) Sense of control over actions.
- 6) Concern for self disappears (returns later & stronger)
- 7) Deep, effortless involvement, removing external awareness
- 8) Sense of time/duration is distorted

January 11, 2007

Flow in Games (Sweitzer – See flow handout)

- Principle 1 – the game
- Concentration
- Challenge player's skills – balancing act; continuous rebalancing; different difficulty levels.
- Principle 5 – Control (nice user interface)
- Principle 3 – Clear goals – player must find goals, and find how to get there. Hint systems
- Principle 4 – Feedback
- Principles 6,7,8 – Immersion
- Social interaction

Immersion (Ermi and Mayra handout)

- Sensory immersion (eye/ear candy) – like a screensaver
- Challenge-based immersion – puzzles; satisfying blend of skills & challenges
- Imaginative immersion – imagine that one is “in” the game; thinking as the game character, within the game's constraints, etc.

Important Game Property

- *Illusion* of winnability
- Player must feel that a loss is due to their own (correctable!) failing, not the game's.

Player Types (Bartle Player Type handout)

- Multi-user Dungeons (MUDs)
- Ecology of players.
- 4 types
- Spades
  - Explorers
  - Not necessarily wanting to finish the game/interact...just explore
- Hearts
  - Socialites
- Diamonds
  - Achievers
  - Maximize their numbers!
- Clubs
  - Killers
  - PvPers
- Type revision?
- Make 2d graph.
- X-axis goes from -x (Player) to +x (World) – i.e. what player focuses on
- Y-axis goes from -y (interacting) to +y (acting).

- Acting+Player == Killer
- Acting + World == Achiever
- Interacting + World == Explorer
- Interacting + Players == Socialite
- Nifty ecology diagram!
- 4 stable configurations (approaching Game Theory?):
- Type 1 – killers/achievers in equilibrium, very few socializers and explorers. Achievers are targets for killers (i.e. to get ninja'd/pwnd)
- Type 2 – Socializers in dominance, everyone else in bit parts
- Type 3 – Balance of all 4 types, enough explorers to control the killers (?) (Explorers set aside; hard for killers to find them? Explorers are too poor?)
- Type 4 – Empty virtual world.
- *Not* stability of a given player's experience; that can be unstable at points (indeed, the fun is derived from this). Stability of *entire system*.

<u>Rotated component matrix paper – Alix [2005]</u>
- Survey of hardcore gamers
- Based on large set of questions
- Look for groupings/clusters
- Warriors
  - Fighting/equipment/combat. (~killers?)
- Narrators
  - Plot, characters, exploring world (~explorers? But explorers aren't necessarily interested in plot)
- Strategists
  - Challenges, challenging gameplay, mastery over the game (~Achievers?)
- Interactors
  - Socialites. Duh.

<u>Typified players [Yee 2005, see online slides]</u>
- Another survey/analysis
- Again looked for clusters/principal components
- Three groupings, but those broken down further
- Achievers
  - Advancement – progress, power, ...
  - Mechanics – Numbers in the game, optimizers
  - Competition – rough analogue of killers? Challenging others; domination.
- Social
  - Socializing – talking, helping others
  - Relationships – personal, support (in the game)
  - Teamwork – guilds, raids; collaboration/group behaviour
- Immersion
  - Discovery – exploration, finding hidden things
  - Role player – roles, fantasy
  - Customization – Change character's appearance/behaviour; accessories, colour schemes
  - Escapism – relax, ignore outside world

<u>January 18, 2007</u>

<u>Game Design in General</u>
- various guidelines in "practical" texts
- Generalities:
  - I/O structure – computers are limited, one does not start walking to make character walk. Limited interface.
  - Game structure – how does the game "work", what does the player have to do, how do they do it, etc.; core mechanics
  - Program structure – actual construction of game
  - Often treated more like a movie – require different documents; levels of treatment – High-level "pitch", medium-level (how many artists/programmers, etc.), low-level (actual script, UI, design, etc.)

<u>Programming (SE challenge)</u>
  - Different teams: programmers, level designers, artists (visual/audio), testers
  - Cyclic work flow: lots of prototyping/revision
  - [Rudy Rucker] (Game SE model): Requirements; Architecture; Specification [1..n] & detailed descriptions; alpha-n version (cycle back to specification step); Final design; Beta-n version & documentation?; Testing & Debugging (cycle back to Beta-n step); Final version!
  - Game maintenance?

<u>Other SE models</u>
  - Key idea: *Lots* of prototyping/repetition
  - Large state machines: core update loop
  - Typical:
    - Step – a discrete chunk of time: adjust game parameters (adapting, checking win/lose); listen to the user (user input); player move; non-player moves (AI); Collision detection/response (game physics); Cleanup (characters change/disappear – get rid of them, but in mid-action/dialogue? Delete events?); Animation (graphics/sound/text responses);
  - Event-based (from simulation):
    - Priority queue of events
    - Event: action + timestamp; process earliest timestamp
    - Processing an event could generate more events.
    - In most cases, update-loop performs better because it is more specific/rigid.
  - Cellular automata
    - Divide world into grid (tessellation)
    - Update loop: process all cells; have some number of fixed neighbours; updates work on local sphere of information; change own state based on neighbourhood.
    - Sim-City an example.

<u>Narratives</u>
- Lots of games are story-centric
- Certain genres more than others: adventure games
- Story reveals itself as the player plays, player actions *guide* the story
- Text-based – interactive fiction (IF)
- RPGs
  - Story from adventures, plus character development
- Some FPSs
- Arcade? Aliens are invading. Wow. "Emergent" narratives? Player actions generate narrative

through own imagination.

What is a good story?

- ...and how do we write one?
- Formulaic (see EA)
- Classic text: [Georges Polti 1921]
  - Divide all narratives into 36 catagories/templates
  - Based on "fact" that there are 36 emotions
  - High-level perspective
  - Examples:
    - 3) Revenge story: roles (avenger, criminal)
    - X) Conflict with the gods, etc.
    - ...And some older ones?
- Hollywood style screenwriting
  - Hero's journey
  - [Vogler 1993]
  - See Hero's Journey handout
  - Most time spent in Step 6 (Tests/allies/enemies)

What makes a good *game* narrative?

- Must be an interactive narrative
- Player is making choices (recall "A game is a series of interesting choices")
- Early models were just very linear chains of events.
- Excessive linearity isn't the most interesting
- Another model: several *main* events, provide choices along the way. But choices don't really matter – they all lead to the same place (meaningless choice).
- Make a huge branching tree. Infinite choice. More interesting to player (?). Overwhelming. Huge problem for developer/designer workload.

January 23, 2007

Good game narratives (cont'd)

- Graph of efficiency/effort vs. time spent on the game
- Divided into thirds...bottom third is too easy, top curve is too hard. Middle road is le funn.
- Optimal is to have many branches, then all collapse together again (??) -- gets more difficult, then less difficult (i.e. growing-shrinking)
- Most games look like the sqrt(x) graph...game starts hardest, then levels off. Once mastered, a game is easier, and slides into the easy area of the graph (esp. longer games)
- With branching/merging quest tree, we try to be bounded in the good envelope of the graph
- Unfortunately, this ideal difficulty is hard to measure.
- Need a formal way to present/analyze narratives.

How do we represent narratives?

- Classically – graph of Tension vs. Time, looks like a Gaussian. Gets hard in the middle, thin again at both ends.
- Existing IF technologies – plot D.A.G. (NO cycles, duh)
  - Nodes – represent game states
  - Edges – choices connecting one state to another
  - Good for simple game narratives, but not very good for complex stuff
- May have cycles in graph (door goes locked <--> unlocked)

- But a cycled graph (locked <--> unlocked, etc), how do we know initial states?
- Better solutions:
  - Narrative languages (like programming language)!
  - Formal models – logical and/or operational
- Why formal models?
  - be precise about ideas like convexity
  - narrative analyses and matrices
  - would be nice to show that narrative is correct

Narrative Flaws
- Pointlessness – if the game is not winnable, must make player lose within a short period of time
- Unexpected scenarios – hard to keep track of all possible player actions & reachable game states. Designers didn't anticipate these actions, can lead to arbitrary behaviours (narrative crashes)
- Non-sequiturs – characters/events don't behave logically (e.g. complete quest before actually encountering events that are supposed to give you the choices)

Logical Representations
- Describe the narratives in a given system (e.g. as a series of deductions)
- Query it for flaws
- Ex.: Pd = player @ dragon lair; Pw = player @ wizard; D = dragon dead; Q = player has quest.
  - Rules: Pw -> Q; Pd -> D; etc.
  - Can verify if game states are consistent.
- Petri Nets (look in Wikipedia)
  - lower-level
  - provides an operational model, still formal and analyzable
  - cycles/choice
  - can represent resources
  - can still grow exponentially

In practice
- 2 systems for narrative games
  - Inform v.6: www.inform-fiction.org; Inform games are compiled to binary files. Typically extra library fn's are incorporated to provide language parsing and other fn's. Game info + libraries -> Inform compiler -> game.z5 -> Z-machine, which checks it (?)
  - A research system that is much less feature-rich, but which provides a single useful environment for game analysis; PNFG. (gram.cs.mcgill.ca/nfg.php). Game source -> compiler -> interpreter -> player. PNFG system is different from Inform in that: language for games is simplified; output is in terms of a Petri net model of the game narrative (specific structure/analyzable); NFG interpreter is also simple, but not game-specific
  - In both cases: player directs an avatar in a virtual game; interaction is turn-based and command line; player types a directive each turn; world composed of rooms & objects (rooms are just like big objects that contain other objects) – room contains desk contains box contains key – player contains bag contains dwarf contains love.

January 25, 2007

Review
- Narratives: convexity, narrative shape (i.e. grow, collapse, etc.)
- Narrative flaws & analysis: games based on narrative (IF, adventure, RPGs, ...): avoid pointlessness, etc.; analysis: logical approach (exponentially awful); operational (petri net – PNFG [Verbrugge], [Natkim & Vega]); cyclic behaviours, choice must be reckoned for.

<u>Two Narrative Systems</u> (see “Narrative code” handout)

- Inform (v.6)
  - Established, many examples, good documentation
  - Language superficially similar to OO (can define objects, have (multiple!) inheritance), but not really. Messy language. Requires 5 tokens of look-ahead (see compiler courses) – most need only 1, ye Gods!
  - Properties can be associated to objects, incl. boolean variables (“attributes” -- e.g. light switches, carrying object X, being openable/is open)
  - Commands: directional (NSEW) – map of world; look X; take Y; etc.
  - Functions: associated with objects (or global); all functions return something; often true/false (on handout, n_to has string “...”, Inform emits string, returns True)
  - All rooms are objects
  - Some commands are implicit: take/drop/etc
  - [Initialize _____]: set up the narrative (start location, start inventory, place objects in map, etc); then enter wait loop, listen for user input; execute appropriate command; loop.
  - Natural language input; natural language (external) library converts human input into canonical, simplified, unique form
  - Constants: 16-bit!
  - How to win/lose? Deadflag -> 2 = win? 1,3 = lose?
  - Inheritance specified by order you specify objects...pay attention.
- PNFG
  - Objects
  - Rooms
  - Containers
  - States
  - Very low-level compared w/Inform – have to build in all features; very simple language
  - “Start” action; +obj.state, -obj.state
  - Move objects: “move x from y to z”
  - (you, e) – i.e. player moves East
  - Sets: “x = {a, b, c}”, can query, “x $z ... if(you in $z) {move you from $z to bar}”
  - Much less docs than inform – an M.Sc. thesis, a conference paper, that's it. Dang.

<u>Game Engines</u>

- History
  - Doom/Quake instrumental in commodified code reuse, game engine (re)use in similar games.
  - Have engine, feed it world description (textures, models, sounds, logic, etc.), produces playable game. Game development goes into world description.
  - Game engine itself is basically a graphics pipeline/engine
  - Input: world description. Work: High-level geometric transformations (world->screen); triangle preparation; renderer (fog effects, lighting, shadows, etc.).
  - Other components: sound; physics; network code; input; AI (fast heuristic search routines, ftm);
  - See devmaster.net for game engines.

<u>How to Construct Virtual Worlds</u>

- Most games (esp. FPS, RPG, simulation) have some kinda environment
- Outside (wilderness, jungles, plains, deserts, etc. -- mountains, rivers)
- Inside (houses, offices, underground (dungeon), etc.)
- Large or small scale
- 2D virtual worlds (Risk, etc.)
- 2.5D: heightmap, Fallout, Doom, etc. Pushed up from the ground, no flying stuff – like a pure

cityscape
- 3D world – things can float, fly, hang over things, etc.
- How to create? Hire artists – best results, but expensive, slow.
- Library of external sources: G.I.S. of most of world for FlightSim, etc. (can even find Mars).
- Building it algorithmically?
  - Structural/abstract? Relations between objects in the world, but no visual representation. Good for text-based games. Also in FPSs. Basically, building a map/maze (trivial, nothing; map with randomly-generated obstacles (Centipede); connected rooms/areas – i.e. a maze)
- How to build a maze?
  - Different kinds of mazes!
  - Simple/perfect maze: only one path to solution; no cycles; all areas are reachable; ends up being a tree!
  - Braided maze: no dead ends, but possibly loops; take a simple maze, and connect dead ends
  - Simple < --------- > braided; a continuum. Can make games more difficult by making "more braided".
  - Unicursal: no braching, just a long path/tube.
  - Properties of mazes:
    - Amount of "branchiness": many branches, vs. long runs.
    - Cover all areas?
    - Unique solution?
    - Solution traverse all/most of maze? Don't want a shortcut to miss 99% if you put lots of effort into it
  - Several *simple* algorithms for maze generation
    - DFS: Start with grid (any shape/tiles); choose random start; look at neighbours of current node; randomly choose neighbour that's "not part of maze" (initialized to just be start pos), randomly connect them; recurse.

<u>January 30, 2007</u>

<u>Maze Generation</u>
- Mazes: "room"/location connectivity; often grid-based (cells as nodes, edges indicate connectivity)
- Creation algorithms:
  - DFS – often very long, initial run, with small branches only
  - Prim's Algorithm: Some nodes are "frontier" nodes (i.e. unconnected neighbours of connected nodes); can choose uniformly to add any of the frontier nodes (approximately breadth-first), rather than just those neighbours of the most recently-added node (a la DFS); much "branchier" than DFS; can give bias weights – towards latest node, get DFS, or away from latest node, get branchier one; Prim's algorithm "grows" the graph
  - Kruskal's Algorithm: Start with all nodes, no edges; start "sprinkling"/adding edges between unconnected nodes to graph; note that for perfect maze, we need a tree graph, and thus cannot create cycles – *we need to verify whether adding a node will create a cycle*, which may take some time; can brute force the check, or use disjoint set structures to make this more efficient – each connected component is a set (initially every node is a single component), and we merge sets that are in different but adjacent components, until only one set is left. Bias towards connecting larger components?
  - Aldous-Broder/Wilson Algorithm: Assume connected graph already; choose random unconnected point, then perform random walk until connected to rest of graph (initially, one point is "connected", and another point will be randomly chosen – leading to longer initial runs, but nice overall randomness

Incorporating Mazes into Games

- Many games would prefer some things maze-*like* rather than a perfect maze
- Can we use a “sparser” maze?
- Don't need to generate perfect/tree-like mazes – cycles/dead ends might be desirable, encourage exploration
- [J. Buch] Start with a perfect maze; clean it up – pick some dead ends to shrink; results in longer main run with small branches on the side (this is a parameter you can adjust); add in a few cycles – find some dead ends, do a random walk until we hit the graph again (sorta like Alders-Broder); then add random rooms in “empty” spots – i.e. open areas that don't have paths through them – and connect them to the path with a doorway

How do we measure mazes?

- Difficulty?
- Pointless regions? Players may not explore whole dungeon, thus obviously game events must happen at specific “chokepoints” that a player *must* visit; other areas players *may* (but might not be on THE path to winning) visit shouldn't have game-necessary stuff in them; other branches, players don't *ever* need to visit

Actual Terrain

- Lots of terrain out there (incl. inside of buildings, cities, in objects, etc.)
- Focusing here on world-like, large-scale terrain
- GIS is one way of doing pre-existing world; or copy-pasting the Rockies into Oblivion, e.g.; however, very inflexible
- Tile-based approaches:
  - Have an artist “paint in” tiles representing different terrain – this is lots of work, and again is slightly inflexible
  - Randomly assign tiles: often looks pretty damn unnatural – no correspondence between adjacent tiles; okay for some things (close-up view of a forest maybe), but not for larger worlds; add constraints between tiles, but then you could be left with one tile to place, but nothing “fits” constraints in there
  - Randomly place, then do iterative refinement: permute tiles, do replacements, etc. to clean up the grid
  - Image processing: throw noise onto graph; do blurring/averaging (akin to anti-aliasing) to get smoother/more continuous world; usually works out pretty well.
- Fractal techniques
  - “Fractal” often used to indicate “recursion”
  - Use recursion to generate structures with interesting features at arbitrary/multiple scales
  - Midpoint Displacement [Fournier et al. 82]
    - Draw a line, pick a midpoint (or more generally a bisector)
    - Displace the point by a random amount along the normal to the line, up or down
    - Recurse over remaining two line segments
    - Stop when no longer interesting or displayable on screen.
    - Often gives nice mountain-like terrain
    - Note that amount of displacement should be proportional to length of the line (i.e. small segments do not get shifted by huge amounts)
    - Displacement could be uniform (tends to jagged terrains), or, for instance, Normal distribution (smoother terrain, esp. at fine granularity)
    - For 2D planes (as opposed to lines), we can do the same with triangles (note that with squares, we could inadvertently twist the square – triangles always form a flat plane); could recursively find midpoints and getting smaller triangles (a la Triforce) and randomly shift

        them up/down; but note that we could have shared midpoints, and must not have two triangles trying to displace in different directions; further, these shared midpoints have two or more different normals along which to displace – either one will skew the other triangles; do we take the average normal? Start with a given structure, and *then* recurse based on this canonical normal?
    - Faultline method: randomly draw a line – one side higher, the other lower; recursively draw these lines and eventually get nice world.
- To “finish” the world, we could add water
    - Choose a water level and that's it; flood the world – parameter for how much land/water you want, then use altitude to determine decoration (i.e. snow, etc.)

<u>February 1, 2007</u>

<u>Perlin Noise</u>

- Make a simply-drawn structure look more natural
- I.e. Draw upside-down V for a mountain – abstractly is a mountain, not very natural
- Drawing simply random points is too jagged – but does have randomness (a bit too much, uncorrelated)
- Add simple drawing and random drawing! A bit more natural – not quite perfect (still jagged)
- We want a more “natural” noise, with some correlation between noise points
- Perlin Noise – correlated noise!
- Multi-level noise: one base level of noise, then add “harmonics” / “octaves”; generate random (elevation) points in space; smooth out noise (linear interpolation aka straight lines between points, cos/sin interpolation, higher-order polynomials which aren't always worth the extra effort). But we want more detail! Generate more points! With double the frequency, but half the amplitude (i.e. a harmonic) – less drastic, but finer structure. Keep halving amplitude, increasing frequency (exact ratios are parameters) – graphs of smaller and smoother curves. Keep on going only so far as what can be represented *well* onscreen. Add up all the functions. Now have large structure with fine-grain, correlated randomness. Can be used for many features – mountains, clouds, texture generation (make a regular pattern look like an interesting, natural irregular pattern)
- NB: Level-of-detail – how much detail do we represent? World should support multiple LODs.

<u>2D Graphics, 2D Games</u>

- Many games, esp. arcade/abstract games, offer 2D (overhead, isometric, etc.) views. Side-scrollers, etc.
- Representing the world: a bitmap; vector graphics; little bitmaps – game elements
- Sprites! Little bitmaps, huzzah. Use like rubber stamp, or give unique identity
- Most 2D games are based on sprites – game engine becomes 2D sprite engine
- “Rubber stamp” leads to memory savings (re-use same sprite over and over)
- Engines can use: rectangular bitmap w/alpha; transformations (shrink/grow, skew, rotate, etc.); animations (several sprites in a row – must specify location *and* frame)
- Some game hardware directly supports sprites; sprite maps (program simply has array of sprites, producer-consumer paradigm, sprite manager pumps these sprites to the screen). Or screen has a sprite representation instead of a pixel representation – each index shows which sprite, not which pixels, go there – but this is coarse, as sprites can only be in grid cells
- Most 2D games actually have multiple “layers” -- at least, background and player; background is bitmap or sprites (static – may not need transparency), but player must be a sprite, and probably need transparency
- Others have even more layers: background (split? distant, closer, etc.; for parallax), moveable objects (can be many more layers here), then player.

Movement & World Design
- In side-scroller, what happens when you reach the end of the screen?
- Conceptually, like a long film strip, with one small window on it.
- What designs make sense?
  - Made up of "rooms", no scrolling – a la King's Quest III. Triggered by events/player movement. Can make background images "match up" for visual continuity
  - Continuous movement – centre-based approach, where player is dead-centre, and world moves around him; balances how much the player sees around them (the "rooms" approach puts limits to left and right)
  - Can we do something more natural with rooms approach? Bounded movement – margin on either side to define when "page" gets flipped (a la KQIII) or shifted (a la previous); be careful to avoid "thrashing," i.e. you flip the page and player gets placed back at other boundary, page flips start recurring due to small player movement – shift margins so that they are asymmetric
- Infinite world? Just have very large (finite) bitmap – but where to store? Create world dynamically (using terrain/maze algorithms) – but loss of consistency, and takes a long time to process. Both methods require a buffer, advance warning of player movement, speculative loading (though we may be wrong, and player might turn back). Have double-margin – one for actual screen movement, and a larger one to indicate when to preload next screen.

February 6, 2007

Resource Budgets
- Game loop continues without end
- Need to estimate/measure the time for each component (AI, physics, network, etc.)
- This bounds the update rate of your game
- Aim at frame rate – faster, the better. But why go faster than the refresh rate of the screen? Upper limit. Is there a lower limit?
- Movies – 24fps. TV – 30-60fps (depending on region).
- Refresh rate is different in games vs. "analogue" methods.
- Pause a movie – may see blurring; but pause a game, we see "still in time" since each frame is discretely rendered. Motion blur in movies lets you get away with slower frame rate – but games require higher (discrete) frames.

Concurrency in games
- Many discrete elements/processes in games
- Better performance when things done in parallel
- Rather than serial game loop, have multi-threaded game loop
- Structured parallelism – i.e. distribute/grow, then join/shrink. They all pass through specific "chokepoints"
- XBOX360 – 3-way PowerPC.
- PS3 (cell) – 1 big CPU (PPE – Power Processing Element), 6-8 "slave" (SPE) CPUs. Big one used for directing & co-ordinating little CPUs. Can set up independent pipelines. Pipeline with all elements active, where processors are in stream.
- Pipelines: producer/consumer paradigm. Must make sure consumer's ready to consume, that they get everything in the right order, etc. Make sure data from P is correctly sent to C, in order, no loss.
- Pipelines use basic locks, i.e. *critical section* which enforces mutual exclusion.
- Producer: Produce, lock, store data, unlock.
- Consumer: Lock, get data, unlock, consume/process.
- Still require a buffer or a flag to say that consumer has taken data.

- Lock, check if buffer is free, if not, unlock (no producing). Spinning is expensive, but needed for correctness.
- Producer/consumer pipeline can only go as fast as the slowest node.
- Cell processors are very complex systems. BAD FOR GAMES POO POO. Not just 3 simple machines on which to run things – have to optimize for the pipelines, partitioning & load balance issues.

<u>Collision Detection</u>
- Multiple problems together
- Basic idea: need to find when/where/how game objects collide. Object interactions/game world boundaries.
- Main problem: *Detecting* collisions
  - Where collision occurs (world, or player body, etc.)
  - Exactly what time
- Secondary problem: Collision *response*
  - How objects respond to the collision
  - Hard, since objects are rather irregular.
- For ideally "realistic" collisions, need *lots* of CPU power, but have a limited CPU budget.
- Need approximation techniques, trick the user, har har
- To do CD: object representation (boundaries), what questions do we need answered (friction? elasticity? exact point of collision, or boolean yes/no?), need environment/setting (size of world, number of objects – simple world can use brute force, realistic world can't, must be MUCH more efficient), performance, robustness (numerical error/stability of floating point numbers), complexity of implementation (more complex, more time spent programming, more bugs).
- Representation
  - Polygons: points and straight line segments; simple polygons may be concave, but all points within it are connected; reduce complexity futher by thinking of convex polygons
  - Simplest convex shape: thhhhheeeeeeeeeee TRIANGLE
  - Could also consider solid geometry: "doughnut joined to a box", reduces numerical error, can mathematically model these "blocks" for better accuracy.
  - Object representation can be different from visual, as CD req's are different from graphics req's, so we often have 2 representations, which must be kept in sync (but often are not).
- Queries
  - Boolean query – i.e. for (most) boundaries.
  - More detail – two boxes, where/when do they collide? Find region of intersection, aka contact manifold – intersection of 2 convex objects a bit easier (thus why we like triangles), non-convex is much harder. But we still only need an approximation – do they collide, and if so, how much?
  - Knowing approximately when.

<u>February 8, 2007</u>

<u>More collision detection queries</u>
- For most queries, we will approximate things – keeps cost down, but still looks "reasonable"
- E.g.: Two shapes aren't colliding in one time frame, then they are in the next – need to find out where they contacted; often have to "undo" time steps to find out point of contact (could do this analytically, but often complex).
- Is movement simultaneous or not? In RL, two boxes can move with same velocity and arbitrarily small separation, and never touch. But modelling this in a game is hard. We often use sequential modelling instead, i.e. move one box, then the next box. Can cause "following" box to collide with other box – a collision that shouldn't actually exist, "spurious" collision. But we don't need to make

it simulation-quality, can keep sequential movement anyway.

Dynamic CD (i.e. continuous vs. discrete)

- Discretely, we can "miss" collisions, i.e. fast-moving box passes thin barrier between time steps; at each time step there is no collision, but the box has moved through the barrier!
- *Tunnelling!*
- Barriers must be wide enough that the fastest object cannot pass through it in one time step: i.e. minimum bound on object sizes, or speed up simulation time step, or bound max. object speed; that is (m), (s), (m/s)
- Another approach: "swept volume". "Smear" the object between its two endpoints, get all the points it occupied during the time span. Detect collision using these swept volumes, but if two swept volumes collide, how do we know if they met at same point at same time? They might have just missed each other, but we'd still declare a collision; but on reasonably small time frame, it's okay. Can break swept volumes down into primitives (i.e. a rectangle and two circles for a moving circle)

Bounding Volumes

- Testing a complex shape for collision can be expensive
- Can break down shapes into simpler components – concave shape -> collection of convex shapes
- Can approximate even more by finding a reasonable, *simple* bound on object
- Different choices:
  - Box around everything: bounding square/prism, easy computation (check lines crossing, and degenerate cases where box is inside another box); what happens if two boxes simply touch?
  - We've been thinking of axis-aligned boxes from the start, but objects may rotate. "Fit" of a bounding box is also critical for accuracy.
  - Allow bounding boxes to rotate; but now CD is much harder, as lines no longer parallel to axes! Overlap can be detected in 3D by projecting onto 15 different axes
  - Circles! No complexity in calculating points, axis-aligning. CD with circles easy: just find centre-centre distance, and compare against their radii. But still a fit issue.
  - Convex hull! Usually a better fit; complexity roughly equivalent to sorting (i.e. O(nlogn)) -- quickhull, graham scan – which is do-able at run-time, but better (esp. for more complex shapes) to preprocess. However, convex shape intersection with more and more points is still difficult.

Co-ordinates and Computation

- World co-ordinates, makes sense since two colliding objects are in the world. But the world is often very large – how many bits can you devote to representing one scalar value? Floating point numbers have uneven distribution along number line, more dense near 0, thus we want our co-ordinate representations to be centred around 0 (i.e. not "he is billions of miles in this direction", etc.)
- Object co-ordinates: translate one object into another's co-ordinate space, thus co-ordinates are centred around 0.

February 13, 2007

Collision Detection (cont'd)

- Circles: CD very easy (distance between centre points compared to sum of both radii);
  - use std distance formula between two points (d = sqrt(delta(x)^2 + delta(y)^2), where delta(x) is distance between centres along x-axis), how do we do sqrt efficiently? Hardware/software? In software this is very expensive, in hardware it's cheaper, but still much less so than add/sub/mult/div/etc.
  - Notice, to do CD, we can do everything in squares – d^2 = (delta(x)^2 + delta(y)^2 – no need

for sqrt().

    - For ideal circle we want one that minimizes "wasted" space. Easy way: make a bounding rectangle, make a circle that minimally bounds that rectangle. But why add extra wasted space outside of rectangle? Brute force: think of shape as a cloud of points, we'll need two (i.e. exactly opposite) or three (define a circle's curve) – then check all pairs and triplets of points to see if they define a bounding circle (i.e. all other points are within the circle), take smallest such circle. Wow. Slow – O(n^2) for pairs, O(n^3) for triplets, O(n^4) when comparing against all other points.
    - Could use heuristic [Ritter '90]: look for a "good" starting/centre point, expand circle until all points are enclosed. Computing "good" centre: mean of all points? But is biased toward centre of mass. Really want the geometric centre, Ritter's alg. tries to approximate this. Find and enclosing bounding box (i.e. min/max co-ords in each axis), consider all extremal points that define bounding box and find all possible pairing of these points; get furthest apart & use this as diameter, choose point (centre of longest edge, or centre of BB). Now, check each point, expand radius until this point is enclosed. As opposed to "draw box" above where circle is drawn around rectangle, in Ritter's alg., circle is drawn *inside* the box and expanded.
    - [Welzl '91] works well in practice, but bad worst-case. Idea: can remove a point, find spanning circle for the rest, and if that SC contains the original point we're good; if not *we now know* that this point must be on the perimeter of the minimum SC. Ugly algorithm (two recursive calls), but works decently in practice:
        - if(#pts == 0) create new circle
        - set-of-support (<= 3 pts that define circle)
        - if s-o-s = 0, null circle
        - if s-o-s = 1, one point-circle
        - if s-o-s = 2, get diameter from pts.
        - if s-o-s = 3, get definition of circle
        - *these are all base-cases*
        - otherwise: choose point p, and remove from open points.
        - recursively find minSC of remaining points
        - p inside this circle? MinSC is good, add p back in and return.
        - Add p to incoming s-o-s (not one that was recursively sent from here), another recursive call with new s-o-s – definitely p is inside the circle.
    - Keep recursing until we get to zero points. One step up, a point cannot be in this lowest null-circle, it is outside this MinSC – add lowest pt. to s-o-s, recurse again with s-o-s of one point. Return to pt. 2 (one level up), with 1 in s-o-s. 2 is outside MinSC. Add 2 to s-o-s *when we entered 2, which was null*, since we throw away the lowest recursively-defined MinSC (the one with 1 in it), recurse with s-o-s including 2. Down to 0, return to 1, add 1 to s-o-s (which is now {1, 2}). Pass up again, etc, etc, etc.
- Other shapes: Axis-aligned bounding boxes (AABBs)
    - Oriented boxes are complex, axis-aligned ones are nifty.
    - Finding MinBB is easy, computing intersection is easy. Project objects along axes, see if projections overlap. If they overlap on one axis only, they may or may not overlap, must check projection on each axis.
    - Another approach is to look for actual intersections. One line of one object intersects a line of another object, or one is contained within the other (what of case where edges are touching? one directly on top of another?) Note: we do not always need the actual intersection point.
- Technique (computational geometry & graphics): signed area (measure clockwise or counter-) of a triangle. Find if two lines intersect: make two triangles with one base line and two end points of another line...if each of these two triangles have same sign, no intersection; if different sign, then the lines cross! Don't get point of intersection, but have boolean yes/no. Easy calculation: p0, p1 on one line, p2 on another line. isLeft() { (p1.x – p0.x) * (p2.y – p0.y) – (p2.x-p0.x) * (p1.y – p0.y)} If

>0 p2 is left; if <0 is right; if == 0??? Get 0 area if p2 is on original line segment, i.e. is co-incident.

<u>Collision Detection: Space Division</u>

- To do better – if we do just pairwise intersection/overlapping tests for ever pair, in each game loop, well my Jesus, that's ruff – O(n^2) tests at each iteration. Can we find objects that are more *likely* to collide? Triage! If objects just can't collide, we ignore any tests between them.
  - Grid: On 2D world, have a grid, assign objects to cells (they can overlap up to 4 if cell size > object size), and check if two objects occupy same cells. Ignore things that are not in same cells. But extra costs due to construction maintenance. Could make one giant cell, all objects are in one coarse cell, less costs; could make superfine grid, but then will basically be keeping track of every point of an object.

<u>February 15, 2007</u>

<u>Space Division (cont'd)</u>

- Hierarchical grid: use a quad tree to recursively subdivide the playing space: more depth in densely-populated places, less depth in sparsely-populated places; but how do we store info? Leaf level?
- Hgrid: looks like a quad tree; but store data (objects) not only at leaves, but in higher-level as well. As we descend, we do not always need to reach the leaves to do collision detection.

<u>Other Methods</u>

- Hierarchically subdivide by object sets (ideally non-overlapping); based on "load balancing" of objects, etc. Trivial, non-hierarchical: project objects along an axis, group based on clusters of projections.

<u>Collision Response</u>

- How do you react to a detected collision?
- Could just detect boundaries & discard movement.
- Note: we could use collisions for invisible objects -> triggers, etc.
- Physics: we want "realistic" physical responses, but this could involve many equations, too complex for game use
- Need approximation to *look* realistic.
- CR can be divided into a series of steps:
  - Prologue: Filtering/triage of collisions (which ones do we care about?)
  - Actual CR: For simple ball-hits-plane scenario – need many parameters to make even simple calculation, such as force, direction vector, angle to normal (normal can be from an actual edge, i.e. a box, or a *collision tangent*, i.e. a circle).
    - Find collision normal: something reasonable; there are lots of physical factors which affect this in reality; we will make simple assumptions (no friction, spin, etc.). Easy to find normals for circles, much harder for arbitrary polygons, so we try to reduce it to collision of a point and a line.
    - Find the inter-penetration distance and move our objects apart, i.e. where it's bouncing from: between two time-steps, two objects have gone from not colliding, to being overlapping. How do we move the objects apart? Look for smallest vector that separates the objects, least perturbation, but still might involve strange bounces (i.e. supposed to bounce off top/bottom, now bounces off sides...weird failure); Other option, base collision normal on movement vectors, just reverse object movement and find shortest separating vector. For more complex movements (with extra cost): we can do a binary search in (reverse) time until we find something that is acceptably close to the collision; how many iterations? Use

iter. # as tuning parameter for accuracy of CD/CR

  - Figure out the new vector/velocity, i.e. where it's bouncing to. Angle and force; many objects in real life are compressible, but this is waaay too complex to calculate. We assume only rigid-body response. Object has forces acting on it, what force generates the bounce? But amount of force depends on how much time we have to change an object's direction (i.e. arbitrarily small distance – need arbitrarily large force to change its direction). We use *impulses* – a force applied for a 0 length of time, i.e. instantaneous, but bounded, and don't have to worry about other forces. Have bouncing-ball situation, input vector is v-, output vector is v+, normal is nhat. F = -m*v- / delta(t), i.e. infinite value as time approaches 0. F*delta(t) should approach some bounded value J (our impulse force). What is J? Component of velocity in direction of collision normal, this should be |comp(n) v-| == |comp(norm) v+|, or v+n = - epsilon v-n, where epsilon is a factor approximating compression of objects, and 0 <= epsilon <= 1. If eps == 0, then there is no bounce, object will just slide along edge. If epsilon == 1, we get perfect reflection, very bouncy. Finding J involves a wee bit more work.

<u>March 1, 2007</u>

<u>Review</u>

- Collision response:
  - Collision normal. Inter-penetration problem – binary rewind
  - Actual response, movement vectors, what force(s) do we apply? Need to find the force that generates the right response. Coefficient of restitution – (see impulses above)
  - Need to know what force to apply to get v+n (above)

<u>Driving a collision response (i.e. bounce)</u>

- J = j * normal (where J is applied force)
- v+ = v- + J/m (to keep things massless, if we worry about mass) = v- + (j * normal)/m
- v+n = norm * v+ = norm * (v- + (j * normal)/m) = norm * v- + norm * (j * norm) / m
  v+n = norm*v- + j/m = v-n + j/m
- v+n = v-n + j/m want it to == -epsilon * v-n
- j = -(1+eps)m * v-n. Note if eps = 0, we're simply applying exact opposite force, i.e. object hits wall and "sticks" to wall. If eps = 1, we're applying -2 times incoming force, i.e. perfect reflection
- Beanbag vs. superball
- Don't have to bound it to [0..1]; doing >1 we can gain energy, etc.
- We can add other forces: friction, moment of inertia, etc.
- We are figuring out all the forces on the object; now do force accumulation/summation to determine actual movement
- For some situations, we may find objects that do not always behave as we'd expect, esp. as error accumulates. May need to add a corrective force to maintain stability.

<u>Particle Systems</u>

- Lots of particles, whose movement we control by tracking (for each particle) position, velocity, accelerations, etc.
- Used for graphical images – fire, explosions, clouds, etc...
- Can also use for physics
- Normally, we update a particle p with p' = p + vel. * delta(t) (timestep); v' = v + a*delta(t)
- Nicer, more stable way to do these updates than with objects as above. Also lets us handle constraints much easier.
- Use Verlets [from Verlet 1967] (orig. devel. for molecular dynamics). Tinker toy model: have series

of balls (particles) with links connecting them; can rotate in any direction, but must keep links constant.

- With verlets (verlet integration); speed/velocity is implicit – instead we track current and previous positions. Where p* is previous, p is current, and p' is new position = p' = 2p – p* + a*delta(t) (note that this comes from p + (p-p*) --> 2p etc).
- This is an approximation! Sometimes energy is not properly conserved...but it works out pretty damn well.
- Ex. Start at (0,0), next point is (1,0), and acceleration is (0,1). Next point is (2,1). Next point is (3, 3). Then (4,6), then (5,10), etc.
- Easy to handle constraints in this model. For instance, the above point reaches the top of "the box"; guaranteeing this constraint may be difficult in general; though individual ones are easy, as they add up the complexity grows; huge system of constraints may not have (easy) solution.
- With the verlet model, we just push particles that violate constraints in some trivial way so that the particle *doesn't* violate the constraint. Other models do things based on simulation of movement and forces. Verlets just calculate new positions for particles...we can just artificially enforce constraints (without need for backtracking, etc) and still keep it looking good.
- With multiple constraints: (naive) satisfy each constraint iteratively – for bounds, we just project the particle backwards, iterate over again. No guarantee that the iterations will converge to a satisfying state; but we can scale amount of iterations we want to do – again, works very well in practice, thought perhaps not perfectly accurate.
- Technique used in many games – Hitman (see tech paper -- Jakobson) uses verlet engine; build human models as a series of rotation points with constraints between them – ragdoll. No physical constraints to stop elbow from bending back – just add "invisible" links between joints to prevent, for instance, legs wrapping around each other, etc.
- Extends to cloth as well.

<u>Game AI</u>

- Formal/classical AI:
  - NNets, etc.
  - Don't pay much attention to complexity/efficiency of problems
  - Often a little slow (how fast does a NNet "learn" from a player? How many times must a player interact with the AI before it's "interesting"?)
  - How accurate our solutions will be? Variance in AI?
- (Combinatorial) Game AI:
  - How the state space evolves in a competitive situation (chess, etc.)
- Modern Game AI:
  - Pathfinding
  - Strategy
  - "Realism"

<u>Pathfinding</u>

- Get from A to B.
- Players drive avatars (i.e. no pathfinding needed)
- Players click somewhere, game "drives" avatar for them (i.e. Mammoth).
- NPCs have special needs, loves.
- Our goal is a route from A to B. We want shortest path, or least cost path. Avoid enemies. "Natural" path...may not necessarily be the purely optimal path.

<u>March 8, 2007</u>

Pathfinding

- need to evaluate distance
- space, distance function (usually have 5-6 properties)
- W/ obstacles and complex spaces (dynamic?), move to more of a general graph search
- Solutions for graph search (find shortest/lowest cost path):
  - Dijkstra's alg: like expanding circle, keep track of next layers of path – exponential, wastes time searching in other directions!
  - Greedy alg: take closest node to destination (through some heuristic, distance measure), make estimate as to which one is best; can get trapped in local minima. Dijkstra's gives optimal but slow, greedy gives suboptimal but fast.
  - A* alg: "combines" the above two. D's cost is distance-to-source, greedy's cost is distance-to-destination. A* does both distances as a cost. Consider the frontier nodes vs. the interior nodes, i.e. open set vs. closed set. Choose least-cost node in open set, a) is it destination? b) look at all its neighbours. If neighbour was already closed, and new cost is greater than existing distance to that node, no improvement; otherwise, found new cost to new node, add it to the open list. Our heuristic h(n) will change our expectation of optimality (i.e. closer to Dijkstra's); if h(n) is always an *under*estimate, then A* is still optimal!!! Euclidean distance is almost always an underestimate in our cases (thus, h(n) = distance from n->target). This means g(n) and h(n) must be same units in order to add them for A*, note that f(n) = g(n) + h^2(n) (in order to avoid the distance sqrt as way above), you bias the A* toward the h(n) term.
    - Other h(n) heuristic is the Manhattan distance (i.e. on grid, can only move NSEW). But note, Manhattan distance can be overestimate (i.e. if we can actually move in diagonals) – thus we do not have a guarantee of optimality! But tends to be a bit greedier, thus can be faster.
    - 8-way distance, if we can move diagonally: d = max(deltax, deltay). I.e. if deltay is greatest, then we can move diagonally AND up so we always move toward target. Can determine the diagonal part of the path vs. the straight part and weight them differently. Closer between Manhattan and Euclidean.
    - Beam search: in general, open sets can be large. Solution – bound the open set! When we reach a bound, we discard nodes that have highest f(n) in our open set. In theory, have a "searchlight" pattern toward destination; but not always complete, might accidentally discard potentially optimal points.
    - IDA*: iteratively deepening (for saving memory). Set a bound on f(n), if we don't reach destination, increase f(n) bound, repeat. How much do we increase f(n)? Not the best in games because you keep repeating searches.
  - Bidirectional: search src->dest *and* dest->src. Instead of expanding way too exponentially in one single direction, we search a lot less space going from both directions and (hopefully) meeting in the middle. Nice idea, but often doesn't work [Pohl 1969]. More complex versions exist, but lots of overhead.

Hierarchical Pathfinding

- PF is expensive; can we use coarse-grained PF + fine-grained tuning and get reasonable paths, for much cheaper?
- Simple sub-sampling. On a coarse grid, find high-level path; at each finer grid, have to find more smaller-ish paths. But we need to track connectivity between coarse-grained cells.
- See handout [Botea et al.] for hierarchical PF. Coarse & fine grid. For each adjacent pair of coarse nodes, need to find all entrances between them, identify specific places in entrances as transition destinations between coarse-grains; for large entrances, can add many nodes to make things more "natural" (i.e. don't have to pass through only one point in entrance). Then do fine-grained A* search between *every pair* of entrances in one square (thus why you don't want too many entrances), pre-computation. Then choose start/end points, then connect these points to all possible

close-by transition paths, this then forms a search graph – much denser than it would have been if we used only the fine-grain grid. Not necessarily optimal. Can do path refinement as a post-processing step. Iteratively flatten out "side-excursions" in path, can even refine dynamically!

Natural Movement
- Add some randomness
- A bit like the opposite of path refinement.
- Content generation!

March 13, 2007

Hierarchical Pathfinding (cont'd)
- Botea et. al worked with Bioware
- Used HP on Baldur's Gate; tried on 120 maps, 50x50->320x320
- HP: 10ms for searching possible paths.
- path length vs. #nodes or CPU time (cost): previous algs. are roughly linear; HP is fairly logarithmic
- NOTE: Also measured error! How close to optimal path? (Though we don't always want exactly optimal, of course). Within 0-1% of optimal paths in terms of shortest distance.

Potential Fields
- See Robotics notes
- Assume start pt. is "high potential" and destination is "low potential," follow the gradient. Obviously can get stuck in traps.
- To handle obstacles, we can handle repulsive forces.
- Best for pre-processing, for static environments; not so great for real-time, dynamic envs.
- But in practice, doesn't work that well – local minima! Even if it's not a concave obstacle – could be two obstacles with a path between them, but they each give repulsive force to object and could eventually slow it to a stop.
- Could add randomness to get out of local minima (and break ties) but how much? Too much, character "stumbles around drunkenly"; often the magnitude we need to ensure we don't get stuck results in very random movements.

Group Behaviour
- Baldur's Gate syndrome
- 5 characters need to go through narrow hall
- All do PF individually -> One goes into hall, the rest suddenly have an obstacle in the way and walk aaall the way around the map.
- Several practical solutions:
  - Plan ignoring dynamic obstacles and only deal with them when they are encountered. But if speeds are not the same, we may catch up with each other. When we encounter dynamic obstacles, we can simply wait a bit, and hope it becomes no longer an issue.
  - Trading/pushing: if we have B->A, flip positions so we have A B->. Pushing; but if walking down narrow hallway, may never be able to get to end (A always in the way)

Flocking [Craig Reynolds 1987]
- Awesmo!
- Motivation: flocks have group behaviour, birds/fish somehow stay together, yet can avoid obstacles and move toward goals.
- Model of "boids" (http://www.red3d.com/cwr/boids/). 3D movement model, geometric flight model.

- Different accelerative forces around boid (braking, steering, acceleration, yaw, pitch, roll – some may be stronger than others).
- Flocks have many of these together, natch.
- Rules (We can convert these rules into specific forces!):
  - No collisions with other flock members (or anything else): Move away from potential collisions
  - Stay with the flock: move toward the centroid of the flock
  - Keep up with the flock: Look at flockmates in the neighbourhood, and align our direction vector toward theirs.
- Could consider all flockmates (Reynolds uses inverse square method to measure influence from one member onto another), or just local neighbourhood (which is probably more efficient, and possibly even more natural in terms of what real flocks do).
- We get 3 new forces from above rules. Weighted average; often different over time and situation (i.e. if about to hit an obstacle, obviously try to avoid it is the main priority, flock coherence is less important).

<u>Steering Behaviours [Reynolds 2001]</u>
- Generalizes the idea of the rules given above
- Primitive rules that we can use and combine to produce complex, "emergent" behaviours

<u>March 20, 2007</u>

<u>Review</u>
- Pathfinding --> A*
- Steering behaviours, group movement
  - Force model (accel, turn left/right, braking), simple local rules for each individual "boid" (see three rules above)
  - We can extend these rules to various primitive behaviours (not just birds, and fish, not just 3 rules)

<u>Other steering rules</u>
- Seek: Moving in a given direction, goal is not along mvmt vector; can add corrective vector to our own direction. The corrective vector will be simply v_c = v_desired – v_mvmt; this calculation is repeated over time. Get a nice arc steering toward final location.
- Flee: Opposite of seek, oddly enough. v_desired is inverted to instead point away from "goal" or "danger."
- But nothing stops you when you're "at" the goal – tend to get "rubber band" effect as you keep overshooting the goal. Problem: not slowing down as approaching target.
- Arrival: Basic seek behaviour, but have radius around target; inside the radius, objects slow down. Must slow down increasingly as approach target, but must reach the target! Have a slow-down function which reduces speed to 0 only at target. Linear/non-linear models – different models for pedestrians and cars, eg. (i.e. humans accel. faster than cars, but cars go faster).
- With dynamic goals – adjust path frequently...how frequently? Depends on cost of calculation.
- With dynamic goal, only extra complication is to aim where target is going in the future. Can make linear/non-linear projections of past/present into future, or project using our own steering behaviours.
- Obstacle avoidance: flee? But makes you move maximally away from an obstacle – i.e. running completely away from a static wall. Want to move along, only worry about obstacles that we may collide with. Imagine boid moving through obstacles – could imagine CD/CR. But want a much simpler version...we surround everything with bounding circles. Then make swept volume of our moving character. Can do simple (hierarchical, eg) CD to check circles for collision with swept

volume. Could look at all collisions, but as with other steering, we do things locally – look at closest intersection (as heuristically most important collision). Use vector from circle centre to swept volume (actually the overlap of the vector and the volume) as the minimum corrective force on vehicle's movement.
- Symmetry issues: flock approaching a wall, two boids randomly decide to avoid wall by moving toward each other, cause potential second collision (and some poor boid could be caught in the middle). In combining forces, we need to find the right weighting of forces, and the right order of calculation.
- Blending: Averaging/summing forces every iteration tends to work poorly [Reynolds] – sometimes (as above), averages sum to zero and things just stop, which is weird.
- Solutions: apply each force on a boid sequentially, which gives chance to do a little bit of movement and get out of symmetric situations. Prioritizing over time (CD gets prioritized with proximity to obstacle (perhaps static over flockmates as well)). Prioritized "dithering," we weight steering forces and then choose a force to apply by random value and weights.

<u>Future Behaviour</u>
- Pursuit and evade require some model of future behaviour of opponent.
- We nearly always need to predict opponent behaviour.
- This has been explored in more than theoretical environments.
- John Laird's QuakeBot, "Soar"
- Goal-based: choose high-level goals, select appropriate low-level operators to achieve these.
- Cycle:
  - Sense the environment: mimic a player or more realistic character, what can it see/hear/etc. LOS (line of sight), FOV (field of view) contains many LOSs. What is a straight line (for LOS)? How can we calculate FOV? Dijkstra's? Can always cheat. The game knows where you are, often make use of internal info not available to player – through try to hide it ideally (i.e. don't implement LOS, but have animation where opponent occasionally turns around and choose 3% of time to "spot" player behind them...looks plausible). In Soar, some sense data is used.
  - Plan goals: move to power-up; attack player; search/wander; etc.
  - Implement them using primitives. eg. Collect power-up -> get item -> goto item -> (face item, move toward it, stop when there, notice if missing)
- Primitives here may not have been optimal
- But! Key idea developed: *anticipation*. Soar gets the ability to project itself onto the opponent, "what will the opponent do? what would I do?"

<u>**March 22, 2007**</u>

<u>Recap</u>
- Game AI != academic AI
- Game AI sometimes cheats (more resources, more info)
- Want a challenging AI, but (ultimately) weaker than player (player needs to believe it's winnable)
- In Soar and other AI, we get interesting behaviour – anticipation, predicting opponents' moves.

<u>Content (Pedestrian) Generation</u>
- All the things in the game. Whoa.
- Largely visual "set dressing," and simulation
- Focus on pedestrian generation – "set dressing" of urban environments
- Urban envs. are fairly common.
- Constraints – want "realism," but at what cost?

- [Feurty '02] Statistics
  - Regardless of age/sex, people seem to do ~6 minutes (depending on context), generally less than 5km, ~100% are <0.4km
  - Weather: shopping, ped. movement drops 60%, in residential area, drops 40%
  - Patterns of activity: morning – go to work; evening – go home (surprise). Not symmetric, morning has smaller rush hour than evening (i.e. people "drift back" in the afternoon). Mid-morning, get business trips, deliveries. Mid-day, lunch traffic.
- Individual pedestrians (culturally specific)
  - Women and men have different comfort zones; tends to be larger for men. Women have 0.09m^2, for men it's 0.14m^2 (increases toward front when travelling fast, for instance).
  - Level of intimacy – gradations of intimate (<=0.5m2), personal (0.5-1.2m2), social (1.2-3.7m2), public (> 3.7m2)
  - Speed also important – matching speed is a kind of communication. Makes sense for intimate, but doesn't make sense for complete strangers (unless you want to model stalking).
  - Ped. speed: 44m/min <-> 122 m/min (gait changes outside these limits: shuffling vs. running)
  - These are all local measurements! (Why bother caring about matching speed with someone 1km away?)

<u>How do people move?</u>
- Observe each other
- Basic behaviour
  - Communication via vectors – project into the future, do prediction, alter our vector appropriately; sometimes both folks alter, sometimes one-sided (biased according to ped. aggression).
- Grouping pedestrians.
  - Checkerboards – people maximize room around them, leave space in front and on either side. Groups spontaneously form these structures very often (from basic principles). Generally form when closer than 1.5m from person in front; dissolve around 4.5m.
- Individuals (again)
  - Goal-driven behaviour; often divided up locally into simpler goal (e.g. straight lines). But these "straight lines" are often not straight, tend to "wiggle" a bit. The zig-zag may have varying regularity, amplitude, frequency, etc.
  - Implies something for passing other people.
- Semi-groups
  - < 50% of pedestrians are walking alone. Big groups are hard to maintain – dominance of pairs, a few triples, much fewer quads, etc. Larger groups end up hierarchically split.
  - Matching speeds, try to line up.
  - Group members stick together. People will actually wait for each other.

<u>Pedestrian Simulation [Loscos, Marchal, Meyer 2003]</u>
- Aim at large scale pedestrian environments.
- Incorporated goals, local information, semi-groups.
- Discretize the space (i.e. a grid) with buildings on it. They take building structure, then construct area where peds. would go. Around each building is a sidewalk. Then establish goals, drop "goal point" bread crumbs, esp. around corners (though they could be anywhere – can stop and look into a window) – lots of effort to heuristically find the corners (why? Could just use geometry of input). Coarsen the graph – merging adjacent/close nodes (edges between nodes indicate visibility/reachability). Edges going across roads are crossing points. Now we can simulate pedestrians.
- CD b/w pedestrians
  - Template-based scheme: frontal (deviate frontal vector within a bound to avoid the collision),

following (leader deviates, or follower deviates), or perpendicular (the "slower" collidee is the one who deviates).

<u>March 27, 2007</u>

<u>Last Time</u>

- Content
  - Content generation == doing things algorithmically, procedurally
  - Manual generation is expensive
- Pedestrian simulation
  - Based on statistics, we have simple models
  - Collisions -> tend to be avoided (rather than detecting and responding).

<u>Pedestrian "flow"</u>

- Not as dogmatic about "keeping to the right" as on the road
- Pedestrians travel as groups (often), and often form "platoons", i.e. cluster – space – cluster – space, (....) ---- (....) ---- (....) ---- etc., driven by traffic lights, desire to stay in groups, etc. But pair/triple travel slightly different than cars.
- Other patterns – following. Do whatever person in front of you does. "Ant-like" behaviour – movements leave behind pheromone trails. They get detected/reinforced, etc., scent decays over time, and can be given different priorities. This sort of behaviour used to control groups; leader lays down scent trail and followers follow it. But if scent decays (too) quickly, followers far back get lost – so let initial followers drop a little pheromone of their own.
- OMG JOSH LEFT CLASS JUST NOW HE'S A JERKFACE
- Groupings – have leaders and followers, but how to represent as pedestrians? Would get a leader separated from followers: [(....)   .] -- can have a "phantom leader" that isn't rendered, so we only actually see the group of followers.

<u>Game Balance</u>

- All players should have an equal/even chance of winning
- Could dynamically adapt balance? Can we even measure it to begin with?
- Could just try to statically analyze balance from the start.
- One way: make everything symmetric between players (e.g. all possible guilds to choose from are exactly the same stats-wise, if with different name). Balanced, and quite boring.
- Dynamic balance measure: *dominance* [Taki, Hasegawa 2000]. Looked at soccer – what is a crushing defeat/victory, what is a back-and-forth game? Can we find out who is *winning* as opposed to who *won*, for instance for "intelligent announcer" (the latter falls under content generation).
  - Algorithm for soccer: dominant region analysis. A player dominates the area around them in which they can get to the ball before anyone else – the larger the dominant region, the better the player controls the game (i.e. can get the ball and use it). A team with the most amount of dominated space is controlling the game.
  - Need to build a kind of "generalized Voronoi diagram". We can use larger structures than single points, that is objects, to make a GVD. Taki et al. look at movement – players are like dynamic particles moving around, have a dynamic diagram which is skewed by the properties of the players' movement (remember Reynolds' Boids movement paradigm). Ability to accel/decelerate affects what you can reach before anyone else -- "reachable" area is ellipsoid projected forward in direction of motion, i.e. at high velocity, can reach far in front, very short in back.
  - Instead of polygonal Voronoi diagram you're used to, get something totally messed up. If A is moving and B is stationary, B can often get to points "behind" A quicker than A can turn around

  and get there; similarly, A can get to some points beyond B since A is already moving.
  - ***IF*** we can calculate all these areas, then we get a sum of "controlled" area by either team. Side with the larger area is winning, or at least in better current position (winning not always just a "greedy" algorithm).
  - Taki et al. did analysis. Took cameras watching actual soccer games and overlaid/calculated dominant regions for players. Plotting area vs. time (i.e. >50% area == winning), we see noisy data increasing above 50% for winning team – neat.
- Static balance:
  - Use a lot of alpha, beta testing – is there a strategy for consistently winning, want to avoid trivial/easy-win strategies.
  - Payoff matrices. Use game theory from economics (nice, but doesn't scale very well). [Olsen (from Microsoft) 2003] – table of P1 vs. P2, and all the actions either one could do (thus why it doesn't scale well – would be nxn for n players, row/column for each action, each cell has result for each player). Cell (c1i, c2j) in the table implies P1 does ci, P2 does cj – table contains payoff/punishment for this combination of actions. In P1-P2 game with cells:

	P2 – c1	P2 – c2
P1 – c1	(P1 = 0, P2 = 0)	(-1, 1)
P2 – c2	(1, -1)	(0, 0)

  - This is a zero-sum game, not too interesting, but balanced.
  - More complex example:

	P2 = attack	block	rest
P1 = attack	P1: -10, P2: -10	0, -1	0, -20
block	-1, 0	-1, -1	-1, 1
rest	-20, 0	1, -1	1, 1

  - Benefit to resting (boosts health, say) high with no attack; very low with attack.
  - Can look for certain greedy situations – run millions of simulations to see if our player's success rate is ~50%; if not, it's unbalanced.
  - Can look for Nash Equilibria – find a local optimal action for an individual player, i.e. all other direct options are worse. Easy: P1 attacks – but then if P2 attacks, it's better to block, same with resting – thus P1 always blocks if P2 attacks. If P1 is resting, P2 decides to rest, since that gives P2 a benefit. Create arrows for each player; as Pi, assume Pj does action k; what is best corresponding action that Pi should do? If we find a cell with all arrows leading in, and none leading out, that is local optimum.
  - In our table above, the equilibrium is that both players rest. BOOOORING.
  - Variation: in cells above, showed player affect, how choice "affects me". For instance, take action with maximum differential between payoffs – instead of (-20, 0) -> 0 to me, do (-20,0) -> 20 to me.

<u>March 29, 2007</u>

<u>Review: Game balance</u>
- Dominant regions
- Payoff matrices – look for local minima, Nash equilibria corresponding to "stable" situations. How stable they are depends on how willing the player/NPC is to suffer worse behaviour/results to achieve better ones.
  - Usually have large matrices, and imprecise/dynamic/multiple goals.
  - Offline analysis can be used to do very simplistic/statistical studies – over millions of scenarios,

does one player win more than another?
- Looking for balance: “different but equal”

Storyline Content
- PNFG, Inform, etc. Template-based idea?
- Done trivially, we can just change names and use stock templates.
- In RPGs -- “FedEx quest”: P1 gives you X, you must bring X to P2.
- Templates really should be well-customized to disguise the repetition.
- As a *tool* for artists/designers on which to improve, templates are fine.
- *Emergent* – based upon actions/desires of NPCs, simple rules for each individual become complex interations. This is the ideal, but in practice end up with very noisy storylines; doesn't always produce larger-scale story structures that form the game.
- Hybrid approach: large-scale structure/story is predefined, smaller-scale story elements are formed dynamically.

Sitcom Generator [Charles, Mead, Cavazza 2002]
- Chose “Friends” as a model, might as well start at the bottom of the barrel
- Used the Unreal engine
- Give high-level goals, reduce to a selection of lower-level goals.
- HTN (Hierarchical Task Network) has AND and OR nodes. NPCs given various options/choices to achieve goals, this can hopefully lead to interesting interactions.
- Many factors are involved in these choices
  - Failure scenarios
  - Physical position (blocking)
  - Randomized events (elevator stops working, etc.)
  - User interactions.
- See the handout with the HTN
- First problem: Rachel is using the phone. Failure scenario, give up on current goal, re-plan with backtracking search, exclude the failed plan (i.e. “Talk to Rachel”). Note that the environment is dynamic, and she might get off the phone in a bit. So instead of completely excluding it, we just degrade its weight.
- Resulting narrative does kind of work: general structure with interesting smaller-scale variations.
- This could be a tool used for story generation.
- Downsides: only in prototype stage? HTN can get HUUUUGE, generated by humans means its difficult to produce/verify/test.

Scripting
- Originates in testing.
- Write some portion of game (character) behaviour in a simple language – game engine interprets/executes it.
- Have bots separated from the bots/scripts (instead of integrating both of them together). I.e. sandbox approach.
- (Properties of) Languages:
  - Simple syntax (for developers)
  - Expose the scripting language? (For consumers)
  - For consumers to use it, must be *not* like a programming language.
  - Developers want complex API, such as procedural language syntax (ideal?).
  - Game-specific components. Have very high-level calls that makes it simpler to consumers, and gives access to game-specific behaviours.

<u>April 3, 2007</u>

<u>Last time</u>

- Content:
  - Story generation: emergent, not pre-planned
  - Becomes important as game gets larger
  - Scripting

<u>Scripting</u>

  - Automatic players, a more structured form of AI
  - See HTN above
  - Can also help testing
  - General languages: Ruby, Python, Lisp, etc.
  - More complex languages: C/C++-like
  - Choice of whether we give high-level, game-specific primitives; or give primitive, low-level script elements.
  - Do we put more power/responsibility in the scripting language or in the game engine?
  - Other languages: Forth, Lua, other game-specific

<u>Neverwinter Nights</u>

- Script-driven RPG
- Generate/describe content; certain things (this key required for this door) happen very often, but coding the specific instances is repetitive and complex – requires "magic names/numbers", often just cut-and-paste. It could help to create a primitive to represent this.
- Idea [McNaughton et al. 2004]
  - Give "standard" templates for common actions.
  - Can adapt template to the situation, and specialize specific objects, *and have a GUI.*
  - Then does code generation automatically, to generate script code.
  - See handout – NWScript (short side) is more C-like, very programmer-oriented; ScriptEase (long side) more high-level.
  - Did experiment: gave NWScript to a familiar user (developer) and gave them a scenario to implement – took ~3 days; gave ScriptEase to a high school student to implement the same scenario – had one week of training with ScriptEase, then took one day, with really only 3 hours of main script, and the rest for making up dialogue
  - McGill summer camp – gave students ScriptEase + NWN (Aurora toolkit)

<u>Multiplayer Games</u>

- Single-player games are numerous
- The word "Multi-player" sells games – whether this works in practice or not
- Different kinds of MP
  - "Hot-seat" games (original MP) – two players are in the same room (think of chess). Not good for games where players don't have perfect information (i.e. Poker)
  - Networking!
  - <= 4 players is a "small" MP game, network problems are usually not severe
  - <= 16-32 players, medium-sized, but with some techniques game can run well without severe major network problems
  - +100s of players, large-scale MP game, limit before we need major network solutions
  - +10,000s of players, Massively MP, get into problems of scale, bottlenecks.

<u>Network Topologies</u>

- Centralized, client-server: S at centre, Ci clients communicate solely with S. Clients are symmetric, communicate with server (messages, broadcast?).

  - Advantages: Single sign-on (allowing choke-point where we can allow/deny entry, make revenue); very simple model (all clients send data to server, server processes, sends back updates)
  - Disadvantage: Choke-point at the server (bandwidth, CPU – depending on tradeoff between client/server usage)
- Clustered server: Servers S1...Sn, each have an equal distribution of clients.
  - Advantages: Reduces network/CPU requirements.
  - Disadvantages: Must enforce inter-server consistency.
- Shards: Servers divide up world disjointly.
- Peer-to-peer: Duh. E.g. Everyone broadcasts everything to everyone else. Could be other forms: Hierarchical, "supernodes" (nice continuum b/w C/S, clustered servers, hybrids, and P2P)
  - Advantages: No single point of failure; fewer network bottlenecks; *scales* well.
  - Disadvantages: How do you make money off of this? Owning company has less control over the game. Friggin complex. Consistency again complex. Cheating hard to enforce/filter/analyze/respond. With C/S, can filter all the packets and look at them for cheating madness.

Network Protocols

- Re: Assign 4
- Many layers: TCP, UDP are on top of IP
- UDP: datagram (up to 64KB) – unsure delivery, not always in order. But very quick.
- TCP: connection-based, stream-oriented. Order *is* guaranteed, as is delivery. May *buffer*, may need to disable the buffer. Slower (re-sending, doing ACKs, etc.).

April 5, 2007

Scalability

- Can we maintain network connection at a good rate for millions of ppl potentially across the world; i.e. amount of data sent/received is manageable

Simple calculations

- Client/server – FPS rate; in a game loop, we send and receive between C/S. How many clients can a server handle? If we want 100fps (send 1K, receive 10K), so we get 100x(1K + 10K) per client.

TCP vs. UDP

- UDP preferred in games, as it is faster; best-effort reliability works out decently
- UDP can be lossy (info could be dropped along the way – load shedding one example – or might get there late).
- See situation in assignment 4, two clients updating their positions from server. What happens when they lose contact?
  - One solution: Stop completely, request and wait on missing info. But in games/multimedia this is no good, as we interrupt the flow (care about recent network info as opposed to old info). Called the *milk* strategy as opposed to the *wine* one (milk best when fresh; wine is better with age).
  - Packet compression/aggregation: reduces network requirements, but doesn't deal with completely lost packets.
  - Most recent info: do dead reckoning.

Dead Reckoning

- Based on past information, we guess at the current state. Last position – accurate if object isn't moving, or isn't moving much (i.e. wandering in small area). Look at speed, acceleration, higher-order, etc. Problems: Imagine A, B, and C. A shoots at B, but shot is slow in network...B still moving, shoots C. Disagreement as to who should be dead, and who isn't. B? C? Both? Neither?
- Flying Tank problem: reckoning places tank further forward, when in fact it turned. Eventually we get proper update from the turned tank, so we simply move the tank to it's proper place. But if real turning tank should have hit a mine, and we teleport it over the mine, then the explosion that should happen does not.

Consistency
- Consistency: 2 sites/nodes, x and y, that have received all events up to time t are considered consistent if they are in the same state. (Note that it does not talk about order of events).

Correctness
- Assume a perfect site P exists that receives all messages without delay. Relationship of a site to P gives correctness.

Example
- At time t2, X sends message a (which P gets immediately), Y receives a at t5. At t3, Y sends b (P gets immediately), X receives at t6. All consistent before t2. X and Y become inconsistent at t2. At time >= t6 they become consistent again (both know a,b).
- X is correct (relative to P) between t2 and t3. Y becomes correct from t5 onward, X from t6 onward.
- *Correctness implies consistency!* (But not other way)
- For X between t2 and t6 (sent a, but hasn't received b) – called a short-term inconsistency.

Response Time
- How fast does a site respond to a message?
- Fast, "eager" clients actually lead to more inconsistencies! In X, Y, Z, if Y sends a message, only Y knows about it at first. If Z is very quick (aimbots? Etc.), and sends its response right away. More inconsistency to other players (not itself).
- What if we "lived in the past" a bit – everything we do will be at a delayed time/rate. Game runs a bit in the future, delays everything so that they hide the short-term inconsistencies – messages sent, only acted upon in a few milliseconds.
- *Local lag*
- More standard distributed simulation, worry about correct communication. Do rollback when you get old missed data. Time warp – simply go back in time to where we get older packet, and continue simulation from there.

Interest Management
- Packet aggregation was a technique for reducing network data
- With IM, only send relevant information to/from clients (i.e. don't send anything between x and y if they are on opposite sides of the map).
- Simple way: give each character an "aura" which defines the area in which a character can perceive. Look for aura intersections, decide to/from whom we send information.
- Break down further into:
  - Focus: what I can see/perceive.
  - Nimbus: area in which you can be perceived/your perceptivity. Check if x.focus overlaps with y.nimbus.

Cheating
- Major concern (mostly in multiplayer)

- In MP games, lots of cheaters (any perceptible amount) turns into *lost revenue*.
- Lots of variation
  - Suppress correct cheat: based on dead-reckoning. Cheater doesn't send info, while they are out of contact they can see game going on; when they are about to timeout/be kicked out, send any form of “ideal” info (i.e. warp right next to someone with a gun to their head).

April 10, 2007

Last time
- Multiplayer issues, from small to large scales
- Topologies: client-server (nice amount of control for company), P2P
- Bandwidth/latency concerns: note the diff. between MBps (bytes) and Mbps (bits)
- TCP vs. UDP
- Dealing with latency/loss – for instance with C/S, with one slow/lossy client that can either slow down the whole system, or “becomes left behind”, loses info, and we lose info.
- Dead reckoning techniques – a good guess, but may be incorrect.
  - Can lead to Dead Man Shooting or Flying Tank problems, due to fact that we do not have all intermediate steps.
  - Can we “replay” the actual movement to check for real sequence of events? Can request missing information, but gets expensive obviously. Furthermore, can have cascade effect – I thought player X was somewhere so I shot at him there, but when I get the update, they never were there in the first place. Very strange inconsistency...instead, try bringing *everyone* back in time to replay the scene.

Consistency & Correctness
- See definitions above
- Can “measure” inconsistency, as well as find where DR may be used.
- Noticed that a fast response time can lead to many inconsistencies among the *other* players.
- We delay our responses – each message is actually “scheduled” just a little bit in the future. If scheduling delay is longer than network communication delay (tough thing to do), we can keep things very consistent (though have to balance local delay per player, with consistency).

Interest Management
- Similar problem as encountered in CD...do we really want to check every object for collision with every other object? Not if they're nowhere near each other.
- See Aura(Nimbus/Focus) above – intersection implies interest.

Cheating
- Common in MP environments. Problem because people pay per month – cheaters piss the real players off, who then stop playing. No cheating -> No lost revenue.
- Affects SP games as well.
  - Casino games – if real $$$ involved.
  - “Enhancing the game experience”, or for comparative reasons between players.

Cheating Types
- [Pritchard 2000].
- 1. Reflex augmentation cheats: monitoring I/O operations, mouse macros, etc.; use computer's response time to improve our own. *Aimbots* – have for instance a filtering process running in the background which instantaneously sends aiming information back to server. How to detect? Hazy area – humans can be very good, and aimbots can throw in random “misses”.

- 2. Authoritative Client: if client is considered the authority on an object such as their player (often done to push processing load away from server), we can give them too much authority. They can then lie to their own benefit or the detriment of others.
- 3. Information exposure: Client gains a lot more information than the player should really have (e.g., has info on a player on opposite side of wall). OMFG WALLHAXX!!1!
- 4. Compromised servers: server cracked. “Inside” information gets exposed.
- 5. Bugs & Design loopholes: is this cheating or just exploits? Lots of bugs initially (shortcuts, unbalancing the game)
- 6. Environmental factors: calling ppl up during poker to discuss cards.
- Other categories of cheats [Yan et al. 2005]
  - Categories A-O (~15)
  - Collusion, abusing game procedures (escaping from the game temporarily), AI cheats, etc.
  - Categorization – know what to defend against

<u>Dealing with cheating</u>

- Baughman et al. 2005
- Cheat prevention
- Assume bucket synchronization as an algorithm. Messages go into buckets/frames.
- Two kinds of cheating:
  - suppress-correct cheat: if we don't have all info about all clients for a given bucket, we use DR. Lots of missed info implies disconnection. Server has threshold n to wait for players – cheater waits n-1 time units to collect information and then send the optimal strategy. Solved with stop-and-wait. No DR, wait for everyone...but game occasionally stalls.

IT'S OVER!!1!