Virtual Worlds

Lessons from the Bleeding Edge of Multiplayer Gaming

Software Design and Production

Greg Corson

Dave McCoy

Key Lesson

The most used part of our whole design was the connection library.

It was also one of the biggest consumers of CPU (since we used it so much).

The Connection Library

- Was a collection of smart pointer, iterator, search and list management routines.
- This supported links between objects and various types of lists (tree, linked, linear, indexed)
- When an object was deleted, it automatically removed itself from all lists.
- Lists were used everywhere
  - Event queues and publish/subscribe links
  - Tracking components that made up a game object
  - Scheduling execution order of various components
  - Rendering functions, like tracking which lights effected which objects.
  - Managing network connections and player lists

Why did we use it so much?

- We found that even when built on the fly every frame, lists were more efficient than other methods (like using skip flags).
- The automatic deletion function made dangling pointers unlikely and eliminated the need for code to explicitly deal with connections between objects when one was deleted.
- A single list/connection management library isn’t optimal for everything, but it does allow it to be highly optimized.
- Virtually every object in our system descended from a connection manager class.

List Usage Example

- The game has a list of all the objects in the world.
- Each renderer keeps a list of just objects that it is “interested” in.
- Each frame, it iterates a culling process over this list and creates a new list of just objects in view.
- The renderer iterates over this new list, and runs the list of rendering routines attached to each object with publish-subscribe links.
- With each pass, the lists get smaller and more cache efficient. Since each pass does a simpler function, the code is more likely to be in cache.

Key Lesson

Build your own memory allocators to aid in debugging and reduce waste of RAM

Memory Allocators

- Each major class had it’s own memory allocator and a private heap.
- Heaps were monitored and peaks recorded so the starting heap allocation for each structure could be easily pre-set.
- Heaps were allocated in memory-manager and cache friendly block sizes.
- A heap could still handle objects of different sizes, and often did but we tried to use separate heaps for unrelated data like rendering and simulation.

Advantages of this

- Since all the elements in any heap were the same size, fragmentation and waste was small.
- Few OS memory allocation calls (after initialization) made memory allocation very fast.
- Related data was kept together in RAM, improving cache efficiency and reduced chance of virtual memory thrashing
- Data specific patterns could be written into allocated RAM to aid debugging.
- Signatures could be added to allocated objects to detect data corruption.

Key Lesson

The best way to insure reliability and speed development of a complex system is breaking it up into loosely connected and completely separate black boxes that can be individually tested.

VWE Software Design

- The system was broken into functional blocks to make coding and design easier.
  - Housekeeping/OS
    - Scheduling, frame rate maintenance, event and message dispatching
  - Simulation Block
    - Motion, Collision, Animation and each vehicle subsystem
  - Interest Management
  - Renderers (sound, video, gauges)
  - Network

VWE Software Design

- Code functions and data weren't allowed to cross boundaries. No rendering code or data in simulation section…etc.
- Functional blocks communicated using queued messages or "publish and subscribe"
- Blocks had few dependencies on each other
- Every block had extensive built in self test routines.
- Every block had optional runtime consistency checking routines.

Advantages of this approach

- Most blocks could function in isolation allowing their test code to work by generating test patterns on the inputs and outputs.
- Major functions (network, simulation, any renderer) could be turned off without effecting the rest of the app. The system would run just as well with no renderers running as it would with three.
- Bugs in one block rarely effected others
- It was very easy to divide up work because major systems weren't interdependent.

Other Advantages

- This approach made the software much more robust and easier to change.
- Because connections between modules were limited, new versions of one module could be reliably tested with old versions of another.
- Our workload was more focused on optimization and debugging network related multiple machine problems as opposed to traditional “bugs”.
- Cache coherency was much higher because related functions were always done in batches.

IE: Rendering by Observation

- Simulation runs separate from renders and publishes data renders need (position, speed, guns firing, lights on, smoking…etc)
- Renderer gets a message when a renderable object is created and runs an script that sets up data structures to render the object.
- Each frame it “observes” the state of the objects and renders what it “sees”
- Some simulations pass through several states in one frame. “State watchers” were built to catch specific transitions for the renderer.

Rendering by Observation

- This made it easy to do multi-screen rendering, since you could just add another instance of the renderer for each screen.
- Because it was script controlled, an artist could easily change placeholder graphics to test or final ones without programmer help.
- Many effects like sparks, smoke or rotating radar dishes could be handled entirely in the renderer.
- Simple effects, like brake lights, could be added by artists without help.

Key Lesson

Network communications and object ownership need to be carefully managed for things to be efficient. A good design up front can make the system much more flexible too.

Network Object Model

- Objects could send/receive messages locally or on the net. Though not everything was net visible.
- Every object was “owned” by one CPU
- Remote CPUs kept “replicants” of objects they didn’t own, which contained only public data.
- Whether you had an original or replicant object, you always interacted with the local version which handled all the communication with the owner.
- Only the original object was allowed to send messages updating the state of it’s replicants.
- In effect, every original object was a server, with it’s replicants as clients.

Messaging Model

- Messages were stateless and never required waiting for a reply
- Messages could be sent locally or on the net, and could be queued with a time delay
- No data locking mechanism was needed.
- The structure of messages were inherited using a C++ class like mechanism.
- Incoming messages were automatically routed to the right class method
- Most messages were broadcast to everyone, but the design allowed them to be filtered.
- Messages were designed to be order independent.

Example

- Damage Transaction
  - Shooter determines hit location, sends “I shot you here with this weapon” to the target and may also create a hit effect.
  - Target determines damage to armor/systems
  - Target sends any changes to it’s replicants such as appearance of damage, parts removed…etc.
  - Target creates secondary explosions and other effects
  - Target sends scoring information to shooter
- Unfortunately, this is harder to make cheat-proof.
- If two people both deliver enough damage to an object to destroy it whoever’s “I shot you” packet gets there first will get the credit for it.

Routing System

- The VWE system was designed to be peer to peer
- Messages within a site were usually broadcast.
- Each site was treated like an “island”
- A machine was designated the “router” and would send messages to a partner at a remote site which would rebroadcast them there.
- This could also be done individually by owner objects to spread the load around.
- A similar system could be used to link several LAN parties together today.

Ownership Transfer

- Ownership of non player objects was distributed at startup but we had a way to transfer ownership.
  - Old owner sends ownership transfer message to new one, along with all private data.
  - New owner creates a full "owner" object from data.
  - New owner tells all replicants of ownership change
  - Old owner transitions to being a replicant, which will forward messages to new owner so nothing is lost.
- Transfer is stateless, no locks needed.
- Unexpected disconnects can be handled by replicating private data on other machines and using a timeout mechanism to force ownership changes if someone disconnects.

General Thoughts

- Ownership transfer can be used to reduce latency. Transfer the ownership of an object to whoever is currently interacting with it the most.
- It greatly simplifies network design if you use an error correcting protocol for everything.
- TCP/IP isn't perfect, but is supported in the net so error correction happens much closer to the user.
- Careful use of TCP/IP settings (nagel, force) can greatly improve performance.
- It isn't as easy to create your own error correcting protocol as you think. Study carefully before you try to roll your own.

Key Lesson

Not everything has to be updated over the network.

"Because people can only see their own screen, the consistency of that screen is all you have to worry about."

Interest Management

- For large numbers of players or lots of dynamic effects, this is essential.
- You need to determine which objects you (and other players) should be "interested in" so you don't send/receive updates that aren't needed.
- Techniques for doing this vary greatly with game design. In our case we used mainly range and time duration (to cull brief special effects).
- If your game takes place indoors or in a more dense urban environment, you can do something more complex.
- The military sim people have done a lot with this.

Network Updating

- Many special effects can be created with a single message, run to completion and self delete without ever sending another.
- It's ok for particle effects like fire to be slightly different on every computer, just send a "start fire" and make sure something similar is generated on each machine.
- Laser beams will become disconnected from the gun or target if you send them as normal objects. Send which gun fired and the point hit on the target, then adjust the local gun angle so that, while it's on, the beam is hitting the right spot.

More Network Stuff

- Be careful with "area of effect" weapons like explosives. Blowing up an ammo dump can create n squared packet storms that will bring down the system.
- Use "encores" to gather a frame's worth of changes into a single update.
- Game objects that have been destroyed may continue to receive messages for a bit, be careful of reusing object id's.

Questions on design?

Key Lesson

Always use a source control system even if the number of coders is small. If nothing else, it will make quality control and finding elusive bugs *much* easier.

Source Control

- Don’t keep files out for days, check in after each function is changed and tested.
  - Makes it easier to backtrack bugs
  - Reduces conflicts with other code
  - Avoids problems if a person goes out sick.
- Mark a revision system-wide before EVERY test-run.
- Don’t forget to have some system for the art too, at least back up all of it (source and final materials) before each version is built.

Source Control Discipline

- Do daily builds of the source tree to make sure it compiles without error.
- When the build is broken, the number 1 priority is to fix it as leaving it broke can stop everyone’s work.
- Whoever checked in the code that broke the build must drop everything to fix it. The inconvenience will convince them not to do it again.
- Always bring your local sources up to date and do a test compile before checking in your new files.
- Have a QC person do frequent test builds and checks for bugs slipping in.

Key Lesson

Proper clock management is essential for a smooth running simulation game.

Physics, Dead Reckoning, Scheduling, Rendering and communications all need a stable high-res clock to run smoothly.

Clock Mechanisms

- These vary considerably by platform in both rate, precision, drift and cost of access.
- Beware system calls that return time in µs but actually "tick" at a much slower rate.
- Games should read the system and "derive" a game clock in standard units (ie: seconds) regardless of what the actual clock counts.
- It's not a sin to use floating point for a clock, but beware of floating point precision issues.
- It's a bad idea to use frame counters.
- You want many counts per frame to use for real-time scheduling and code profiling.

More on Clocks

- Measure the cost of generating/reading the clock, some systems can create unexpected overhead.
- Remember, a 32 bit µs clock will still wrap around every 71 minutes.
- Read the clock once per frame and use in all calculations (except scheduling) to avoid frame-to-frame “wobble” of your motion.
- Send clock reads through a bottleneck routine to let you “spoof” the clock for debugging.

Time Synchronization

- Because all internet connections are different, you must know the latency on each one to get clocks set right.
- Typical time-sync transaction
  - Send a packet with local time to a server
  - Server fills in it’s time and sends it back
  - ((received time-sent time)/2)+server time = time to set your local clock to.
- This works well, but isn’t perfect, do it several times and use an average result.

More Time Sync

- Filtering techniques can increase the accuracy of the measurements and detect differences in send/receive latency (see Internet NTP protocol for ideas).
- PC clocks *do* drift and network latency *does* change, so remember to re-measure and re-sync occasionally.
- Remember, you need a FAST local clock (microsecond) to get this right.
- Time stamping all messages is helpful for maintaining sync and doing replays.

Time & Dead Reckoning

- Once clocks are in sync, they can be used to smooth the effects of variable latency.
- Send position, velocity and acceleration so a remote computer can figure actual position based on time since the packet was sent.
- 150mph = .22 feet/ms so with a latency of 100ms and no dead reckoning every vehicle on your screen will be *22 feet* out of place making accurate collision detection really tough.
- Due to dead reckoning and careful latency hiding in the game design Red Planet could be played with as much as a quarter second of latency before people noticed problems.

When to send updates

- Sending updates on a regular schedule doesn't account for sudden control changes
- Setting a threshold on position, velocity and acceleration and updating when they exceed it works better.
- The best way is to run a dead-reckoning model in parallel with your motion model and send updates when the position error exceeds a certain pre-set amount.
- Try to avoid motion models that are difficult to predict.

Dead Reckoning Techniques

- The most obvious is standard physics equations, but they don't terrain follow.
- For games where you are in contact with the ground, dead reckoning needs to include terrain following and suspension systems.
- You need a system to merge true position (from net updates) with dead reckoned position and correct collision errors.
- Done right, update rates can be very low, Red Planet often ran several seconds.
- To work well, you must avoid motion models that allow extremely unpredictable motion (ie: Quake)

Key Lesson

For frame rates above 30hz, frame rate drops or slight errors in frame timing are very noticeable.

A steady, slightly slower, frame rate is often better than one that runs fast some times and slow others.

Frame Rate Issues

- As frame rate increases, momentary slowdowns become much more noticeable and disturbing.
- At 30hz things begin to feel fluid, at 60 some artifacts go away, like double images of moving lamp posts…etc.
- Maintaining a solid 30hz makes a sim feel much more dangerous/real.
- Make sure your game syncs to the monitor's actual refresh rate. Running at 60 when the monitor is at 72 creates annoying beat-frequency "tugs" every few seconds.
- Running the monitor at it's highest refresh rate reduces the impact of frame drops.

Questions on Clocks/Framerate??

Debugging and Tuning

Key Lesson

A little care in setting the right policies up front can make the debugging process much less painful.

Debugging Discipline

- Turn on all compiler warnings, eliminate all warnings before check in. A serious warning can go unnoticed among a raft of benign ones.
- *Always* fix code bugs as they are discovered.
  - You're only hunting one bug at a time.
  - Stops interaction of multiple bugs from creating more serious looking problems.
  - Prevents "cascade failures" where bugs can trigger a cascade of problems all over the system.
- Don't be satisfied with bugs that just "go away" because they will come back! Isolate, reproduce and understand each bug before you mark it solved.

Debugging Technique

- Clock spoofing can be used to slow down a program to make bugs easier to see and reproduce by slowing down a game while making it think it’s running at full speed.
- Arrange to have the screen and controls of every player video taped during test sessions.
- Record control inputs to RAM and be able to save them and play them back to reproduce a bug.
- “Mission Review” systems can be used to to test rendering, dead reckoning and infrastructure.

The Built in Self Test

- All systems and libraries should have self test routines that can be run independently of the game to verify correct functioning.
- Create a RAM allocator with seperate heaps for each major data structure, this makes finding memory corruption easier since unrelated data types aren’t mixed together.
- Flood fill allocated RAM with NANs or other patterns to detect uninitialized variables.
- Build test routines that do consistency and range checks on all values in a structure or message. Run this check before each use of the data.

Using Runtime Self Test

- Build in a “debug level” that controls how rigorous self tests will be, or lets you shut them off at runtime or compile time.
- Being able to change debug level without a recompile is very valuable during testing.
- If running the self test slows the game below realtime, spoof the clock so it thinks it’s running full speed. This lets all equations run with a normal time step.
- Careful clock spoofing can also let you stop/single step code with a debugger without messing up the time step.

Other Tips

- You want to have many different debug levels (we had 5) so most of the time you can run with at least some of the debugging stuff on.
- Basic system stats can often reveal problems that are tough to spot otherwise such as drawing every polygon twice.
- Remember that the built in debugging & test routines will effect the way the program operates, and can occasionally *cause* bugs to appear.
- Get a “packet sniffer” that can record all network traffic and decode packets. “Ethereal” on Linux works good and is free.

Key Lesson

The time it takes to find a bug is proportional to how stupid a mistake you made.

"If a bug is taking weeks to find, start looking for something *really* dumb"

The bug isn't always where you think it is.

Examples: 2 BattleTech Bugs

- 2 crash problems that took months to find
  - A random crash would sometimes occur as the game started or ended but couldn't be reproduced in the lab.
  - Novice games crashed and expert ones didn't, testers playing in novice mode couldn't reproduce it.
- There were many theories for both
  - A hardware design problem (we had a few)
  - Timing problems in the network packets
  - Power spikes
  - Novices bashing the hardware (*happened a lot*)
  - Some complex interaction of software bugs

The Random Start/End Crash

- Because it couldn’t be reproduced, this went on for many months.
- Software testing and code review turned up nothing.
- The debugging tool that finally solved the problem? My pants!
- Static electricity was causing sparks from some players to the controls at the start of the game, or when the player climbed out of the cockpit before the end of the game.

The Random Start/End Crash

- A classic example of the bug not being where we thought it was.
- A lot of time was lost looking for software, hardware or network bugs because we forgot to look for a simple physical explanation first.
- Crashes couldn’t be reproduced in the lab, because the air there wasn’t as dry as it was on-site.
- In retrospect, crash logs showed fewer of these crashes during wet or humid weather than during dry weather.
- We added a shock test using a simple piezo-electric gas igniter to our hardware test procedure.

The Novice Bug

- Could only be reproduced by letting novices play and after a dozen or so games, the crashes stopped
- All the cockpits would seem to crash at once and we didn’t have the debugging gear to cover all 8
- Execution tracebacks didn’t seem to make any sense, the crash was never in the same place twice.
- Many times we thought it had been fixed but the bug always came back.
- Nobody had a theory that made sense, the bug had to be isolated and reproduced somehow.

The Novice Bug

- The debugging tool that finally solved the problem? A hand full of rubber bands! (and a whole lot of caffeine)
- I was testing network throughput by strapping down the fire button and joystick with rubber bands, pushed the throttle forward and let mechs drive in circles while shooting.
- During the test, a cockpit exhibited the novice crash, the hunt is on!
- I found enough rubber bands to strap down all the cockpits at once, the novice crash happened less than a minute into each game

The Novice Bug

- Debugger output still didn’t make sense
- After about a hundred crashes I noticed the crosshairs of one of the cockpits always pointed at a mech’s shadow.
- Still couldn’t reliably reproduce the error, till I realized shooting the shadow of the lowest numbered mech on the screen was what caused it.
- The problem? An if < that should have been <=.
- Shadows were “decorative” objects that were rendered, but didn’t really exist.
- The bug allowed the first shadow to be treated as an object you could damage.

Explanation

- Shooting the shadow ran a damage handler on random memory sending (if it lived) garbage data to all the other cockpits causing memory overwrites and random crashes.
- Cascade errors and bad messages brought all the cockpits down in an instant, all in different ways.
- The built in self test system would have caught this in an instant, if it had existed then.
- Only a true novice could reproduce the error because once you had a little experience, you rarely shot a shadow instead of the mech.
- The actual error that cost us months was caused by a single character!

Debugging Questions??

Key Lesson

Tuning performance needs to be done methodically and analytically.

The first time someone says “I know where the speed problem is” and does a change that seems to make no difference, you know you’re doing tuning the wrong way.

Performance Tuning

- It's very easy to get into the habit of *guessing* why your code is slow rather than running careful tests and *knowing* why.
- Build statistics into every part of your system from the beginning and know *exactly* how much time everything takes.
- IPEAK and VTUNE are invaluable tools in the PC world.
- Profilers can be given time to run by using clock spoofing.

Build in real time profiling

- Statistical profilers are good for finding overall hotspots, but don't catch transient loads that cause frame skips.
- You can do your own real time monitoring.
  - Send entry/exit times of major routines to RAM
  - Also send stats, like queue depths to RAM
  - Write a simple program to display the results as graphs (or use Excel)
- This lets you see your program's progress in real time. Any routine that occasional takes too much time is immediately obvious.

Other Tuning Tips

- Always keep basic stats such as primitives drawn. Drawing everything twice happens more often than you'd think
- On-screen "Hospital EEG" displays are easy to do and much more useful than a FPS number
- You need a way of reproducing identical conditions so you can test the effects of changes you make. Replaying control inputs is a good way to do this.
- Add a "hot button" testers can hit when they see a glitch. Have it save as much of the game state and historical stats as possible.

Scheduling for Speed

- Many processes (sometimes even physics) don't need to be run every frame. Schedule them!
- The simplest technique is queuing a time delayed "run me" message into an event queue.
- Do essential tasks first, then let the lower priority stuff to run till you're at the end of the frame.
- Avoid schemes (like run when time = X) that might force many periodic routines to be run in the same frame, let them drift apart to naturally balance frame to frame load.
- You need to know the max run time of these tasks to avoid starting one to close to end of frame.

Don’t forget the hardware!

- In modern processors, cache use and instruction ordering can make huge differences in speed.
- Locating related functions together in RAM can often increase speed.
- Locality of code and data is very important, avoid mixing unrelated code/data.
  - Separate data pools for rendering, physics…etc
  - When possible, don’t alternate between unrelated processes, causing cache thrashing.

Simple Example, Searching

- An array of large structures, with one value (object number) that’s the search key.
- Each test of a key will load a lot of the related structure into cache, which you don’t really need.
- Put the keys in a parallel array, each test will cause multiple keys to be loaded into cache, speeding linear and even binary searches.
- Since less memory is touched, less cache is used.
- Not elegant, but definitely faster. Some architectures will run a linear search faster than a binary search if you do it this way.
- Sorting can also be accelerated in this way.

Some General Points

- As mentioned before, security wasn’t considered in the design.
- All this ran on a Pentium 90 with < 8 meg
- Typical design cycle was 9-12 months
- Site-to-site games could run on 64-128kbps ISDN connection with our routing scheme.
- Optimization of communications could have brought this lower, but this was low enough for us.

General Questions???

- Code design?
- Clock use?
- Network strategies?
- Debugging?
- Tuning and testing?
- What’s the fastest land animal?