HALO ENGINE

ANCESTRY

The Halo 2 engine is a direct-line descendant of the 1992 Pathways into Darkness engine originally written by Bungie for 68000 Macs using MPW and C. Not much remains except math functions and overall architecture, but the engine definitely has old roots. The path of descent runs through PiD to Marathon to Myth to Halo. In recent years it has shifted away from being primarily a player-driven renderer to being primarily a world simulation.

I'm going to talk about two main philosophies that you can trace through the engine and most of its components.

1. The unified resource model, editing and import tools, general memory layout, filesystem and streaming architecture.
2. Our approach to runtime data organization, memory allocation and world state.

STATISTICS

- 1.5M lines in 3,624 files for 52.9MB of code (mostly C, some C++, very little asm)
- Compile time for clean build of typical development build: 7:39
- Compile time for final shipping build (includes LTCG): 10:06
- Final executable size: 4,861,952 bytes
- Total assets under source control: 70 GB not counting localization
- Time to load a level, development build: 4 minutes
- Time to compile a level for shipping build: 9 minutes
- Time for complete build of all binaries on build farm: 18 minutes
- Time for complete build of all map resources on build farm: 53 minutes
- Size of final game: 4.2 GB
- Development time: 34 months
- Breakdown of production staff (56):
    - 17 engineers
    - 11 environment artists
    - 8 game artists
    - 7 designers
    - 6 animators
    - 3 producers
    - 3 sound designers
    - 1 technical artist
- Breakdown of support staff (59):
    - 2 administrative
    - 8 web / community
    - 5 test engineers

    - 10 test staff
    - 20 hourly testers
    - 14 localization testers

RESOURCE MODEL

We use a unified model for the majority of our game resource data called “tags”. One tag represents a single asset, from a bitmap to a sound to an AI character archetype.

- so named because they were originally (myth, 1995) stored in a flat file system with a single directory for each type of data, identified by a four-character ‘tag’. they are now stored in a hierarchical singly-rooted filesystem, identified by path and type of tag.
    - e.g. c:\halo2\tags\objects\characters\masterchief\masterchief.biped
- each type of tag corresponds exactly to a C structure. this structure is built up of variable-length arrays, each array element built up of atomic fields (integer, floating point, string ID, etc). arrays can also contain child arrays, raw binary data or references to other tags entirely. in this way a tag may represent essentially any hierarchical data storage format.
- each type of tag has a definition in code, built using macros, which specifies how the C structure is broken down into atomic types. this gives the ability to serialize to or from a file.
    - *problem: tag files useless without exactly matching code definition, low-level reading code means a mismatching definition and file may not be noticed if data transformation is outwardly benign, artists using old tools to edit tags*
    - *problem: need to check in new code and data simultaneously*
    - *problem: versioning support is poor to nonexistent, need to copy and paste old definition structures and provide manual conversion routines from v1 C structure to v2 C structure*
- 99.99% of all game data is stored in a tag of some kind. to load a level you must load the ‘globals’ tag (and all its references) and then the level’s ‘scenario’ tag (and all its references)
- 11.6GB in 39,000 tags under source control, a typical level will load 8,000 of those
- 130 different types of tag, with the most prevalent being
    - sound: 12,000
    - bitmap: 6,440
    - shader: 3,909
    - render model: 1,984
    - model: 1,951
    - collision model: 1,297
    - effect: 1,116

    - animation graph: 903
    - physics model: 776

Tags are edited using our unified tag editing tool called ‘guerilla’. This tool also serves as a source control client.

- each tag definition may also contain meta-information in the form of markup fields used to generate more helpful automated UI in guerilla
    - *problem: programmer-generated UI rather than designer-generated means designers cannot annotate tag (i.e. this feature doesn’t work, good values for this number are 0.2 – 0.5)*
- all tags may be edited directly using guerilla, but some fields are hidden or read-only except in expert mode
- our level editor is just a fancy GUI on top of a set of tags that define an individual level, if desired you can build a level using guerilla instead
- we also have command-line tools for all automated operations on tags, such as importing tags from source data or building lightmaps

While there is one unified format for game resource data, we use many formats for source assets that will eventually be converted into tags. We use the term ‘data file’ to refer to anything not read by the game itself.

- data contains two categories – raw source assets and exported intermediate assets
    - bitmaps: source psd, exported to tif
    - sounds: aiff
    - geometry: source max, maya, exported to text file format
    - strings: unicode txt
- in order to get a raw source asset into the game you must first export it to a tool-readable intermediate format
    - intermediate formats are either industry standards or simple text based formats that are as close to possible as the raw data stored in the source asset
    - we used to push more functionality into exporter plugins for various art packages, but found it much easier to maintain complexity if the complex code was part of our codebase rather than being integrated into someone else’s changing product
    - our plugins consist of just enough code to navigate the scene graph and write it to text format, plus some simple macros to automate the export process
- artists run the command-line import tool to convert the intermediate asset into a tag or set of tags
    - 1:1 mapping between the tags and data directory structures (character equivalence handy)

    - given a path to the tag that needs to be imported, the importer will look for the corresponding intermediate file(s) by name and turn them into a tag
    - c:\halo2\data\scenarios\solo\03a_oldmombasa\work\arcology2.max exported to c:\halo2\data\scenarios\solo\03a_oldmombasa\structure\earthcity_3.ASS imported to c:\halo2\tags\scenarios\03a_oldmombasa\earthcity_3.structure_bsp
        - *problem: because artists can store the source asset file anywhere they want, even though you can uniquely identify the intermediate asset file that was used to generate a particular tag, you cannot automatically determine which source asset file that was exported from. this information exists only in the head of the artist that last exported the asset.*
        - *problem: this imposes a limit on any batch processes, they can only operate on the exported intermediate format and you can never do a batch re-export from the authoring package.*
    - the importer does as much work as possible on the asset so that it can be loaded and prepared for use quickly, but tags must remain platform-independent, i.e. we will build BSPs and perform tri-stripping at import time, but would not swizzle data for SSE or create vertex buffers until runtime when the tag is loaded on a particular platform
    - some import tools will create a hierarchy of tags, for example importing a single environment geometry can create an entire set of unique object tags ready for placement
    - if a tag already exists, the importer will not overwrite it, instead loading it in and inserting the newly imported data. this is necessary because some types of imported tags are annotated with additional data
        - bitmaps have mipmap levels and cache usages
        - sounds have permutation skip information
        - animations have manual keyframe data
        - *problem: artists running old executables can have version conflicts and fail to load the existing tag's annotation, which can silently result in annotation being lost*
- integration into artist workflow
    - import tools are integrated into tag editing GUI so most import commands can be executed with a single click
    - tool can be started in a monitoring mode so that it will automatically reimport anything in a certain directory, avoiding the need for the artist to manually reimport when assets are changed
    - most assets are quick to import with the exception of level geometry, which can take 15-30 minutes with debug tool
        - artists have a release tool version available to them which has no asserts and speeds up the process dramatically

        - *problem: artists can create content that would trigger an assertion and never realise they have a problem since release build skips those asserts, resulting in corrupt data being generated and checked in*

One advantage of the unified resource model is that there is a single path for loading and processing resource files. This allows us to have some uniform processes which are surprisingly powerful.

- tags are loaded as individual files when needed
    - because tags are directly user-editable, we require that the game can never be broken by anything typed into a tag, even a corrupt tag file on disk
    - tags are loaded block by block into individually allocated chunks of memory, so that each tag array can be moved, resized or deleted if needed
        - as each block is deserialized we run byte-swapping (although not needed for halo 2)
    - tags may be loaded "for editing", in which case they remain a direct representation of the on-disk structure, or "for gameplay" which runs an additional set of processes necessary in order to prepare the tag for use
        - all atomic fields are bounds-checked
        - any runtime-only fields are initialized
        - ASCII strings are collected into a global string table and replaced by 32-bit ID for quick comparison, this is not done for localization but for name binding (e.g. an effect location named "muzzle" may be quickly found)
        - references to other tags are followed and those tags are recursively loaded
    - tag definitions may also contain custom postprocess code which runs after a tag has been loaded for gameplay
        - postprocessing may completely change the structure of a tag, deleting old information and creating entirely new arrays and blocks
        - postprocess routines are run after all tags that are referred to have been loaded, and are allowed read-only access to tags that are designated as load-time dependencies
            - e.g. object definition tag can generate a bounding radius based on looking at the render and collision geometry tags that it links to
            - *problem: difficult to enforce no other dependencies for postprocess procedure, easy to write non-standard code that breaks the tag loading paradigm in a non-obvious way*

    - after postprocessing, each tag can turn parts of itself into a cacheable block of data which is then written out to disk and replaced in memory with a handle to the cached block
- loading takes a while for several reasons
    - we are loading from thousands of files at once
        - this is a fairly big problem on xbox as the filesystem is deliberately not optimized
        - changing our folder hierarchy helped a lot, at the time of file transfer from pc to xbox we build a mapping between tag path name and an automatically generated filename which puts at most 100 files in a directory
        - paths look like xe:\halo2\tags\057\38
        - eventually we coalesced tags into a single monolithic file before copying, we need to come up with a better solution for the next generation
    - we are performing byte-by-byte processing as we load
        - inherent to any versioning and byte-swapping resource model
    - we perform hundreds of thousands of tiny memory allocations
        - our dynamic memory allocator is deliberately not high performance (more on that later)
    - load times remained manageable at 1-5 minutes on Xbox
    - about half that on PC in the worst case, but down around 15-20 seconds if the level was already warm in the filesystem cache
- tags may be reloaded at any time
    - creates a completely new copy of the tag in memory and discards the old one, pointing the global tag handle to the new data
    - one consequence of this is that the game is never allowed to store a pointer to any tag memory directly, all access must be through the handle to the root of the tag
        - e.g. suppose I am a character playing an animation. I cannot store a pointer to animation data, but instead must use a handle to my animation graph tag and some array indices that identify how to retrieve the data within that tag
    - some tags are designated as non-recoverable upon reload, meaning that if they are changed then the level is torn down and restarted
        - we will attempt to preserve some state such as camera position and player position
        - this setting is only used for global data such as the world structure bsp or the scenario tag that defines all object placement and population
    - for all others, the tag is simply reloaded while the game is running, and the game engine receives callbacks to indicate which resources have changed

        - any system that stores references to the internal structure of a tag needs to validate these references against the new structure
        - for example, if an animation graph changes then any objects which use that animation graph must look up their animations by name again
        - it is considered a bug that must be fixed if there is any way to crash the game by reloading data – with proper code structure, this is actually a lot less work than you might imagine
        - the biggest reason we can do this is that whenever a reloaded tag crashes the game due to cached assumptions about the tag's structure, we have enough information in our automated crash dump to know that the tag was reloaded recently. this usually helps track down the cached data quickly.
    - determining how to reload tags
        - all PC builds of the game monitor the filesystem for changed files in the tags directory and will reload on demand
        - this includes sapien our editing tool
            - we have separate applications for editing tags and editing the game world, which helps stability
        - you must manually initiate a copy of tags to the xbox
            - takes a few minutes to scan the PC hard drive for changes and copy the new tags to the xbox
            - because we create a new mapping between tag path and filename on the xbox, we do not need to pause the game while copying files as we don't always overwrite files
            - xbox client only has to watch its main index mapping file for changes
- the payoff
    - win scenarios
        - playing the game in the editor, alt-tab across and change the physics properties of some objects, then go back into the editor and they will behave differently
        - change the animation compression properties, reimport and view the new animation instantly on the xbox
        - fire a weapon effect, go to the PC and move a few sliders around on the particle system, go back and see what it looks like now
        - painting bitmaps in photoshop that automatically update on the in-game character model on the TV next to your desk
    - about 75% of our content is authored by people who, while they are creating the content, are looking at it running in the engine on an Xbox displayed through a TV
        - *exception: level geometry and lightmaps (because the import processes take too long)*

        - *exception: animation (because we don't have good tools to consistently set up and replay situations where multiple characters are interacting with an object in the environment)*
    - artists are empowered to make sure everything looks good on the target environment
        - for example they built their own set of reference lighting environments so that they could work on assets in a consistent fashion
        - less unexpected interactions between parts of the art pipeline as everyone is using it end-to-end all the time
        - both a benefit and a drawback: if some part of the engine gets broken or changes, you'll hear about it quickly (a good example is the dynamic object lighting model, which we struggled to make consistent for the whole project)
    - everyone on the team understands that unless the content is in the engine and working, the task is not complete
    - after several years and iterations of the process, we are now definitely in the magic zone where our artists and designers just assume stuff works and the system becomes transparent to them

The tag filesystem allows powerful editing but it is not well optimized for a console architecture. For our final build we have a processing step which compiles all of the resources needed to load a particular level of the game.

- development versions of the game run from thousands of individual tags
    - benefits: flexible, allows incremental modification on the fly, can load multiple levels quickly
    - downside: slow initial load, memory inefficient due to many small allocations, disk space inefficient due to verbose tag file format
- final builds (and release builds and test builds) only run from compiled level files
    - benefits: very fast load, memory optimized
    - downside: totally non-modifiable, takes time to build files, duplicated data between level files
- building a level "cache file"
    - command-line process loads the level exactly like the game, including loading all tags and postprocessing them
        - tags are given a chance to strip debug-only data
        - once loading is complete all tags may also perform level-specific processing such as building prediction lists and culling unnecessary sounds, animations, character variants etc
    - tags are divided up and streamed out into the cache file in pieces
        - global tag resources, always available

            - this resource buffer will be mapped in at a known virtual address on the xbox which is the same for every level
            - as a result it may contain all of the arbitrary tag structures with pointers rebased appropriately
        - loading-zone specific tag resources, only one zone will be available at any time
            - the zone buffers are mapped in immediately following the global resource buffer on the xbox
            - therefore once the global resource buffer is built we know the virtual address to be loaded at and the tag structures may be rebased automatically
        - cached data blocks
            - all texture, sound, geometry and animation blocks are written to individual partitions in the cache file
        - debug information
            - string tables and tag names are written to individual partitions in the cache file so they are available for debugging
            - they will never be loaded by the final shipping version of the game
        - after writing, the cache file is compressed with zlib
            - total size before compression: 180-270 MB for campaign levels, 50-80 MB for multiplayer levels
    - sharing problem
        - as mentioned earlier, having a separate cache file per level introduces a ton of duplication
        - this is mostly a concern for the bulk binary cached data as the tag size is minimal compared to textures and sounds, particularly combat dialogue
        - solution was to make cache files ‘dependent’ on other files, which means that they can reuse resources from those files
            - create new “shared” scenario type which is just a box level containing a bunch of characters and vehicles standing around
            - main menu level: no dependencies
            - multiplayer levels: depend on mainmenu, shared
            - campaign levels: depend on mainmenu, shared, single_player_shared
        - shared scenarios are built just like any other cache file (but must be built first)
            - generates a resource database describing all cached blocks, their originating tag, size and checksum of data

            - when a cache file is built, cached data blocks are checked against all shared resource databases and if a match is found then the block can be skipped and replaced with a reference to the appropriate shared file
        - you do not have to build shared scenarios in order to use a map, they just decrease total file size so we fit on DVD
            - decreases a campaign level from 700+MB to 250MB
    - build process is slow and cumbersome
        - working set of ~1 GB if you count both in-memory size and file cache
        - essentially impossible to do anything else on your machine for 2-8 minutes
        - we used an automated farm integrated with source control to build cache files for most of the project
- loading a cache file
    - file is copied from DVD to HDD
        - copy is performed in the background while playing previous level
        - performing zlib decompression in small chunks in memory
        - must also copy all dependent cache files
        - even though our DVD usage is the most optimal pattern you could imagine (large contiguous reads, multiple retries) we had a *lot* of problems with xbox DVD hardware
    - level load is almost instantaneous
        - page in the global resource buffer and the first zone buffer from the HDD in two synchronous read calls
            - global resources: 6-8 MB
            - zones: 2-5 MB
        - no iteration or fixup of loaded data necessary
            - *problem: no longer true now that we added havok, which needs to store vtable pointers in tag data. this requires us to fix up vtables after loading each resource buffer. this is the only exception to the rule and we will try to work around it for the future*
        - zone load is similarly almost instantaneous
            - *in theory we could make zone load even faster by pre-warming caches with blocks required for the new zone but we never got around to it*
    - memory layout
        - 64MB physical memory on xbox
            - we allocate a single allocation starting at 0x61000 (empirically the lowest we can, the kernel ROM we believe lives in this low memory, 388kb)
            - upper 13.8MB is reserved for all non-resource overhead

        - our application executable
        - static global allocations
        - heap allocations
        - system libraries
        - system allocations
    - remaining 49MB is allocated on a per level basis
        - saved game state, 4MB (more on this later)
        - networking, 3MB in a multiplayer level
        - fonts, unicode strings, etc
        - tag resources for that level
            - size of global buffer + MAX(size of any zone buffer)
            - our budget was 12 MB or less
    - everything else (36 – 40 MB depending on level) used for dynamic caches
        - animation cache
            - 3MB campaign / 4MB multiplayer
            - 2kb page size
            - 8-19MB cacheable data
        - sound cache
            - 3MB
            - 16kb page size
            - 300-500MB cacheable data
        - geometry cache
            - 6.5MB campaign / 7MB multiplayer or splitscreen co-op
            - 4kb page size
            - 20-45MB cacheable data
        - texture cache:
            - consumes all available memory (typically 17-21 MB, but other systems can steal from texture cache if they temporarily need memory for async I/O, etc)
            - 4kb page size
            - 80-140MB cacheable data

RUNTIME DATA STORAGE

Our philosophy behind runtime data storage is simple but has implications on many aspects of our engine. I'll just scratch the surface of this topic here but hopefully give you an indication of why we think this approach is useful. First, an overview of the principles we use in our engine.

- system-specific memory architecture

    - use pool based rather than heap based allocation
    - allows for decoupling and testing of failure modes
    - no unbounded failure cases
- correspondence between saved game and memory formats
    - fast to load and save
    - the closer that a saved game is to a memcpy(), the more likely it is that you can use a saved game as a bug reproduction scenario
    - structural similarity to our tag architecture with bulk resource loads to known addresses
- robust interoperability
    - try to separate code implementation from in-memory structures as much as possible
    - goal is significant similarity between data storage patterns on xbox shipping build and pc debug build
    - allows you to catch more problems early on in simple-to-debug cases

One of the primary storage containers we use is called a “datum array”. This is a fixed-length array with a constant number of slots for elements, or “datums”. Data is stored contiguously with a two-byte header for housekeeping at the start of each element.

- simple allocation policy
    - track the mark below which you know everything is allocated
    - allocate at the first free element above this (don’t use a free list)
    - done for allocation locality
    - allows write protection of a larger portion of array
- allocation and deallocation behaviour
    - zero-fill upon allocation
    - fill with garbage upon deletion in debug builds
    - assert before allocation that garbage pattern is still present
- elements are identified by a datum index
    - datum index is composed of two 16-bit values – absolute index of element in array, and a salt value
    - the salt is a negative 16-bit integer which is stored in the two-byte header of the element to allow a check for equality upon access
        - pseudo-unique salt values are generated upon element allocation and incremented for the next element’s use (wrap at 0xffff to 0x8000)
        - the initial salt value for a given array is generated from the first two ascii characters of the name of the array, i.e. “script” is 0x73 0x63 which becomes 0xe373
        - these patterns of salt generation are occasionally useful for debugging but are never relied upon by code

    - the index 0xffffffff, referred to as NONE is the equivalent of NULL for datum indices (we also use it as a sentinel for most other integer operations)
- access to datum entries
    - datum_get() takes a known datum index (strong reference)
        - asserts that absolute index is within bounds
        - asserts that salt is valid and matches allocated element
        - we strongly encourage that strong references are used everywhere
        - compiles down to &array->data[index & 0xffff]
    - datum_try_and_get() takes a previously known datum index (weak reference)
        - as above, asserts valid salt and absolute index, but salt is allowed to differ
        - will return NULL if element is now unallocated or salt does not match
    - datum_try_and_get_absolute() takes an absolute index without salt
        - asserts valid absolute index (i.e. salt is zero)
    - datum_try_and_get_unsafe() takes any 32-bit value
        - guaranteed not to assert
        - usage strongly discouraged for everything except reading from network or from disk
    - can iterate over all allocated elements
    - entries in a datum array are never referred to with pointers except for temporary caching
- common mistakes that use of datum arrays will catch and provide a verbose assertion
    - access to memory after it is deallocated and/or reallocated
    - usage of uninitialized references
    - bitwise corruption of a reference
    - memory overruns
        - because the unused portion of the datum array is write protected
        - because the garbage pattern is checked upon allocation
        - because the salt is checked upon access
    - allocation / deallocation outside of safe periods (during level load, zone switch, or at system initialization time)

Our memory storage patterns tend to enforce boundaries between systems in order to control interactions, although they are typically all built on top of a small number of underlying memory classes. All systems in our game reserve memory at application launch or level load time, in a constant order defined by code execution. (In other words, no constructors at global scope!)

- sources of basic memory

    - static globals
    - heap allocations (startup only)
    - physical memory (may be adjusted per level)
- many use datum arrays, some use fixed size memory pools
    - some use a datum array of small elements to point into a memory pool of variable size blocks
    - this allows rapid iteration over polymorphic elements stored in a single access-governed structure, and is cache friendly
    - we use this for our world entity storage
- all game allocations are bounded
    - zero heap allocations at runtime!
    - requires some amount of overhead as we are not able to overlap unused heap space
    - however, out-of-memory conditions become isolated to individual systems, therefore easier to design and test in isolation, also ensures determinism in these conditions
    - guarantees a high level of game stability and certainty under load on multiple different systems in unanticipated conditions
        - halo hackers – flying warthog firing rockets, grunt cannon, etc
- big exception is havok
    - havok heap usage is quite predictable based on what it is asked to do during the simulation update
        - however there is no way to know what this will be without actually performing the update
        - result: effectively unbounded memory allocation can occur during a simulation timestep
        - particularly bad are coincident objects, complex multibody collisions and creation of large simulation islands
    - we provide a page-based allocator for it that is reserved some fixed amount of memory
        - after each timestep we assess how close it is to the limit
        - perform garbage collection with varying aggressiveness
        - if the limit is exceeded we *must* get below the limit before we can proceed
    - tiered overflow pools provide temporary excess storage
        - first overflow: temporary buffer stolen from rendering, no ill effects
        - second overflow: start evicting data from runtime texture cache, has noticeable visual impact
    - intra-timestep allocations can still blow all overflow pools
        - more than 10MB allocated -> crash
        - we've never seen this in practice

Independent of the source of basic memory, each system can also be classified according to its runtime usage, which defines its lifetime and persistence.

- runtime memory can be strictly divided into usage classes
    - global application state
        - render targets, pushbuffer, system interface, I/O
    - deterministic world “gamestate”
        - players, objects, physics, AI, sound tracking, scripting, etc
        - also contains some known and tracked non-determinism from factors such as camera, player controller attachment, particle systems, HUD, etc
    - non-deterministic world view
        - rendering, sound playback, networking, caches, UI, etc
- gamestate memory organization
    - systems which need gamestate memory allocate it at application initialization time
    - this memory is allocated sequentially out of a fixed size buffer (~4MB) which comes out of the physical memory map
    - always located at a known virtual base address on both pc and xbox
    - fixed order of system initialization and fixed size allocations mean that a given chunk of gamestate memory is always allocated at the same address every startup
- saved game format
    - write out gamestate in a single chunk to file
        - can be done asynchronously but even synchronously takes <<1 sec
    - to load, read in gamestate over in-memory version
        - check crc, version and level information to ensure compatibility with current global application state
    - some number of functions applied before and after save/load
        - the majority are functions to fixup new gamestate after load
        - primarily, clear out references to non-deterministic world view components that weren’t part of loaded context
            - cached rendering info
            - reattach players to controllers
            - reset sound, I/O, networking
    - gamestate is required to be compatible between different compiled builds (debug vs release)
        - powerful debugging tool for weird one-off occurrences in release builds that you want to examine under the debugger
        - compatible from one executable version to another as well, as long as no gamestate structure changes, the resource files are all still loaded in the same order, and no resource files change internally in any way that breaks references stored to them

- determinism
  - gamestate updates are deterministic from run to run if provided with identical deterministic inputs from players
    - no dependence on rendering
    - this is how halo 1 networking worked
    - this is how halo 2 networked co-op works (not finished)
  - largely deterministic between binary versions and cross-platform
    - some numerical issues due to floating point instruction optimization
    - cross-platform saved games not possible due to different virtual base addresses, have to start from level load
    - powerful debugging tool for reproducing crashes, as we make backup copies of gamestate every 1 second and every 30 seconds
  - majority of havok is not compatible with gamestate principles
    - it stores many internal pointers and has static allocations inside its libraries
    - our solution is to completely recreate havok state from the deterministic gamestate upon every load
    - this requires us to destroy and recreate upon every save as well! in other words, the act of saving the game affects the events in the game world
- consequences
  - no #ifdef DEBUG
    - must use separate parallel non-deterministic arrays!
  - no dependence on non-deterministic world view
    - all determinism-affecting LOD (e.g. animation) must be camera-independent and based on all players in world, not just locally controlled players
    - total separation between interface and simulation state
  - no dependence on file I/O
    - implies we cannot do anything that affects deterministic state based on availability of cached data
    - all we can do is block until gamestate-required data is available
    - fortunately this is just animation, all other cached data is non-deterministic
  - no dependence on framerate or performance timing
  - many of these are desirable properties in an engine anyway, because it's good to have reproducible test cases