Halo

Chris Butcher
and 55 other people

Overview

- Ancestry
- Statistics
- Resource Model
- Runtime Data Architecture

Ancestry of the Halo Engine

- As old as Bungie (Pathways, 1992)
- Primarily written in C, some C++
- Platform-neutral foundations
  - PC / Console
- At heart, a world simulation engine

Vital Statistics: Code

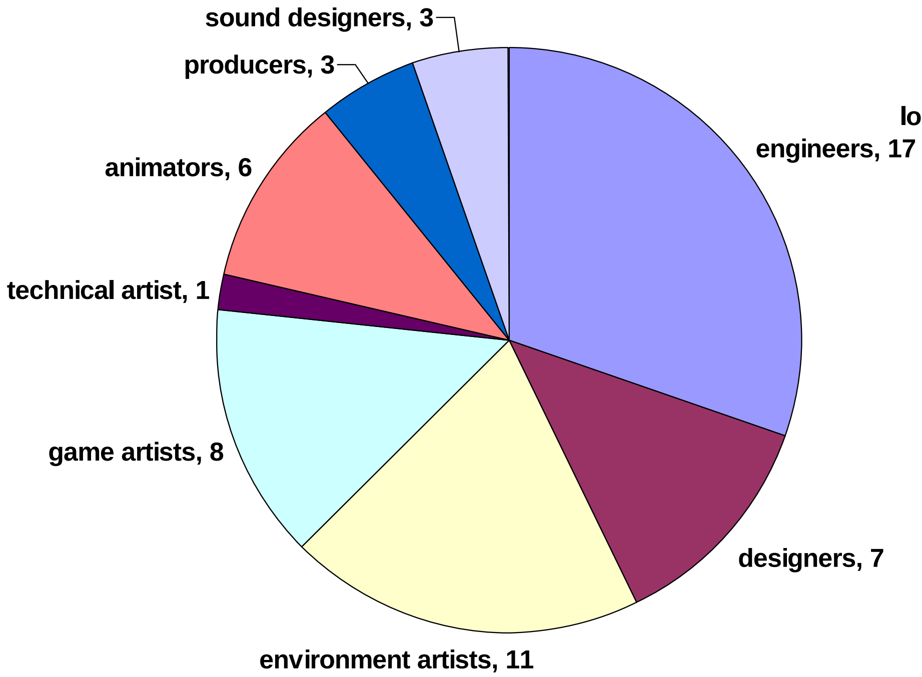
- 1.5MLOC in 3,624 files for 53MB of source
- Decent build times
  - Xbox Development build – 7:39
  - Xbox Shipping build (LTCG) – 10:06
  - Build farm (binaries) – 18 minutes
  - Build farm (complete game) – 53 minutes
- Shipping executable 4,861,952 bytes

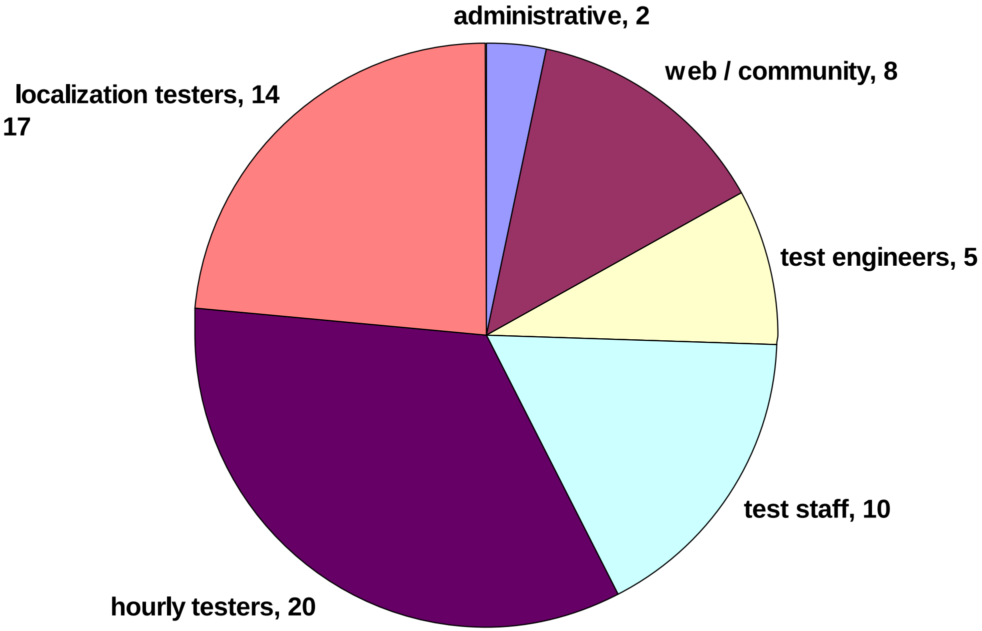
Vital Statistics: Resources

- 70GB in source control (Source Depot)
  - Not counting localization
- Level load: 4 minutes
- Level compile: 9 minutes
- Compiled level load: ~700ms
- Final shipping game: 4.2GB x8 SKUs

Vital Statistics: Development

- 34 month development time (12/01-10/04)





Resource Model

- “Tag” File Organization
- Unified Tag Editor
- Loading / Post-Process
- Compiled Cache Files
- Memory Layout / Streaming

“Tag” Resources

- Name is a historical artefact (Myth, 1995)
- Singly-rooted hierarchical namespace
  - `Type: BIPED, Path: objects\characters\grunt\grunt`
- Stored as individual files on host system
  - `c:\halo2\tags\objects\characters\grunt\grunt.biped`
- 99.99% of all data is a tag
  - Exceptions: loading screens, fonts

Tag Structure

- Hierarchy of variable-length ‘block’ arrays
  - Each block contains 0-*n* fixed-size elements
  - Topmost block contains exactly 1 element
- Block elements are built from atomic fields
  - Integer, Enum, Floating point, String, Text
  - Flags, Map function, Pixel shader
  - Child blocks, Binary data blobs
  - References to other tags

Tag Block Definition

- Blocks map directly to C structures
  - Described by separate macro definition

```
struct ai_properties
{
    word flags;
    short ai_size;

    string_id type_name;
    real leap_jump_speed;
};
```

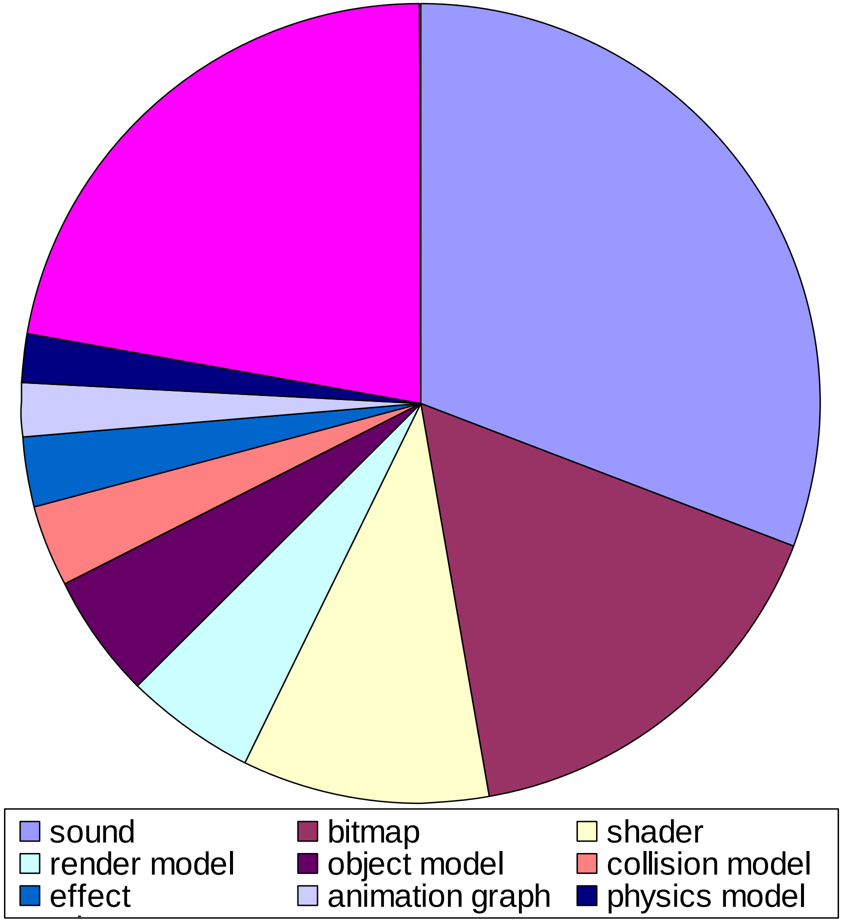
```
TAG_BLOCK(ai_properties_block, 1,
        sizeof(struct ai_properties), NULL, NULL)
{
    {_field_flags, "ai flags", &ai_properties_flags},
    {_field_enum, "ai size", &ai_size_enum},

    {_field_string_id, "ai type name"},
    {_field_real, "leap jump speed"},
    {_field_terminator}
};
```

Tag Block Definition

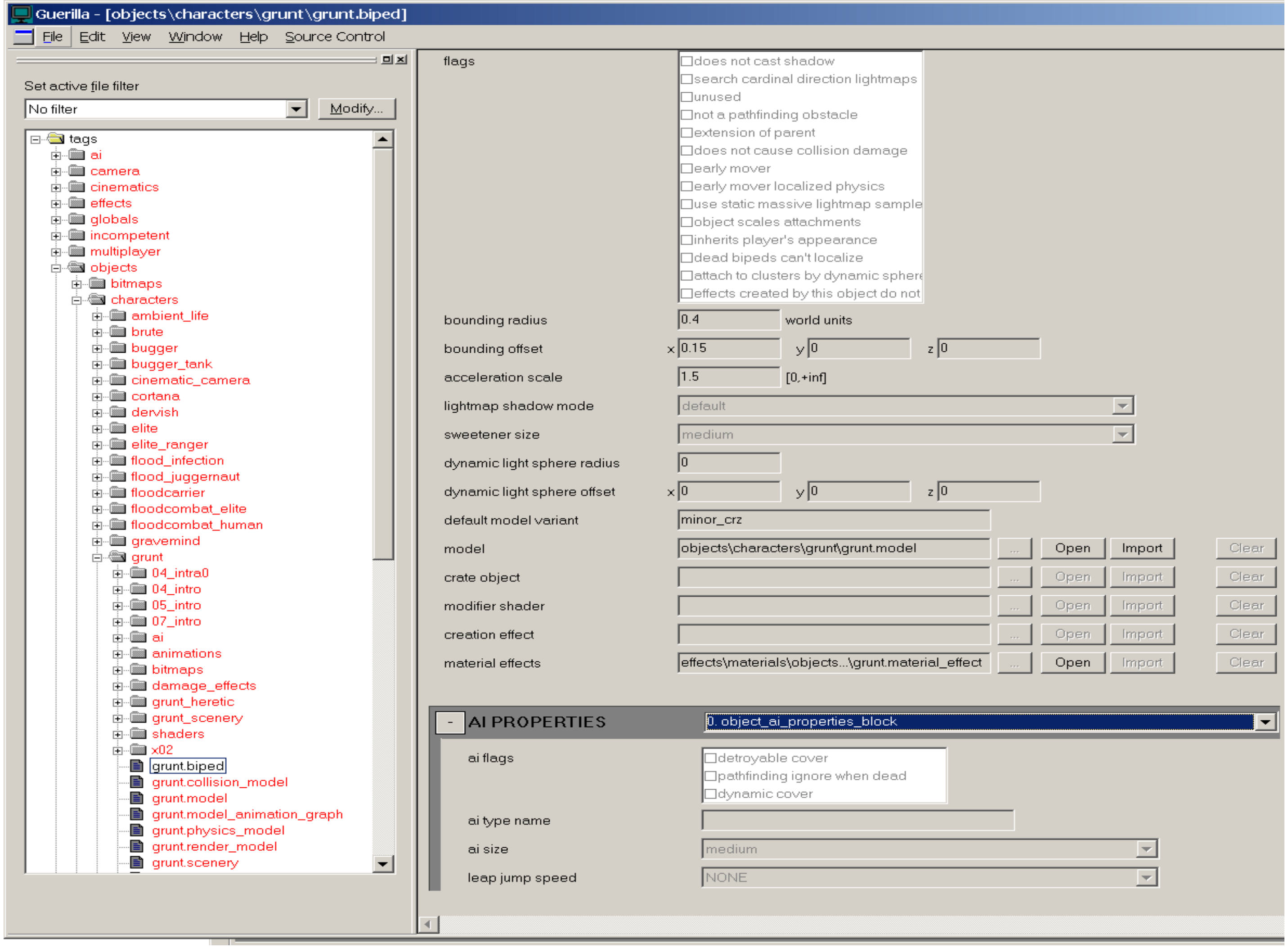
- Definition structure allows introspection
  - Automatic serialization of hierarchical tag
  - Byte-swapped upon load and save
  - Duplication, insertion, deletion of elements
  - Not needed at runtime (no RTTI)
- Simple file format
  - Requires exactly matching code and definition
  - Limited versioning support

Tag Data

- 11.6GB, 39,000 tags
- To load a level:
  - □ Load globals tag
  - □ Load scenario tag
  - □ Resolve dependencies
  - □ Typically 8,000 tags
- 130 types of tag



Tag Editing (Guerilla)



Guerilla - [objects\characters\grunt\grunt.biped]

Set active file filter

No filter

- tags
  - ai
  - camera
  - cinematics
  - effects
  - globals
  - incompetent
  - multiplayer
  - objects
    - bitmaps
    - characters
      - ambient_life
      - brute
      - bugger
      - bugger_tank
      - cinematic_camera
      - cortana
      - dervish
      - elite
      - elite_ranger
      - flood_infection
      - flood_juggernaut
      - floodcarrier
      - floodcombat_elite
      - floodcombat_human
      - gravemind
      - grunt
        - 04_intra0
        - 04_intro
        - 05_intro
        - 07_intro
        - ai
        - animations
        - bitmaps
        - damage_effects
        - grunt_heretic
        - grunt_scenery
        - shaders
        - x02
        - grunt.biped
        - grunt.collision_model
        - grunt.model
        - grunt.model_animation_graph
        - grunt.physics_model
        - grunt.render_model
        - grunt.scenery

flags

- □does not cast shadow
- □search cardinal direction lightmaps
- □unused
- □not a pathfinding obstacle
- □extension of parent
- □does not cause collision damage
- □early mover
- □early mover localized physics
- □use static massive lightmap sample
- □object scales attachments
- □inherits player's appearance
- □dead bipeds can't localize
- □attach to clusters by dynamic sphere
- □effects created by this object do not

bounding radius 0.4 world units

bounding offset x 0.15 y 0 z 0

acceleration scale 1.5 [0,+inf]

lightmap shadow mode default

sweetener size medium

dynamic light sphere radius 0

dynamic light sphere offset x 0 y 0 z 0

default model variant minor_crz

model objects\characters\grunt\grunt.model

crate object

modifier shader

creation effect

material effects effects\materials\objects...\grunt.material_effect

- AI PROPERTIES 0. object_ai_properties_block

ai flags

- □detroyable cover
- □pathfinding ignore when dead
- □dynamic cover

ai type name

ai size medium

leap jump speed NONE

Tag Editing

- Automatic editing UI from definition
  - Additional markup fields to format nicely
- Some fields hidden or read-only
  - Unless you use ‘expert mode’
- Map editor is just custom UI on top of tags
- Command-line tools all manipulate tags

Source Data

- Anything not read by the game
  - Source assets: PSD, MAX
  - Tool-ready intermediate: TIFF, AIFF, ASS
- Command-line import tool
  - `c:\halo2\data\scenarios\solo\03a_oldmombasa\work\arcology2.max`
  - `c:\halo2\data\scenarios\solo\03a_oldmombasa\structure\earthcity_3.ass`
  - `c:\halo2\tags\scenarios\solo\03a_oldmombasa\earthcity_3.structure_bsp`
- Produces one or more tags
  - Still platform-neutral until load time

Artist Workflow

- Import tools integrated into Guerilla GUI
- Monitoring mode for automatic import
  - Single-click export from Photoshop
- Import times in 5 second range
  - Except for level import, 10-30 minutes
  - Artists have release build of import tool

Tag Loading

- Deserialize tag blocks into memory
  - For “editing” or for “gameplay”
  - Bounds-check and sanitize all tag fields
- Custom postprocess operations
  - Read-only access to all dependent tags
  - Generation of platform-specific runtime data
  - Write out cacheable data as binary blobs

Loading is Slow!

- Thousands of files
  - Xbox path remap: xe:\halo2\tags\057\38
- Byte-by-byte processing
- Hundreds of thousands of mallocs
- Still manageable but not great
  - 1-5 minutes on Xbox
  - 1-3 minutes on PC or 20 sec with warm cache

Reload Anything

- Completely new copy of tag in memory
    - Game must never store pointers to tag data!
- Map or BSP reloads force level restart
- Everything else on the fly
    - Game receives callback after load
    - Must validate internal references to tag
    - Crash on reload == bug that must be fixed!

When to Reload

- PC applications use filesystem monitoring
  - Both game and map editor
- Manually initiate tag sync with Xbox
  - Scan hard drive of host system for changes
  - Copy any changed tags
  - Update path mapping file
  - Xbox client watches for new mapping file

The Payoff

- Seamless editing environment
  - Change any data, see it immediately (3-5 sec)
- Everyone in the engine all the time
  - 75% of content authored on target system
  - Artists create directly for target environment
  - Unless it’s working in the engine, it’s not done
- After many iterations becomes transparent

Compiled Levels

- Development builds: 8,000 files
  - Pro: Flexible, incremental editing, fast reload
  - Con: Initial load, memory usage, disk space
- Profiling, testing, and ship builds: 1 file
  - Pro: Fast load, memory optimized
  - Con: Non-editable, compile time, disk space
  - Built locally or by build farm

Cache File Building

- Load level, perform final postprocessing
- Divide up and stream data into partitions
    - Global resource buffer
    - Zone-specific resource buffer
    - Cached data blocks
    - Debug information
- 180-270MB solo, 50-80MB multiplayer
- 1GB working set, machine becomes unusable

Cache Sharing

- Duplication of data across levels
- Solution: Cache file dependencies
  - Blocks compared with dependent cache files
  - Write out reference to dependent file instead
- Custom shared scenarios for SP & MP
  - Not necessary to build a cache file
  - 700MB -> 270MB ensures we fit on DVD-9

Cache Loading

- Copy from DVD to HDD and decompress
- Super fast load
    - Page in global and initial zone resources
    - Global: 6-8MB, Zone: 2-5MB, read in <<1sec
    - No iteration or fixup necessary
        - Well... not strictly true due to Havok
- Warm caches before rendering frame 0

Memory Layout

- 64MB physical memory on Xbox
  - 13.9MB for static globals
    - Kernel, Executable, Globals, Heap, Libraries
  - 4MB world state
  - 3MB networking (MP only)
  - Tag resource buffers: Global + MAX(Zones)
    - Budget: 12MB or less
  - Everything else (36-40MB): dynamic caches

Cache Architecture

- Animation: 3MB solo, 4MB multiplayer
  - 8-19MB cacheable data, 2kb page size
- Sound: 3MB
  - 300-500MB cacheable data, 16kb page size
- Geometry: 6.5MB solo, 7MB MP or co-op
  - 20-45MB cacheable data, 4kb page size
- Texture: Everything else (17-21 MB)
  - Other systems temporarily steal from texture cache
  - 80-140MB cacheable data, 4kb page size

Runtime Data Storage

- Follows many principles of resource model
- Per-system memory compartments
  - Decouple and bound most failure cases
- Direct map from memory to savegame
  - Fast to load/save, good reproducibility
- Data Interoperability
  - Less ship-only bugs, ease of debugging

Datum Array

- Fixed-length array allocation
- Allocate only at first free index
  - Provides locality and allows data protection
- Fill upon allocate and deallocate
- Access elements through 32-bit identifier
  - 16-bit index, 16-bit pseudounique ‘salt’

Datum Access

- Known datum identifier (strong reference)
  - Asserts absolute index bounds, matching salt
  - Compiles to &array->data[identifier & 0xffff]
- Previous datum identifier (weak reference)
  - Salt must be valid but can differ
- Absolute index without salt
- Through iteration

Easy Catches

- Element access after delete/reallocate
- Uninitialized or bitwise corrupted identifiers
- Memory overruns
  - Through data protection, mismatch to known fill pattern, or salt overwrite
- Access outside safe time periods
  - Application launch, level load, zone switch

System Allocation Patterns

- Constant usage pattern
- Reserve memory at launch or level load
  - Code execution path defines ordering
- Basic memory types
  - Static globals at file scope (discouraged)
  - Heap allocations (startup only)
  - Physical memory map (dynamic per level)

All Allocations are Bounded

- Use datum arrays or pool based allocators
- Zero heap allocations at runtime! (Mostly.)
- Incurs overhead due to unused space
- Out-of-memory conditions are isolated
  - Easier to design for, easy to test in isolation
  - Provides general stability under load on multiple systems in unexpected situations

Big Exception: Havok

- Heap usage highly predictable...
  - ... if results of simulation timestep are known
- Page allocator uses fixed memory reserve
  - Monitor usage after each timestep and GC
  - Tiered overflow pools for temporary excess
  - Must get rid of all excess each timestep
- Intra-step allocations could blow all pools

Runtime Usage Classes

- Categorized by lifetime and persistence
- Global application state
  - Render targets, system state, I/O, tags, cache
- Deterministic world “gamestate”
  - Players, objects, physics, scripting, AI, etc
- Non-deterministic “world view”
  - Rendering, sound, networking, input, UI, etc

Gamestate Memory

- Gamestate systems allocate at launch
    - Sequential allocation from 4MB buffer
    - Located at known addresses on PC and Xbox
- Fixed initialization order and size
    - Each gamestate memory chunk is always allocated at the same virtual address

Savegame format

- Write out gamestate buffer to file
  - Single write <<1sec, or can be asynchronous
- To load, read over in-memory gamestate
  - Apply some small fixups before and after load
  - Clear references to non-deterministic state
- Require compatibility between different builds (debug vs release)

Determinism

- Gamestate is deterministic with identical input from external sources (players)
  - Somewhat so between binaries and platforms
  - Some floating point issues
- Majority of Havok not in the gamestate
  - Many internal pointers and static storage
  - Recreate all Havok from gamestate upon load

Consequences

- No #ifdef DEBUG in game data structures
- No dependencies on world view
  - Including game-affecting LOD (e.g. animation)
- No dependencies on file I/O
  - Cannot affect game based on caching!
- No dependence on framerate or perf times
- Many of these are good properties anyway

Summary

- Lots of simple choices
- Implications on engine and data design are interesting

- Questions?
  - Now, or mailto: butcher@bungie.com