Scaling the Pipeline

Stefan Boberg
Technical Director, Frostbite

@bionicbeagle

@FrostbiteEngine

Frostbite Engine

- Started 2004 as common DICE engine initiative
- Now in general use within EA Games label
- ~15 titles in development
- Diverse genres!

Frostbite Engine – FPS

DANGER CLOSE

DICE

DIGITAL ILLUSIONS CREATIVE ENTERTAINMENT

AN EA COMPANY

Frostbite Engine – Racing

Criterion games

Frostbite Engine – RPG

BIOWARE

Frostbite Engine – Action

Frostbite Engine – RTS

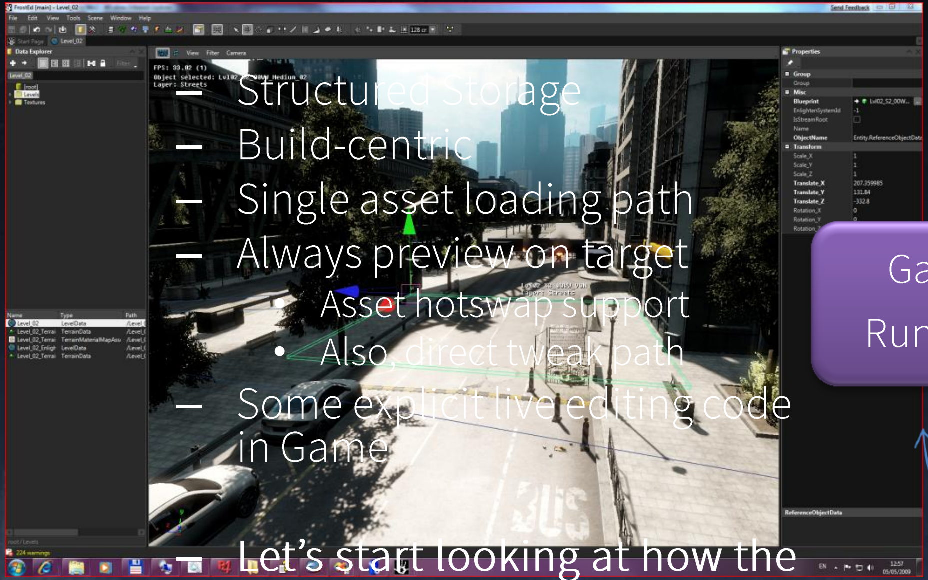
Scale / Dimensions

- Multi-site collaboration
  - Shanghai, Europe, North America
- Large teams
  - 400+ contributors in some cases
- Multiple VCS branches
- Many target platforms
  - PC, PS3, Xbox 360
- Content rich games

Scale – Example (Battlefield 3)

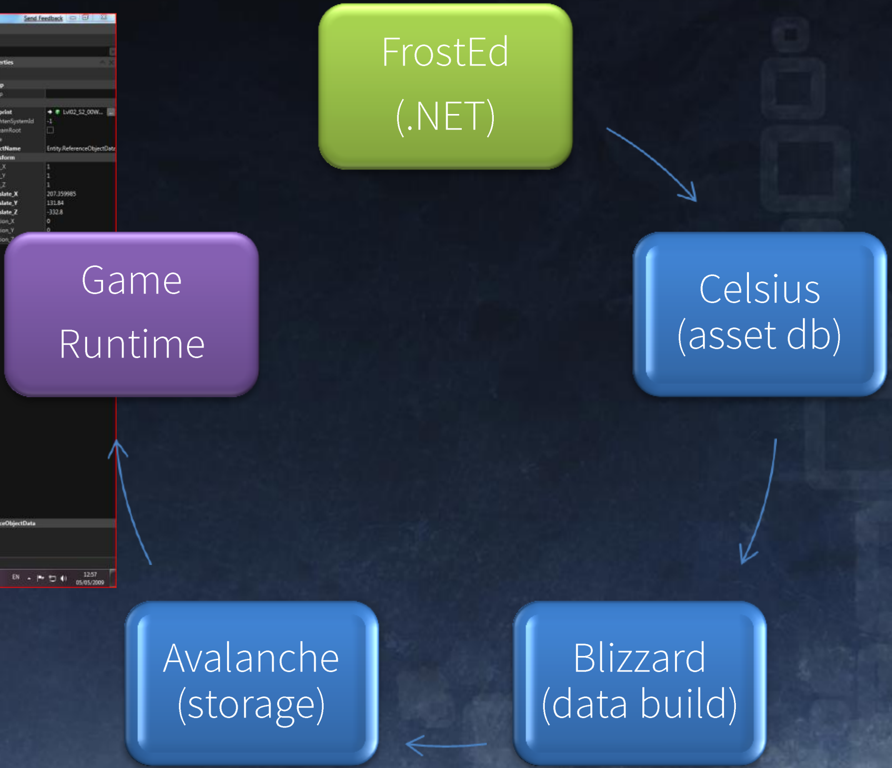
- Ballpark size
  - 500GB raw DCC assets
  - 80GB native Frostbite assets ('source data'), 100k files
  - ~18GB target data (PC)
  - 100,000 individual build steps (PC)
- Current games in development are larger
  - Hello Bioware! ☺

Frostbite Architecture

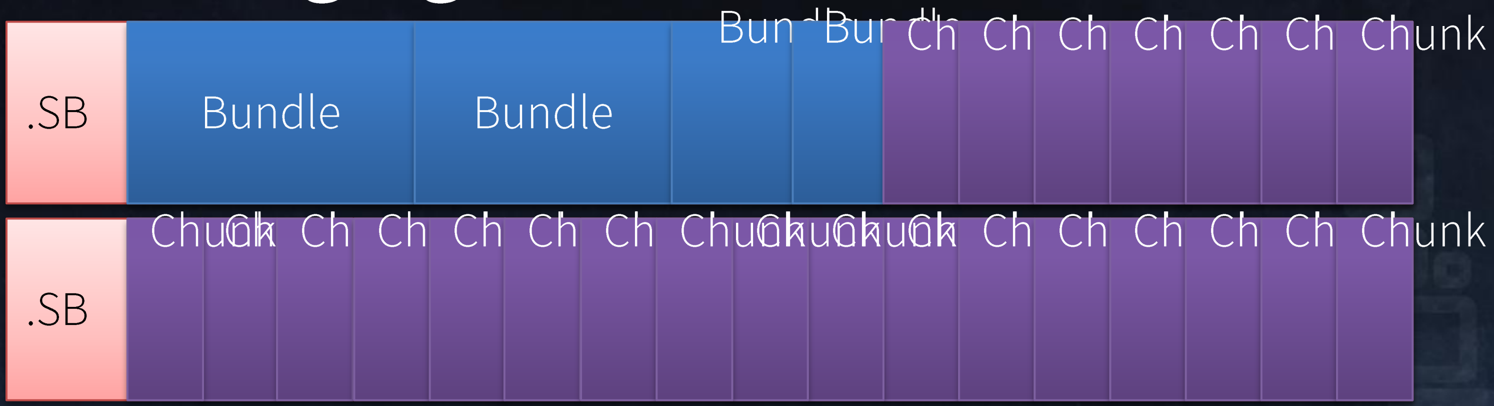


– Structured Storage
– Build-centric
– Single asset loading path
– Always preview on target
    • Asset hotswap support
    • Also, direct tweak path
– Some explicit live editing code in Game

– Let’s start looking at how the game handles assets and work our way ”backwards” through the pipe.



Asset Packaging Model



- Bundles
  - Linear stream of assets (usually)
  - Levels, sublevels (streaming)
- Chunks
  - Free streaming data blobs
  - Texture mips, movies, meshes

- Chunks are *random access* (pull)
- Bundles are *linear read* only (push)
- Superbundles are container files, storing bundles and chunks
  - Data inside is visible once superbundle is *mounted*

Packaging

- During development, layout is stored in Avalanche Storage Service
  - Stores full description of bundles and superbundles
  - Stored as packages with *chunk* references
  - Bundles are assembled on-the-fly when requested by game / tools (via HTTP)
  - Game does not know the difference between network and disk builds (single path)
- Complete packaging logic is executed every build pass
  - Including iterative builds!
  - So must be very fast

Avalanche Storage Service

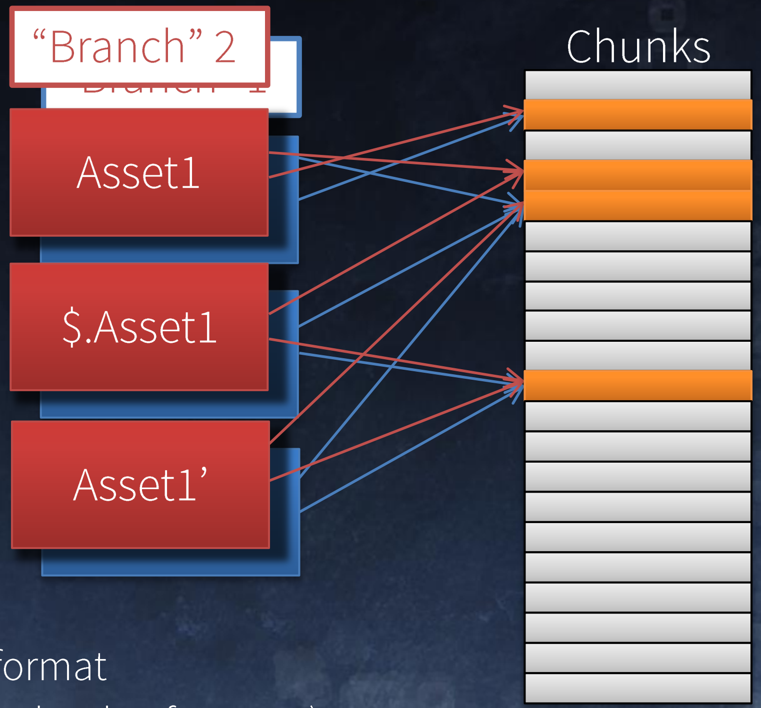
- Core Frostbite component
- Every developer runs an instance locally
- Windows Service
  - RESTful HTTP interface
  - Via HTTP (server implemented using **http.sys** API)
    - **http.sys** can serve data directly (in kernel mode) from system page cache (zero copy)
    - Very scalable
  - … with significant optimizations for local access

Avalanche Storage Service

- Target (i.e built) data storage
- Chunk store
- Build results and build dependency tracking information
- Layout information (defines packaging)
- Network cache / build cache
- Peer-to-peer build distribution
- Message bus
- Tracing/diagnostics infrastructure
- Production metrics infrastructure
- ... and more

Avalanche Storage Service – Chunk Store

- Content Addressable Storage
  - Key-value storage (immutable)
  - **key** = SHA1(**value**)
  - Basic deduplication
  - See: Venti, Git
- Other services rely on this heavily
  - Build Store
  - Build Cache
  - Asset Database (Celsius)
- Base primitive: ”Package”
  - JSON/BSON-like binary document format
  - Usually with attachments (stored as chunk references)



Asset Pipeline Goals

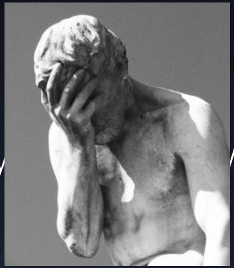
- Time spent waiting for build = waste
- Optimize bootstrap time (initial build)
  - Build throughput
- Optimize feedback time (iterative builds)
- Large games require extremely scalable solutions
- Challenging!
  - And a bit of a thankless task… if people notice your work, it's probably because you broke something, or it's too slow! ☺

Back of the envelope… bootstrap time

- Building entire game from scratch
  - Input: ~80 GB source data
  - Output: ~18 GB target data
  - Total: 100GB

- If we would read and write all data at full speed
  - Let's say ~50MB/sec throughput
  - 100*1024/50 = 2048 sec = 34 minutes
  - … and that's without any CPU work at all
  - … and with no additional I/O for temporary assets

System Performance

- CPU Performance
  - Well understood
  - Reasonable tools for analy  /ery Sleepy, GlowCode, VS)
  - Strings, strings, strings ….
- Storage
  - Not quite as well understood among game developers
  - Often overlooked, often the bottleneck!
  - Limited analysis tool knowledge
    - ETW/Xperf (Windows Performance Toolkit)
    - Resource Monitor, Performance Counters, code instrumentation

Storage Hierarchy

	Typical Latency	Typical Throughput
Registers	< 1nsec	n/a
Cache	< 10nsec	> 100G/sec
DRAM	< 500nsec	> 1G/sec
Network Cache	< 50 µsec	n/a
SSD	< 200 µsec	>200M/sec
HDD	< 20ms	>50M/sec

Storage *Hierarchy*

- It’s a CACHE HIERARCHY
  - Larger caches help performance
  - Free system RAM is used as cache
- DON’T FORGET TO PUT A LOT OF MEMORY INTO WORKSTATIONS
  - It will reduce the impact of I/O
  - Working set fits in free RAM -> GOOD!
- If the working set does not fit in system cache, performance falls off a cliff
  - Just like CPU work when you don’t stay in L1/L2/L3 cache
- We recommend our teams to get 32GB RAM workstations when purchasing
  - *They don’t!*

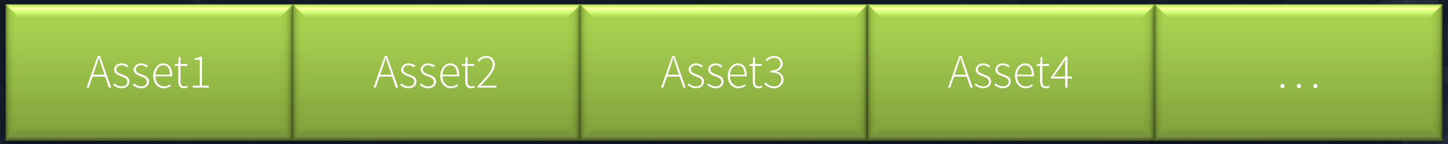
Storage

- The fastest I/O request is the one you don't!

- Mission:
  - Reduce seeks
  - Reduce blocking on I/O

Build Output - Avalanche

- We don’t use traditional file system storage for storing build function outputs
- We use packages and Log Structured Storage



- Benefits:
  - Single ’oplog’ file stores all build state (also used for dependency tracking)
  - Mostly sequential I/O (attachments are stored in separate CAS pool)
  - No fragmentation
  - No file open/close overhead
  - Cheap ’branching’
  - Simple copying of build state from one machine to another – pick up where they left off

Network Cache

- Network is often faster than local storage
  - *Assuming data is in server RAM*
  - Remember the storage hierarchy!
  - So ensure cache server has plenty of RAM
  - Ideally the entire working set should fit in memory
  - DICE server currently has 32GB RAM
    - Should probably be upgraded
- All nodes run an instance, and queries always go through the local instance
  - Hierarchical
  - Flexible topology (also: WAN replication for remote sites)

Network Cache – Basic Cache

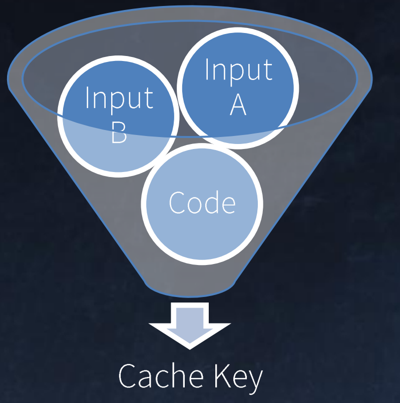
- Uses key-value store over HTTP
- Values are opaque blobs
- Implemented in *Avalanche Storage Service*
  - *Using HTTP.SYS API – very efficient and scalable*
  - *Leverages system cache for maximum throughput – no dedicated buffering*
- Data stored in Chunk Store
  - Content Addressed Storage (CAS), SHA1 key
  - Same content – same key => basic deduplication (”single instance storage”)
  - As used by Git, Venti, etc
- Metadata in Google LevelDB
- Used for misc ad hoc caching (shaders, expensive computations)

Network Cache – Package Cache

- Structured values (”package”)
  - Essentially, JSON documents (but in custom BSON-like binary format)
  - Each package may have BLOB attachments
- Used for data build caching
- Same basic format used to persist ’normal’ build results
  - Same data can be referenced from build results and cache package
  - Zero copy
  - Async fetch of bulk data not required for making build progress

Build Caching Implementation

- Keys generated from build inputs
  - Input file contents (SHA1)
  - Other state (build settings, etc)
  - Build function version ('manual' hash)
- Cacheable build functions split into two phases
  - First phase registers all the inputs
  - Second phase does the work
- Build scheduler
  - Executes first phase
  - Queries cache
  - Use results if available – otherwise run second phase



Build Model

- Apply function to map source data to target
  - $asset_{target} = f(asset_{source}, \ldots)$
- Goal: purely functional, no side effects!
  - Easy parallelism
  - Lazy Evaluation
- Not quite there yet.
- Requires some adjustment and initially more mental energy than the unfortunately very common basic ”blobby” and very stateful build structure.

Benchmark – BF3 (PC)

- Produces ~18GB build
  - Time includes "indexing" – i.e determining relationships between assets, metadata extraction, SHA1 hash for all files, pre-parsing XML files etc (@ 45s)
- Best case build time - **15m30s (SSD, 2x Xeon 2687w, 32GB RAM)**
  - ~1GB/min
  - Cached data already available locally
  - CPU limited, not very parallel (avg 3 LP busy per target platform)
- Clean system build time – **25 min**
  - Pulls down all data from network
  - ~500MB/min
- Room for improvement! (more async work, more parallelism)

Asset Database (Celsius)

- Data managed in Avalanche Storage Service
- Similar implementation to build store
- I.e Log Structured
- Produced by a mapping process ”importing” data into the database
  - Very much like the regular data build process!
  - Data may be imported from native format files
  - ... or other data sources (SQL, Excel, whatever)
- Saving involves ”exporting” database assets back to files
  - I.e a reverse mapping

Celsius - Benefits

- No need to save to disk (or check out) before build
- Snapshot isolation for builds
- Cheap branching for creating multiple sessions
  - I.e preview same level/object/shader side-by-side, different settings
- Tightly integrated with build system
- Fast sync
  - Seconds to get up and running
  - Lazy fetch
- … tons more

Q & A

Slides will be available shortly:

- http://www.bionicbeagle.com
- http://publications.dice.se