



Asset Build Management

Bob Sprentall
February 7th, 2012

Terminology

Asset File	JSON that describes an asset .model, .texture
Source File	Any file that is referenced by an asset .tga, .dae
Target File	Built from an asset using a specific set of build parameters
Dependency	Any file that is read by a builder An asset, source, target, application, etc.
Fingerprint	Value that build system uses to determine if a file has changed Timestamp or hash

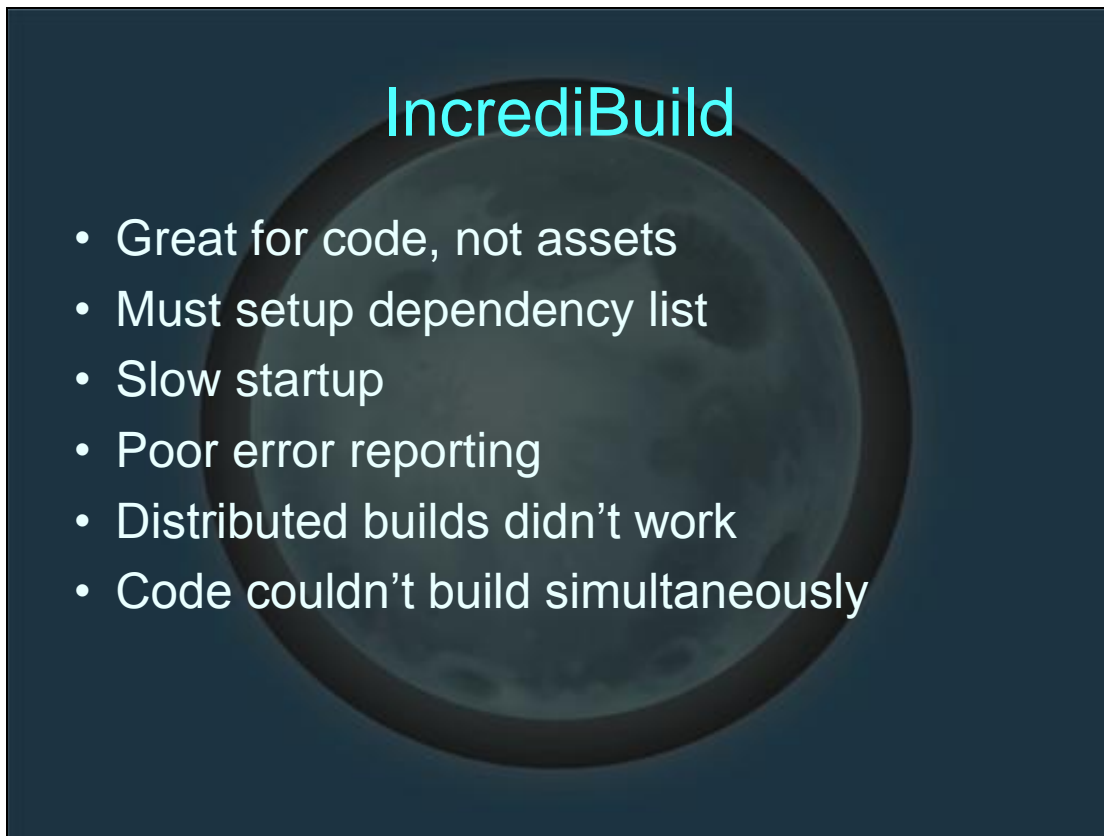
Approaches To Building Assets

- Export directly to engine format
- Button/command to build an asset
- Script to build a zone or set of assets
 - Like code...

Direct to engine is ok for a small project, but scales poorly. Version changes become difficult. Okay for prototyping, but usually still need a build pass because there is built data dependent on other built data.

Building specific assets is tedious and error prone. May not realize that things are out of date.

Build scripts take time to gather a list of things to build.

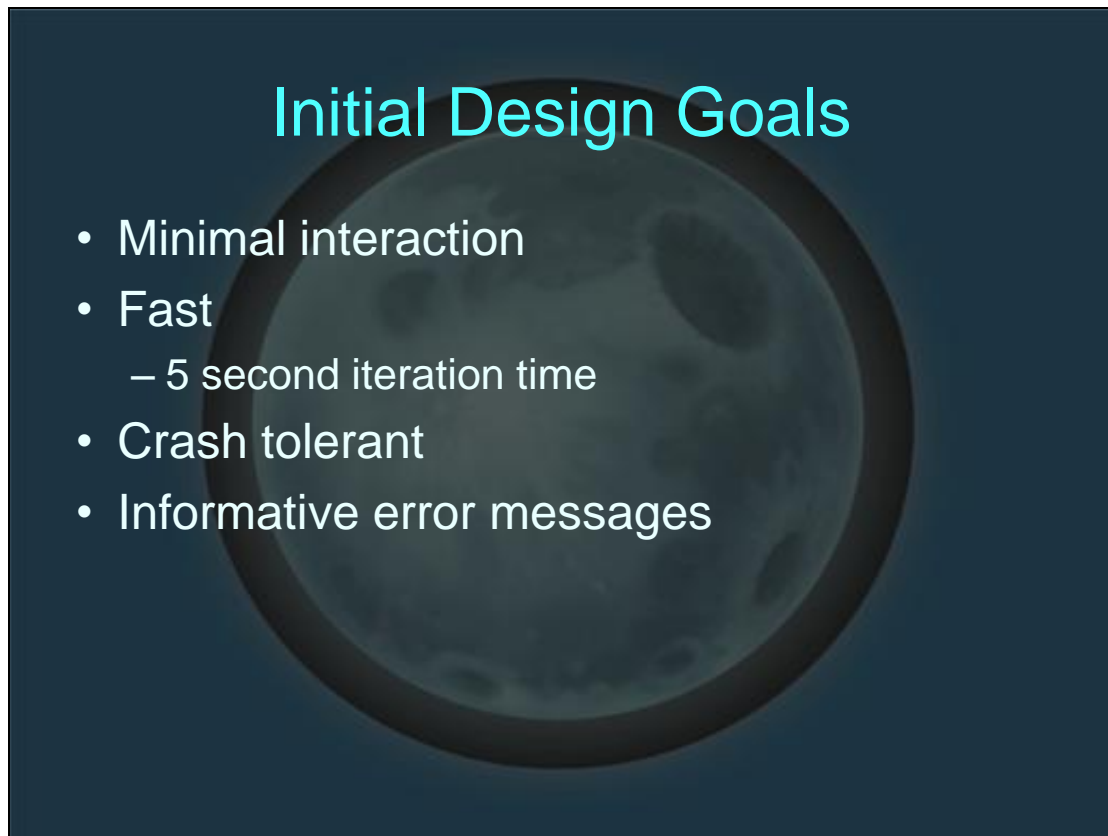


Required a dependency list to be maintained. As builders changed they would have to update the dependency list generation code.

Took several seconds before it started building with a relatively small set of assets. Seemed like it wouldn't scale well.

Hard to know something failed to build. Harder to know why.

Using 64-bit builder executables apparently prevented distributed builds from working.



System should know when assets need to be built and do that automatically.

Goal of 5 seconds between when an asset is changed and when it updates in the game.

Builders and tools crash. The build manager should recover from such crashes and keep working.

Information about builds that is relevant to programmers and content creators is different. All information should be easily accessible.

Initial Design

- Tracker watches for file changes on disk
 - Using USN journal
- Check each asset file
 - Required too many DB queries
- Build or download each out of date target
- Update target dependencies
- Continue watching for changes

Run continuously so assets would build as soon as something changed without needing to start the build process.

File systems like NTFS support a log which lists what files have changed.

Scan all assets and determine which needed to be built.

When building, get files from a central cache server if the dependencies match what they were when the version in the cache was created.

If building locally, upload the built file to the cache with information about the dependencies.

Revised Design

- Scan to find asset files
- Create a list of targets
- Tracker watches for file changes on disk
- Mark dependent targets when a change occurs
- Build or download each out of date target
- Update target dependencies
- Continue watching for changes

The asset root is scanned to find source files.

For each source file, there are typically multiple targets. This list of target includes information on how to build each one.

When a file changes, targets that depend on that file are marked out of date meaning they must be built.

An external process is used to perform building because builders have a tendency to crash. Don't want build system to go down when that happens.

When a target is built, the builder reports dependencies to the build manager which stores this list which is later referenced when a file changes.

Decreasing Build Time

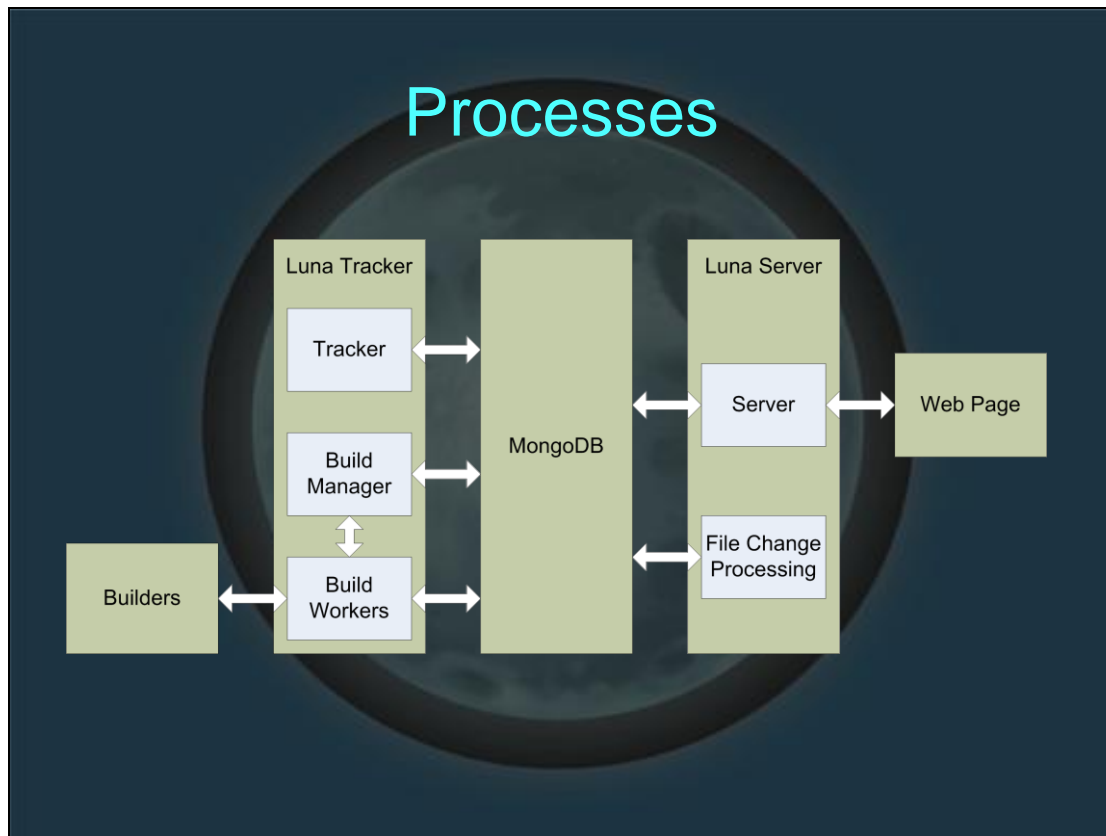
- Run multiple builds at once
 - One manager thread queries DB
 - Configurable number of worker threads
 - Asynchronous with tracker
- Multithread tracker
 - One thread parsing journal
 - Fingerprints generated in separate thread

Manager thread queries DB to find something to build. It sends information about the target to a builder thread which kicks off the build process.

System Considerations

- Everything crashes
 - Use separate processes
- Accessible through web page
 - Web page generates content from DB
- Disk contention
 - Throttle building to compensate
- Auto dependency generation

When the tracker is behind, high priority targets don't get built as quickly. Throttling builders lets the tracker keep up with changes and high priority targets get built first.



Implementation Details

- Metadata contains dependencies reported by builder
- List of targets kept in database with target metadata
- Use MongoDB

Metadata contains dependencies and other information used by the build system. It is kept in a database for rapid access.

Chose MongoDB because it uses JSON which fits well with our existing architecture. Want to use the database for other things so it is nice to have a common DB interface. MongoDB also has useful features for profiling and debugging.

Metadata

- Contains each dependency, its fingerprints, builder version, other stuff
- Need to transfer metadata with target file
- Kept in database for fast access
- Handy to have metadata on disk when database is rebuilt
- Stored in an alternate data stream in the target file

Log is separate from metadata because metadata gets read into the database while the log doesn't belong there.

The database gets rebuilt when the build manager loses its place in the change journal or the database becomes corrupt. This happens alarmingly often, so it is good to have metadata on disk which the build manager can use to determine that what files need to be built.

Storing the metadata in an alternate data stream means that whenever it is copied from one NTFS volume to another such as to or from the cache server, the metadata automatically moves with the target. There is no extra work the engine has to do to get at the built data. However, future file systems may not support alternate data streams.

Fingerprints

- Identify a file at a point in time
 - Timestamp or file hash
- Generated by the tracker when a file changes
- Stored in database
- May not match file, but eventually will
- Generated by builders when they first access a file
 - Can't take fingerprints after a build is complete because they may have changed

Timestamps work well for source assets, but require a special Perforce flag to be set. File hashing works well for files that are not synced between computers like intermediate and target files. Hashing also helps keep accidental timestamp changes from causing asset rebuilds. This was a problem due to integrations changing a file's timestamp even though nothing had changed within the file.


Build Caching

- Fingerprint each dependency
- Hash fingerprints to form a signature
- Look on server for matching signature
- Get latest built file if dependencies are unknown

Smarter Builds

- Intermediate files to avoid redundant work
 - Usually a PID file. Transferred through cache.
- Disable platforms you don't care about
- Priority builds
 - Build what is open in the editor first
- More precise target versioning

Priority Builds

- Used by clicking on 
- Recently released second version
 - New version worse for cache population, but works much better
- Tricky because references are needed
- Multiple thumbnail builds used to approximate references
- Priority propagates to dependencies

First version sort of worked, but not very cleanly. It had a queue of high priority targets that needed to be built which was separate from the main collection of build targets. It was a lot easier to mark targets in the main collection as high priority and find those first, but this was a tradeoff because build order can no longer be randomized which had helped with distributing build load among computers using the cache server.

Target Versioning

- Target version used to determine if a file can be loaded or needs to be built
- Only needs to change when data format is not compatible
- Added subversion
 - Used by build manager, ignored by engine
 - Good when a builder bug is fixed or new features supported

Noticed that people were changing the target version a lot. This meant when a change was released, assets had to be built before they could be loaded in the engine.

Looked at why versions were changing and often it was to support new features or fix targets that previously failed to build.

The subversion provides a way to force assets of a certain type to be rebuilt, but older versions can still be loaded by the engine

Lessons

- Version EVERYTHING
 - Built format, builder, build rules, external metadata, internal metadata, exporters, tracker database, database version
- Explicit build rules
- Keep history of builds in log
- Building quickly isn't enough
 - Need simple interface, control system load

Versioning is critical so when a data format changes, you can throw out old data. It's also important to check that processes are running the version you expect. There was a problem that came up a few times where the build manager was up to date, but the builders weren't so the build manager thought it was building assets at a more recent version than it really was. This was fixed by reporting the version along with file dependencies from the builder.

Keeping a detailed history of what caused a build and the results of that build has been very helpful in diagnosing issues. Especially helpful in determining the cause of problems like build loops.

When comparing building systems, people often look at the speed difference. This isn't enough. Build systems need to prioritize build duties and build things that are important first. They also need to help developers quickly figure out what to do to fix build problems.

Future Goals

- Error messages with suggestions
- Currently only track directories in branch
- Dependencies are not reported if the builder crashes
- Database contention and speed
- Managing load on system
- Continuous integration
- Open to suggestions

Every update to the database locks the entire database. Some tweaking for speed has been done, but there is still room for improvement. Setting hundreds of things out of date, which can happen when a single dependency changes can take a couple seconds.