GAME DEVELOPMENT TOOLS

MARWAN Y. ANSARI
EDITOR

CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

Game Development Tools

Game Development Tools

Marwan Y. Ansari, Editor

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

Cover Image: "The Iron Worker and King Solomon," engraving by John Sartain after the painting by Christian Schussele, courtesy of the Scottish Rite Masonic Museum and Library.

The image depicts the story of an ancient tool maker. According to Jewish legend recorded with the painting:

When the temple of Jerusalem was completed King Solomon gave a feast to the artificers employed in its construction. On unveiling the throne it was found that a smith had usurped a seat of honor on the right of the king's place, not yet awarded, whereupon the people clamored and the guard rushed to cut him down.

"Hold! Let him speak," commanded Solomon.

"Thou hast, O King, invited all craftsmen but me, yet how could these builders raise the temple without the tools I have fashioned?"

"True," decreed Solomon, "The seat is his of right. All honor to the iron worker."

Like the vital but underappreciated iron worker whose tools made possible the building of the temple at Jerusalem, creators of game development tools may too often go unnoticed but their tools are the essential prerequisites to all that is possible in computer games.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20130822

International Standard Book Number-13: 978-1-4398-6772-3 (eBook - PDF)

For my sons,

Mason and M.J.

The difficult we do right away; the impossible takes a little longer

Contents

Preface

Many might expect the preface of this book to try to convince the prospective reader of the importance of tool creation and design in the hopes of making a sale. Although a sale would be nice, I think the fact that you are reading the preface in contemplation of a purchase (or even after the purchase) shows that you already know the value of tools in games and software development. So, rather than preaching the value of the topic, why don't I spend these few pages the way Jim Blinn suggested, writing a preface that serves as a link between the author's mind and the reader's wallet in an effort to convince the reader not of the premise of the book, but rather that it will be a valuable addition to their library.

Originally, I wanted each article to describe something that the reader would be able to build. Now, however, it's obvious that was flawed thinking. A number of contributors submitted proposals on third-party tools (such as Genetica, Mudd Box, and FBX) that were clearly valuable because they give the reader another insight into how these tools may be used in their development pipelines.

Not only are there articles on third-party tools, I also received many articles on philosophical topics such as managing complexity, planning game streaming, and continuous integration. Due to the quality of these proposals, it became clear that the book would become more than just a set of recipes for do-it-yourself tools.

Many articles have relevant and valuable insight to aid in decision making and planning your asset pipeline such as "Workflow Improvement via Automatic Asset Tracking" (Chapter 3) and "Real Time Tool Communication" (Chapter 6).

Just as important is the sampling of third-party tools where developers of Intel's Threading Building Blocks (TBB) library discuss "Optimizing a Task-Based Game Engine" (Chapter 17) and Autodesk provides an article on vector displacement maps and using FBX (Chapter 16).

In the area of do-it-yourself tools, we have an equally impressive lineup. The "Low Coupling Command System" (Chapter 13) is explained as well as a method for "Improving Remote Perforce Usage" (Chapter 15). Our authors also contributed in-depth articles on "Real-Time Constructive Solid Geometry" (Chapter 8) and "Shape-Preserving Terrain Decimation and Associated Tools" (Chapter 10). Virtually all games programmers have heard of or already use COLLADA. Even though the wide-range implementations can make it difficult to write robust code, "A COLLADA Toolbox" (Chapter 9) will help make your code more bulletproof.

You will find the book is divided into three parts. Out of the gate, we have philosophical articles in the first part, followed by the do-it-yourself tools in the second part, and finally, third-party tools in the last part. Ultimately, time will tell if this is as useful as I hope it will be.

My choice in the cover art was made for several reasons. Originally I was going to pick a screen shot from one of the tools covered in the book, but after some reflection, I remembered seeing a painting at a Masonic temple where my lodge used to meet in Riverside, Illinois. Right away, it struck me as the perfect cover art.

Essentially, the moral of the painting is about recognizing the labor of the tool maker. The story goes that at the completion of King Solomon's Temple, a great feast was held in honor of the craftsmen and artisans who accomplished the great undertaking. Having sat in the honored location at the right of King Solomon, the ironworker earned the ire of the craftsmen and artisans. After his breach of protocol was brought to the king's attention, the ironworker was asked to account for his taking the seat of honor before the king could invite someone to take it. Not discounting the work of the craftsmen and artisans, the ironworker explained that without his tools, they would not have been able to build such as magnificent structure. Acknowledging the ironworker's point, King Solomon let him stay in the seat of honor.

Now, I did not start my career in the hopes of being a tool maker. You might agree with me that very few of us in the field of video games do. Tools still need to be made though, and we can't all be engine programmers, shader writers, and artisans. Having been a driver developer, a demo developer, a shader writer, a physics programmer, a video game developer, and even a data base developer (long ago!), what I have noticed is that every company needs a tools developer. Invariably, though, the tools developers always feel (and actually are, IMHO) that they are undervalued.

Not many people can get excited about a new 3D Studio Max exporter or a nice library that handles exceptions well. Given the need for the role, and being undervalued, I thought the painting of "King Solomon and the Iron Worker" a perfect choice for this book.

Every large undertaking, such as this, is done with the help and support of many friends and family. Listing each person would, of course, be prohibitive and would probably take up half the book. Sadly, I'm sure that I'll leave someone out, and I hope they don't take it personally. Everybody forgets something, right?

First and foremost, I would like to thank my family for their support in this process. I am lucky to have been given the time to organize this work when I probably should have been doing things like cutting the grass or washing the dishes. Thank you Jackie, Mason, and M.J.

The staff at A K Peters was invaluable during this process. Going from contributor to editor is a bit daunting, and Alice Peters' and Sarah Cutler's faith and support was incredible. Selecting the articles was no small feat. Drew Card, Dave

Gosselin, and Jon Greenberg kept me from cutting some articles that turned out to be about a thousand times better than I would have foreseen on my own: thanks fellas. Of course, the authors of this book are not to be forgotten. Each of them gave up personal time as well as time with their friends and families to help. I must give a special thank you to Sebastien Noury, Jaewon Jung, and Amir Ebrahimi for helping me proofread some of the drafts. Also, thank you to Craig Barr and Mark Davies for spending the time talking out some fine points of their articles and working particularly diligently to ensure that their articles were as forthcoming as possible. A personal thank you to Victor Lerias and Mike Irby, colleagues of mine, whose levity during a work crunch (which interfered with this project) made everything a little less stressful. A thank you cannot be forgotten for the Scottish Rite Masonic Museum and Library and especially for Maureen Harper for helping us get the rights for the artwork on the front cover.

Also, I'd like to just list a few folks who have helped me in little or big ways over the years but I simply don't have enough space to include the reasons: R.W. Br. Raymond J. Babinski, R.W. Br. Vytoutas V. Paukstys, the members of Azure Lodge 1153, Western Springs, Illinois, A.F. and A.M., Eric Haines, Dr. Roselle Wolfe, Dr. Henry Harr, Mr. Dan Keibles, Mr. James Deacy, Narayan Nayer, Maher and Marty Ansari, and C. Scott Garay (last but never least).

Finally, a special thank you to Wolfgang Engel. Wolfgang first got me started publishing articles in *Shader* X^2 and has kept in touch ever since. His high standard of excellence in the ShaderX and GPU Pro series is something that everyone has come to expect and something that all books of that nature now strive for. Thanks Wolfgang.

—Marwan Y. Ansari

Part I
Philosophy and Methodology

It's never easy trying to decide which path to take when starting a large project. Should you write your own exporter? Should you use something off the shelf? Each has its own set of advantages and drawbacks, but unless you have gone down that road before, it's often difficult to know what the cost of the pitfalls really are.

This part offers some guidance into how you might structure various parts of your next large project or how you might begin to approach refactoring some areas of your current code base.

To name just a few, we start with Chapter 1: "Taming the Beast: Managing Complexity in Game Build Pipelines" to discuss different approaches that will be useful in various aspects of your next or current game. Chapter 3 discusses "Workflow Improvement via Automatic Asset Tracking." However, general strategies can be applied to lower-level functionality as we see in Chapter 7: "Robust File I/O."

1

Taming the Beast: Managing Complexity in Game Build Pipelines

Fernando Navarro

1.1 Introduction

It is not a secret. Game companies face fierce competition to attract gamer's attention. Every holiday season, many products compete for the honor of being played. As a consequence, games need to be bigger, nicer, even funnier! Their plots need to be deeper and longer. Game scripts use many more levels, scenarios, and quests. Engines struggle to squeeze all the computing power to render highly detailed textures, models, and animations. Playing online and using downloaded content is also a must. In short, games have become awfully complex.

From a technical point of view, each title requires more assets and increasingly more complex relations among them. For artists to raise the quality level, they need to rely on tools that allow quick iteration. Releasing a multiplatform game is also the norm for many publishers.

Every aspect of the current generation of games proves more challenging for the production pipeline. Traditional designs are no longer capable of handling such pressure, and these not-so-old models do not scale well. New approaches are required.

The importance of new solutions is so obvious that terms such as *content management* or *asset pipeline* have become frequent guests in the agenda of many management meetings.

With this in mind, we are going to discuss different approaches that will be helpful during the design, implementation, and refactoring of content processing and asset build pipelines. Even if these notes do not represent a specific implementation, they describe generic methods that can be used to reduce downtime and improve efficiency. Many of them are orthogonal and can be implemented independently without requiring a full revamp of the system. These guidelines are a combination of common sense tips, answers to the evaluation of practical "what if...?" scenarios and information scattered across the Internet.

1.2 The Asset Build Pipeline

The term *asset build pipeline* can be found in many wordy flavors: content pipeline, asset build pipeline, build pipeline, pipeline or simply build. Under this concept, each company can fit a radically different implementation of a system whose main duty is transforming assets. In order to give a clear overview of what it represents, we will briefly describe its contents and its (sometimes) blurry limits.

1.2.1 What Is Included and What Is Not

As far as this chapter is concerned, we will consider the asset build pipeline as covering any processes designed to transform raw assets as produced by the *digital content creation tools* or DCCs (3D modeling or animation packages, image painting software, audio editing suites, in-house editors, ...) to the files that can be loaded by the *game* in a fully cooked or temporary form. In their simplest form, the associated processes are executed at each user's workstation and are the main method to push content from the DCC into the game. Figure 1.1 shows the location of the system as part of the global set of production tools and how the system connects to each one.

In general terms, the asset build is a framework that allows the execution of generic transformation tasks. Each square node in Figure 1.2 represents a single step that massages a set of inputs into one or many output files. This conversion is the result of executing a compiler, a script, or a tool that transforms data so it can

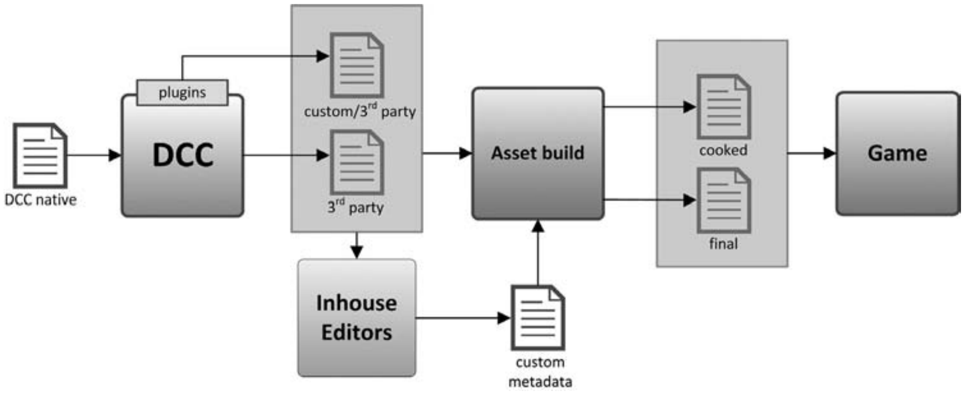

Figure 1.1. Block diagram of a content production pipeline, showing the tools and exchange formats involved in the production of game assets.

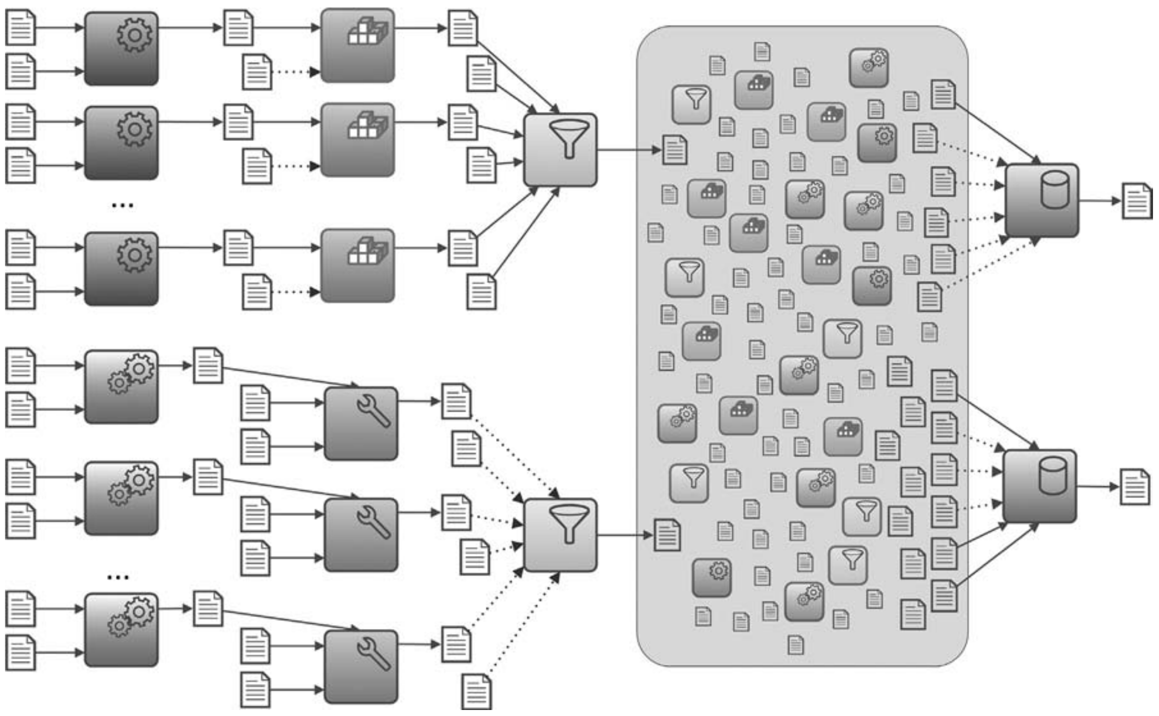

Figure 1.2. Diagram showing an imaginary dependency graph. Compilers are displayed as rounded boxes, with a different color representing alternative transformation steps. Dependencies connect compilers, input files, and output files. Explicit and implicit dependencies are drawn as solid and dotted lines respectively. (See Color Plate I.)

be consumed by the game. Each input is modeled as a *dependency* for the node, the node becomes a direct dependency for each of the outputs, and the outputs themselves are dependencies for later processing steps. In its minimal implementation, the system will be composed of a method to extract the dependencies, so a set of tasks can be *scheduled* and *executed*.

Our focus will be on the framework itself, not on the details of each individual compiler. Each compiler can integrate complex conversions involving geometry, image and sound processing, database accesses, etc. which can also probably be implemented using alternative methods. Together with custom editors and tools, they deserve an independent discussion and will not be covered in this chapter.

We have implicitly assumed that the target of a build is the production game assets. From a broader point of view, other professional environments use conceptually similar frameworks that are wired to compile source code, render and postprocess CG images, process natural language, crunch physics simulations, or untangle the mysteries of DNA sequences [Xoreax Advanced Grid Solutions 01, Sun Microsystems 11, Pande lab Stanford University 11]. These frameworks can also benefit from what is explained in the following sections.

1.2.2 Features and Design Decisions

In this section, we list the requirements for a hypothetically ideal system. We also group the requirements according to degrees of desirability. Most of them are purposely open and, in some cases, vague. You as a designer need to find what each of them means depending on your particular environment and the characteristics of your project.

As a common-sense rule, you should clearly determine the constraints, complexity, and targets of your system. A careful study will help in answering the single most important question you need to ask yourself: *Do I really need to implement my own solution, or can I use an off-the-shelf product like XNA Build, SCons, Makefiles, Jam, . . . ?* Even if the vast majority of projects require some sort of build pipeline, adapting an existing package can be a wise decision that may eventually save your company a significant amount of development and support time.

Your project will also impose alternative requirements and priorities. The following list tries to cover a wide set of situations. Even if some of them are just guidelines, they have to be seriously considered, as they are the source of project failures and frequent design pitfalls. As general as they are, use them as a starting point after which you will probably want to consider more specialized features that are not included here.

Required features of a build.

- Given a set of input and target files, the build must be able to automatically determine what files need to be processed and in which order.
- It must be able to handle the amount and complexity of transformations required to process each file.
- It must be able to detect processing errors and report them accordingly.
- It must be able to handle a variety of sources of data. A generic system needs to be agnostic of the contents of the processed files.
- It must be able to produce assets for a number of target platforms.
- It must finish within a reasonable time frame and always within the constraints defined by the project.
- Finally, a key factor for its success: It must provide a smooth user experience. Even if internally complex, an effort needs to be made to make it look simple, clear, fast, or at least to keep users informed.

Desirable features of a build.

- Provide granularity and allow executions on reduced sets of assets.
- Allow quick iteration for the most common asset types or at least provide alternative tools for them.
- Signal any issues as early as possible. Any user should be able to understand any errors, warnings, and important messages.
- Handle broken or missing data. On any issues, a partial build may be a valid result. A failed build should be the last option.
- Design it to be resilient to machine failure. Hundreds of users may be relying on the service, so on server failure, it must continue working even if it is in a degraded mode.
- Implement fine file control. Avoid *master files* that cannot be simultaneously edited or automatically merged. They will become a bottleneck.
- Design it to be extensible and scalable. Games are dynamic environments and evolve over time. Expect new requirements at the final stages of the project.
- Use conceptually simple models. It will allow more efficient execution, debuging, and maintenance. Remember that complexity has a direct impact on development costs. Downtime and support also determine the final budget of a project.
- Do not reinvent the wheel. Use existing libraries, reuse modules, apply design patterns and follow known good practices.

Optional features of a build.

- Support changes in the file formats of input and output files. Otherwise, full rebuilds may be required.
- Reduce the overhead of the system. Allow fast starts and retries. Null builds should be quick.
- Offload computation. Large sets of similar assets and complex or slow transformations can be processed by *number crunching machines* or *computing farms.*
- Focus on efficiency. Rely on advanced techniques such as caching, file reusing, and proxying.
- Provide methods to prevent pollution from broken or malfunctioning hosts and tools to clean and purge caches.

- Split the pipeline into several independent steps. Frequently, dependency extraction can start and end without executing any compilation steps. A task scheduler can be fed based on an execution plan built from the dependencies. Packaging of development versions, daily builds, and release candidates may be postprocesses.

1.3 Dependencies

Automatic asset building can be thought of as the execution of a series of steps determined by a complex recipe. A given step can be safely executed only after all its requirements, or dependencies, are satisfied. On the other hand, a node needs to be reevaluated only when its direct dependencies or the dependencies of their direct dependencies differ in a significant way. Knowing that, the order in which the tasks are executed will be fully determined by the overall set of dependencies.

1.4 How to Determine Dependencies

Establishing the correct build dependencies is a conceptually simple process: it requires the construction of a directed acyclic graph (DAG) resulting from the aggregation of each node's dependencies. Frequently, these dependencies are modeled at the file level, but finer grained approaches can also be used. In the later case, dependencies are extracted from the files themselves.

We will assume each processing step is deterministic; that is, it always produces the same output for a given set of inputs. Even if this is not a requirement, advanced techniques such as caching and proxing can be greatly simplified. Therefore, any processing that relies on random number generation, timestamps, IDs, and references to other objects will need to be carefully designed.

There are two basic methods that can be used to determine the full set of dependencies.

Hardcoded dependencies. This is the most primitive approach, as dependencies and order of execution are hardcoded into the scripts. As such, each new entity requires an update of the system. The asset build simply executes every known step in the order determined by static dependency data. Because files are compiled assuming every dependency is ready, this approach restricts flexibility. Missing inputs due to user error may be difficult to avoid and debug. Smart scheduling is also out of the capabilities of the framework. This method is the equivalent to batch and shell scripts or really simple makefiles. It can only be used in simple scenarios where dependencies are described at file level only.

Automatic extraction. Each compilation step is represented as a templated description of the inputs that are required, the outputs that are produced, and the parameters used in the execution of the compiler. The full set of inputs and the templates are used to generate a full graph of explicit dependencies. These dependencies are gathered without any knowledge of the contents of the source files by applying the recipe that matches a given set of input and output file types.

The build will rely on dynamic dependency extraction when the dependencies cannot be known in advance, they are stored inside a file, or this file is produced during the build process itself. In these cases, a scanner will read, parse and extract the relevant implicit dependencies to be added to the overall set. This means that areas of the DAG are not known until the build has started and the corresponding scanners are triggered. Implicit dependency scanning has been successfully used in systems such as SCons and Waf [Knight 11, Nagy 10].

In general, everything that can possibly modify the result of a compilation step needs to be tracked as a dependency. As such, it may be desirable to track the version, compiler command line, and execution environment.

1.5 How to Use Dependencies to Your Benefit

1.5.1 Determine Dependencies as Early as Possible

There are many advantages to establishing dependencies as early as possible in the pipeline. Knowing the dependencies even before the asset build is invoked can potentially reduce the complexity of the system and the size of any transient data. The cost of extracting asset dependencies in the DCC is small and will be amortized by the amount of times they are used. Scanner execution is minimized, and larger areas of the graph can be fully determined in advance. This makes the system more predictable, increases system stability, and reduces processing time. Both early data validation and smart scheduling become more feasible. This same data can be used by DCC plugins to track references and allow the production team to forsee the implications of changing a given asset.

As desirable as this is, early determination can only extract the dependencies from the assets that are directly processed inside the DCCs. Figure 1.2 represents the nodes without input dependencies on the left side of the graph. In most scenarios, static determination can define large areas of the DAG but needs to be complemented with alternative methods.

1.5.2 Dependency Granularity: Coarse or Fine Level

In many situations, a file can be considered an atomic entity that provides a high enough level of granularity. In some other cases, when, for example, hundreds of compiler instances share the same file as an input, finer dependencies can be an interesting option. By using file-specific scanners, the graph can be populated with detailed information. In case the original file is updated, only the tasks in the path whose refined dependencies have changed will need to be reevaluated. This may imply a reduction of orders of magnitude in the number of tasks compared to tasks using coarser dependencies, just by avoiding redundant compiler executions.

1.5.3 Out-of-Order Evaluation

The DAG contains patterns that provide interesting advantages. Certain asset types, mainly those located on the left of Figure 1.2, are usually numerous, independent of any other assets, and their compilation steps are similar. Textures, meshes, skeletons and skinning data, animations, prerendered videos, audio and speech files are examples of these assets.

By precompiling these files, complete areas of the graph can be evaluated offline, so the processed files are available even before an instance of the asset build is started. Since no interdependencies exists, these tasks can be easily distributed and calculated in parallel. For this to be effective, any output files need to be stored and reused using caching techniques (see Section 1.6.1). Existing packages can reduce the complexity of the implementation [Xoreax Advanced Grid Solutions 01, Electric Cloud, Inc. 11, Sun Microsystems 11].

Continuous integration techniques that have traditionally been employed with source code [Duvall et al. 07] can also be used with certain asset types. Changelists added to a versioning system can be monitored for assets that can potentially be built and cached. As with the original approach, a farm of servers may be responsible for the execution of the corresponding tasks.

The need for this approach and the associated complexity has to be considered in light of the characteristics of each particular scenario, but in general they can be easily implemented once a caching strategy is in place.

1.5.4 Limiting the Scope of the Build

In the same way that certain asset types can be precompiled, we can also reduce the extension of the dependency graph by ignoring specific asset types. Builds focused on providing quick iteration may not require the generation of a fresh copy of every single output file, even if their dependencies say otherwise. For example, audio files can be safely ignored when the intention is testing meshes and textures. In other cases, the build can be reduced following gameplay-related divisions: levels, quests, cutscenes, and global assets are frequently self-contained and independent.

For this to work, with the dependency graph already determined, every node that will produce a file that can be ignored is tagged as inactive. Any dependencies that lead to these nodes can be safely disabled, given that they are not required by any other active nodes. After the DAG has been processed, only nodes that are still active are scheduled for execution.

In some other situations, it is interesting to consider regions of the graph as a whole. Any dependencies entering the selected area can be considered inputs for an alternative node representing the region. The same occurs with outputs and the dependencies that originate inside and crossing this area's borders. If none of the inputs have changed, the area can be safely tagged as fully processed.

Knowing that the number of dependencies can be as large as several million, pruning areas can provide big savings both in the size of the data sets and the number of tasks that will ultimately be executed. The advantages will become evident after knowing the details of the proxying techniques of Section 1.6.3.

1.5.5 Dump the Dependency Graph

Once the dependency graph has been built, dump it to a file! Later builds can read it and use the contents as a starting point. An updated DAG can be built at a reduced cost, allowing faster start-up times. Moreover, many implicit dependencies will be ready, with a considerable reduction of the number of scanner executions. The chances of performing conceptually simple but technically difficult tasks, such as giving an estimate of a build time, are also improved. If none of the task templates have been modified, any subsequent builds will only need to regenerate the areas of the graph whose inputs have changed. "What has changed" can be redefined based on the expectations of the current build: a quick, incremental build focused on a single level, a full build, replacement of a few objects, etc.

Another important consequence is the fact that graph generation and task scheduling are now decoupled. This provides interesting opportunities: Tool updates become more localized, and the determination and evaluation of tasks become independent. It is even possible to use alternative schedulers optimized for quick turnaround, make use of heavy parallelism, or rely on the facilities of certain computing hosts, or use simple heuristics such as first-come, first-serve, etc.

Finally, the information contained in the dependency graph can be invaluable for finding and studying bottlenecks, locating areas for improvement, and fine-tuning the system. Understanding how such a complex system is performing may be a daunting task in the absence of execution data and clear logs. Simple formats such as plain text, xml, .doc, or GraphML and tools like Microsoft Excel and languages supporting xquery or even the command grep can be invaluable.

1.6 Advanced Techniques

1.6.1 Caching

This method focuses on file reuse and is heavily inspired by the equivalent systems for source-code compilation. A successful example is the CCache package [Tridgell 11].

With a caching scheme, the outputs of compilation steps are stored in a repository. During the evaluation of each node, the scheduler checks for the existence of a cached result that corresponds to the given set of input nodes. In the event of a cache hit, the result can be efficiently retrieved. Cache misses fall back to a standard compilation that includes uploading the results to the cache server. It is up to the designer to allow the method of populating the server from users' workstations and dedicated machines. In general, any result is acceptable, independent of its origin.

Establishing how the set of input dependencies are determined, stored, and queried is a fundamental decision. Simple methods such as filenames and timestamps may be inaccurate as they do not fully represent the contents of the files. Hashes, signatures, and file digests are popular alternatives. They can be calculated using different variations of the CRC, MD5, or SHA algorithms. In general, any method capable of converting a variable-length stream into a fixed-length key is valid. For our practical needs, the methods need to generate keys with low collision probability; that is, for two different files, the probability of generating the same key is really low.

Hashes can also be combined together, so several dependencies can be represented by a single key. For a combined hash, a successful search in the database will retrieve a file that has been compiled from all the inputs whose hashes were merged.

Cached results can be stored in many ways. Two of the most common methods are versioning systems and dedicated servers:

Versioning systems. Compiled results are stored in a versioning system (Perforce, Alienbrain, CVS, SVN, Git). Cache population and cache hits are, in fact, repository check-ins and retrievals.

This approach works well in those cases where each file is compiled with a single set of inputs and where the most recent version is generally usable by any build. Every source and its compiled results can be checked in at the same time. Syncing to the latest version of the repository will mostly retrieve files that do not need further recompilation. This approach is not so flexible when files are processed with variable compiler flags and dependencies. In these cases, the versioning system needs to be complemented with an external database capable of linking each hash with a given version.

Dedicated servers. The use of dedicated servers represents the most flexible approach. A cache server can integrate sophisticated algorithms, but in its most basic configuration it can be built from a database and a file server. In some circumstances the hash management system is accessed through an ad hoc interface that implements advanced querying features. They will be described in the following sections.

Not to be overlooked: These machines must be able to support high disk/network loads as well as to efficiently handle concurrent uploads and downloads. In general, file accesses will range from a few bytes to several gigabytes. However, the distribution is usually biased to files containing metadata and art assets due to these files being more numerous and having reduced variability in their input dependencies.

As pointed out before, with the server being a central resource, in the case of failure the build needs to rely on alternative servers or be able to work in a degraded mode.

1.6.2 Results That Are Close Enough

In some cases, it is desirable to use compiled files, cached or not, even if they have not been generated from exactly the same set of dependencies. For certain types of nodes, and assuming the game code supports it, a result that has been generated from the closest set of inputs may be used without significant differences in the game experience. This option is especially interesting with files that are generated after long compilation processes. Examples of this are precomputed lighting, baked ambient occlusion, collision meshes, and navigation data. The advantages are clear compared to the standard approach, where a change in a single mesh triggers a complete reevaluation.

Without loss of generality, file hashes can be calculated using smart methods that ignore the parts of the file that cannot modify the compiled result. Good candidates are comments, annotations, and object and material names that are not externally referenced.

1.6.3 Proxies

This second technique tries to avoid unnecessary work, namely, compilations and cache transfers, by performing lightweight evaluations of regions of the DAG. While similar in nature to the optimizations considered in Section 1.5, proxies are designed so the build can operate using file hashes instead of the original files.

The relationship between a file and its hash can be exploited in many different ways:

- During dependency generation and early scheduling, a file can be fully represented by its hash. The file needs to be transfered to the client's hard disk in just a few situations. For example, cache retrievals can be delayed until we have clearly determined that the file is going to be read by a compiler or a scanner.

- The hashes of the outputs of a compilation can be stored and retrieved from a cache server. This allows the dependency graph to be updated without requiring the execution of any compilers.

- Implicit dependencies can also be cached, so the evaluation of certain scanners may not be needed.

A build can avoid unnecessary compilations by simply determining the hashes of inputs and outputs and using them to update the dependency graph. Cascading this process can complete large regions of the DAG. In the best cases, a build may replace several compiler executions by repeated hash retrievals, completely bypassing any intermediate results that are needed to produce the final files.

1.7 Minimizing the Impact of Build Failures

In a production environment, it is as important to implement the right technology as it is to ensure a seamless operation of the system. Build systems are operated as part of organic environments where new content is delivered in increasingly tighter deadlines. The evolution of every project will impose a relaxation of the assumptions that were accepted during the initial design process. All of this pushes against the stability of the system and may imply frequent system updates.

As worrying as it looks, a progressive degradation of the system is to be expected. On the brighter side, there are simple approaches that can improve the chances of success. These methods, without tackling the source of the problems, will surely raise user satisfaction.

1.7.1 Exploit the Difference between an Error and a Warning

Most content creators are interested in propping and previewing their assets in game and are not so concerned about the latest version of *every* asset. With this in mind, the system's priority becomes completing partial builds that accurately represent views of a reduced set of assets. This is true even if, on the global picture, some other second priority areas may display artifacts.

Let's assume every system incidence is treated as an error and, as such, forces the build to stop. Applying simple statistics, if we consider that each asset is processed with a nonzero probability of failure, the chances of successfully completing a build are extremely low.

It becomes evident that a correct level of criticality needs to be assigned to each build message. For example, in the context of a geometry compiler, finding polygons with irregular shapes may be a reason to avoid generating the processed mesh. In the context of the whole build, this situation certainly has a less dramatic relevance.

Fatal errors should be the exception. In many cases, failed compilations can be logged as a warning, and their outputs replaced with placeholder assets. These fallback mechanisms do not fix the source of the problem, so additional maintenance will be added to the workloads of the support team. The issues may be solved via changes in the code and data, or, in some other cases, they will involve the artist updating the assets. Clearer messaging will increase the opportunities for the content creators to be able to identify and fix the problems by themselves.

In this context, multiple message categories need to be defined. Each information displayed will fall into one of them. A trivial approach that is commonly accepted classifies messages, in order of increasing importance, as *debug*, *verbose*, *information*, *warning*, *error*, and *fatal error*. Logs tagged using this simple method can also be easily filtered.

1.7.2 Early Data Validation and Reporting

Early validation can improve the user experience in many ways. First, it will quickly point out problems that otherwise would only be found after minutes or even hours of computation. It can also give a better idea of if the build will finish successfully or if fundamental pieces of data are missing or corrupted.

Efficient data validation may focus on metadata files: entity and hierarchical data, compilation and run-time information, and gameplay balancing among others. On the other hand, asset contents tend to be too specific for a generic system, and they represent big volumes of data, so it is not always suitable for early validation.

As such, metadata can be processed using simple tests aimed at finding duplicates and detecting missing references and assets, empty data records, inconsistent hierarchies, files with unexpected sizes, or malformed XML data.

Even if early validation is not a complete solution and does not cover all possible data paths, it can be complemented with asset data validation as part of the export process. The next section explains this approach.

1.7.3 Prevent Data Corruption

Data corruption is always a possibility in a environment where files are edited concurrently using tools that are under constant development. In the worst cases, corrupted data can produce a meltdown of the whole system and bring the team to a halt.

In order to limit the extent of the damage, local data validation must be performed before metadata and assets can be submitted to the versioning system. In some cases, the checks detailed in Section 1.7.2 will suffice. In other cases, a quick compilation at the time the asset is exported from the DCC will be beneficial. However, the ultimate proof of damage is a full build. For efficiency reasons, this approach is not always feasible. In all cases, every check-in must be properly labeled, and in the event of data corruption, user builds can be forced to use the latest data known to be safe.

Fixing corrupted data and eliminating build warnings are usually manual processes. They need to be performed by staff capable of checking the integrity of each file format and applying the corresponding updates. We encourage the use of human readable formats such as XML that allow simple editing, diffing, and merging operations.

It is also worth considering the fact that a defective tool may generate corrupted data that can pollute the caches. In these cases, any derived files need to be purged and replaced by recompiled assets. Cleaning operations are frequent, not only due to errors, but also as a maintenance operation. Removing the files compiled with a given version of a tool, uploaded from a certain client, or not referenced for a period of time are common duties in any system.

1.8 Conclusions

In this chapter, we have presented a series of approaches that can improve the chances of setting up an asset build pipeline that is both efficient and resilient to failure. These directions will prove useful for both new and existing frameworks.

As initially stated, these are a set of guidelines and checkpoints that can anticipate common pitfalls and design issues, the reader may be faced with when implementing a complex systems like this.

The never-ending search for the perfect AAA game will eventually challenge each of these ideas. However, we think this chapter offers some insight into different alternatives that build engineers can use to keep their systems up and running and enjoy a satisfied user base.

Bibliography

[Duvall et al. 07] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley, 2007.

[Electric Cloud, Inc. 11] Electric Cloud, Inc. "Electric Cloud Home." Available at http://www.electric-cloud.com/, 2011.

[Knight 11] Steven Knight. "SCons: A Software Construction Tool." Available at http://www.scons.org/, 2011.

[Nagy 10] Thomas Nagy. "Waf: The Flexible Build System." Availabkle at http://code.google.com/p/waf/, 2010.

[Pande lab Stanford University 11] Pande lab Stanford University. "Folding@Home." Available at http://folding.stanford.edu/, 2011.

[Sun Microsystems 11] Sun Microsystems. "Grid Engine." Available at http://gridengine.sunsource.net/, 2011.

[Tridgell 11] Andrew Tridgell. "CCache: A Fast Compile Cache." Available at http://ccache.samba.org/, 2011.

[Xoreax Advanced Grid Solutions 01] Xoreax Advanced Grid Solutions. "Incredibuild." Available at http://www.xoreax.com/, 2001.

2

Game Streaming: A Planned Approach

Jeffrey Aydelotte and Amir Ebrahimi

2.1 Introduction

The game industry has gradually been shifting away from traditional boxed titles and toward digital distribution. As of the writing of this chapter, Steam, a digital distribution service operated by Valve, currently has over 1,100 titles available through its service [Steam 10]. You might be hard pressed to find that many titles on the shelves at your local retail store. It's no surprise that this shift is occurring, as the numbers are there to back it up. In early 2010, Valve announced that it saw a 205% increase in Steam's unit sales year over year [O'Connor 10]. On September 20th of that very same year, NPD announced that PC digital game downloads surpassed retail unit sales by three million units [Riley 10]. Traditional game retailers finally took notice too, as GameStop announced in late 2009 that they would begin offering digital downloads, which they began testing in early 2010 [Paul 09].

With this shift towards digital distribution, it's not only important to provide a digital download, but it has become increasingly important to provide instant play. OnLive, a streaming game service, mitigates downloads by moving them to a central location: its own servers [OnLive 10]. There are inherently difficult problems that OnLive has had to solve and continues to address in order to provide that service to gamers, namely, overall bandwidth and latency. For the rest of game developers who are not building their games this way, streaming game assets to the client is *de rigueur*.

Some games on Steam are developed in such a way that players can begin playing the game before it finishes downloading. However, streaming has yet to get the same widespread attention in a game production as memory budgets, poly counts, textures limits, and bone counts. If that were not the case, then more of our games would be instantly playable. Just as with traditional categories for