

PIPELINEFX WHITE PAPER: Qube! for Game Development

Executive Summary:

Driven by insatiable consumer demand, the power of creative tools and the limitless imaginations of digital artists, today's game studios are producing CG content that defies the imagination in its depth, its vision and its realism. But with that ground-breaking content comes new challenges....

The competitive games market is demanding faster turn-around times while consumers are demanding better content, and more of it. To be competitive, studios are hiring larger teams and utilizing next-generation hardware, all of which is driving budgets into the stratosphere.

If the build process is not updated to address these changes, increasing data strains on build pipelines will ultimately slow down the entire process. This bottleneck can spell the difference between a studio's success and its failure.

Processing Builds – The Old Way

The old way of building games was to build on a developer's machine overnight, or on one or more dedicated build machines within arm's reach. This approach was easy to implement and required no re-tooling of existing build methodologies or pipelines. But it has some glaring weaknesses:

It doesn't scale well. Over the course of a project, data sets increase, driving build times up. Concurrently, the time window for building can shrink as deadlines loom. Adding machines in the midst of development can prove to be difficult and time-consuming. Some of the common problems with this approach include:

- Provisioning more or faster machines as dedicated build hosts takes time; procuring the machines, installing all the necessary software, and setting up accounts and network connections are all time sinks.
- When building across a small number of dedicated build hosts, the game build is usually manually partitioned; hostA builds assets 1-10, hostB builds assets 11-20, etc. Increasing the number of build hosts necessitates manually re-partitioning the game build.
- When a particular build host crashes or hangs, that piece of the game normally built on that machine is not being built. Temporarily decommissioning a dead build host requires another manual re-partitioning of the build.
- Manually load-balancing the build across all the build hosts is difficult, if not impossible. Often, the fastest machines end up sitting idle.

It's user-dependent. Build hosts are often set up to build only one aspect of the game, and as such, there is usually only one person who is fully cognizant of each machine's configuration. If this person gets sick, is reassigned to another project, or ends up leaving the company altogether, it interrupts the normal flow of the build process and time is lost trying to get the build process back to a working state.

Processing Builds – The Distributed Way

Cluster-based processing - like that offered by the **Pipelinefx** solution **Qube!** - can provide a solution to the shortcomings of standard build processing. Rather than depend on a few machines and one person to handle the build, cluster-based processing enables teams to efficiently generate, prioritize, dispatch and oversee build jobs by distributing builds across large clusters of inexpensive servers.

It scales. Once a build pipeline is structured in such a way that it can run on any arbitrary host in a cluster, it can run on all of them. Allocating additional resources to shrink the build time becomes trivial, since the build time simply becomes inversely proportional to the number of CPUs allocated. In addition, processor allocation is dynamic, ensuring that build completion by a certain time is an easily managed task. Load-balancing across a cluster becomes intrinsic due to the way that **Qube!** jobs are constructed and how jobs get assigned to hosts.

It isn't user-dependent. Cluster-based processing systems are smart systems. All knowledge that would normally be in someone's head or incompletely documented is encoded directly into the pipeline itself, minimizing dependence on one individual.

The Benefits of Distributed Builds

The benefits of these systems are clear. By distributing the build across multiple machines and minimizing the dependence on one individual, cluster-based systems:

Save Money on Build Management. Cluster-based systems save the time and resources it takes to manage complicated build processing, such as provisioning machines as dedicated build hosts, manually re-partitioning the game build every time the number of hosts increases, or manually load balancing the build machines.

Improve Productivity of the Entire Game Team. By mitigating the challenges of build processing, cluster-based systems increase hardware and personnel efficiencies. Load balancing minimizes downtime by ensuring that throughput, not machines, is maximized. In turn, these efficiencies allow programmers and artists to review their content more quickly, and get back to their creative tasks within seconds or minutes, not hours or days.

Improve Quality of the Content. Less time spent on the build process and build management equates to more time spent on the tasks that ensure a better quality product. Faster build turn-around gives programmers and artists infinitely more time to work on improving their content. QA teams can test the product more regularly and find bugs earlier in the process. These improvements ultimately give the team more quality control and ensure that the market receives the quality it expects.

Requirements for a Cluster-Based Build

Every build style has their requirements, and cluster-based builds are no exception. In order to get the best results from a cluster-based build, **Pipelinefx** suggests the following:

Cluster size: The larger the cluster, the more robust the system. That way, the failure of a single machine does not appreciably impact cluster performance. It does not have to be an entire machine room full of servers; 10 machines can be considered a good starting point. It's always easy to scale up later.

Similar configurations across all machines: To enable a job to run on any host in a cluster, it is imperative that the runtime environment be consistent across all machines. There must exist some mechanism of ensuring that a particular version of a piece of software is available when a job needs it. Hosts can either be identically configured, or a 'just-in-time' mechanism can exist that will install missing software as instructed by a self-describing job. Having a 'just-in-time' mechanism assumes that it is in a working state on each machine in the cluster.

Dedicated vs. non-dedicated cluster hosts: What are the merits of dedicated machines versus harvesting idle CPU time from end-users? The answer is usually based on the choice of operating system running on the machines, and is directly related to the ease by which multiples machines can be kept in a consistent state.

Linux- or Unix-based machines lend themselves very well to being non-dedicated hosts in a cluster; remote administration is a trivial task, and end-users do not normally run with administrative privileges. Consequently, each machine can be counted on to be in a known state at any time.

A large Windows-based cluster should only be comprised of dedicated machines. Most game development happens on Windows-based machines, where remote management is an arduous task and end-users almost invariably run with local administrative privileges. In these environments, software is often installed or upgraded without knowledge or permission from IT departments. The chances of build failure as a result of misconfigurations are therefore higher unless the team has dedicated some machines to the build process.

Design Tips for Distributed Pipelines:

Keep the atomic unit small. Write your pipelines so that the atomic unit is small. 'make arena tokyo' can be seen as moderately large, but not as bad as 'make arenas'.

Pick the proper time threshold. Pick a time threshold below which the sub-division of assets begins to yield diminishing returns. Five minutes might be a good place to start. There is overhead involved in starting up jobs on a cluster.

Write local builds and distributed builds identically. Write your pipelines so that a local build is identical to a distributed build. Not necessarily in the dependency collection or deletion phase, but the actual build itself should be identical. That way, the build process can be debugged locally, and can be run on a machine not in arm's reach of the developer with some confidence.

Provide debugging mechanisms. Provide a mechanism whereby temp files written out by your build can be written to a network disk. This can aid in debugging when problems arise.

Understand your dependencies. Know your dependencies; what's common across assets of a similar type, and what's asset-specific. Ensure that asset-common dependencies have a unique target in your pipeline. Don't assume that the first asset to build will construct these dependencies. If you assume this, you'll build the common dependencies on each machine as it receives its first agenda item.

Use shared locations for output. Write intermediate and final output to a shared location; don't rely on file transfer agents. You can easily debug crashed and/or failed builds in this way.

Conclusion:

The game environment is changing. Larger teams, next generation hardware and more data are placing greater strains on game development studios. **Qube!**'s state-of-the-art techniques for batch queuing, distributed processing and render farm management maximize job throughput and accelerate the millions of computational tasks that occur during the software build process. As such, **Qube!** is the right choice for studios that need to keep pace with the growing complexities and decreasing schedules of today's game development projects.

Accelerate your creative world. Discover **Qube!**.