

# **PIPELINEFX CASE STUDY: Using Qube! for a Distributed Game Build**

## **Game Development Case Study: *Distributing a Sports Game Arena Art Build***

### **Overview:**

The basketball game team was facing a daunting challenge on this year's game. In previous years, the game had been based on a one-arena model, with team-specific art loaded in at run-time to give the appearance of different arenas for each team. It was decided that this year's game would have custom geometry for each team's arena in order to add a higher level of realism to the franchise. However, this approach escalated the number of arenas from one to over 200. The build times for all arenas was now estimated in excess of 60 CPU-hours, a huge increase.

In addition, the team was concerned about maintaining the quality that had made the franchise a success in the first place. Creating art for a video game is an iterative process. The more iterations an artist can execute in a given timeframe, the better the end product will be. However, with build times now climbing, the development team worried that the product's quality would be negatively impacted.

To manage the increased data set and concurrently drive quality up, the team decided on a solution from **Pipelinefx** called **Qube!** While **Qube!** had historic ties to the film industry as a render farm solution, they recognized that its unique blend of features, in particular its distributed processing features, held all the necessary components to meet the particular demands of their project.

### **The Qube! Workflow**

#### **The first step: Creating Make Targets**

The first step in creating a distributed processing pipeline was to create make targets that could build an individual arena. This would prove key in the ability to distribute the arena build. The pipeline needed to be able to build only a single arena in one pass. By doing this, multiple instances of the build could run simultaneously across multiple hosts and each one could build a specific arena.

#### **The second step: Job Structure and Execution Flow**

The next step was to divide the dependencies that had to be satisfied to build arenas into two categories: dependencies that were common across all arenas, and dependencies that were specific to an individual arena. The common dependencies were again sub-divided into two categories: common geometry that needed to be built before any particular arena could be built and common art or textures.

### **The Qube! Workflow**

When a job is started on a cluster, the **Qube!** Supervisor instructs enough hosts to start up a worker process so that there is a worker for every sub-job. Increasing the number of sub-jobs once a job has started only involves the Supervisor instructing yet another worker to start up. Once each of these workers has started, the worker process then contacts the Supervisor and requests an agenda item. "I'm ready to

build an arena. Which arena should I build?" In the case where common geometry or textures is required, they are pulled down to local disks after the worker has started up, but before an agenda item is requested. This job initialization phase avoids getting common dependencies multiple times.

Once an agenda item is requested and received, the agenda item is examined. Agenda items aren't simply names, but are data structures built by the front end and passed to the back end. Embedded in these data structures, along with the arena name, is the list of all arena-specific geometry and texture dependencies. Once these dependencies are pulled down to local disk, the actual arena build can begin.

After an individual arena is built, any dependencies that are specific to it are deleted from the cluster host's local disk. This has two functions: it avoids filling up the local disks on the cluster hosts and eventually crashing them, and it ensures that the build never occurs with old art. While this may seem to be a profligate waste of network bandwidth, in actuality it does not add an appreciable amount of time to the build, and it contributes greatly to the overall robustness of the cluster and timeliness of the built art. Tracking down stale art on even a small cluster can be difficult.

At this point, the worker contacts the Supervisor and requests another agenda item. If there are more arenas to be built, the supervisor will dispatch another agenda item to the worker, and the process repeats. If the agenda is exhausted, the Supervisor instructs the worker process to shut down.

Once the Supervisor has informed the worker that the agenda is exhausted, the worker will delete all common arena dependencies from the local disk. The worker process shuts down, and the host then becomes available for running another build.

## **Qube! Benefits: Faster Build Times**

While it is impossible to estimate how much time **Qube!** saved using this specific workflow, or how many more iterations of artwork were created as a result of faster builds, it is possible to provide a real-world example of how the team benefited from using **Qube!**:

In the final stages of development, QA had come back with a bug that necessitated the re-building of all artwork due to a licensing-related legal issue. Time was short. If a substantial delay occurred, the game would get bumped in manufacturing, which would almost certainly result in a late ship date. It was already late in the day, so missing that day's submission deadline seemed imminent.

Since the **Qube!** pipeline had been written to scale linearly across an arbitrarily large compute farm, the team was able to commandeer all processors in the farm and complete the 60 CPU-hour build in under 80 minutes, making their deadline and ensuring that the game shipped on time.

### **About the Author:**

John Burk has worked in the game/CG animation industry for over 10 years, specializing in pipeline and workflow development. Companies he has worked for include Oddworld, Electronic Arts, Square, Mainframe Entertainment, and Radical Entertainment.