

THE ALL-IMPORTANT IMPORT PIPELINE

////////// HOW AND WHY TO
RID YOUR ARTISTS OF AN
EXPORT-BASED APPROACH

IT'S NO SECRET THAT THE WORLD of art development for games is getting more complicated, with dynamic rig LODs, real-time tessellation, sub-d surfaces, mega-textures, and the like. The complexity of assets is increasing, while at the same time, there's a drive to simplify the process to create them. If the initial technical hurdle is lowered, artists will be free to be more creative and spend more time iterating art, and less time wrangling their tools to try and just get the assets into the game (or so the thinking goes).

One really good initial step toward helping your artists do what they do best is to alleviate their dependency on one specific software package. Long gone are the days when a studio could get by just with a copy of [insert your favorite package here]. Studios should be aiming to sever the umbilical cord and let their art teams work with what they know or what's best for any given task. Beyond the implicit benefits, there's also a world of good to be had in using this approach when working with outsourced employees.

The trick to making the switch is to change the studio's mentality by breaking down the

export barrier and implementing an import-based pipeline.

EXPORT VS. IMPORT

What's the real difference between export- and import-based pipelines?

An export-based pipeline is typically structured so that assets are exported directly to a custom first-party priority native format. Once exported, the asset is loaded directly into the engine to be binarized. The format is often spec'ed in such a way that the content is written in a very specific structure dictated mainly by the engine's internal data structures. An example would be to de-index vertex data, triangulate, then section out materials, normals, and tangents.

Creating your own custom exporter initially seems like the path of least resistance. Why should we try to interpret or work with someone else's exporter? We can just quickly write our own and make it do exactly what we need!

While this is true and can help jumpstart production, it will ultimately confine and limit your production team.

One of the problems with an export-based pipeline is it becomes challenging to support

artists who are working in multiple art packages. A unique exporter has to be built and maintained for each software application. Some studios avoid this maintenance nightmare by establishing one core package as the gatekeeper of data (see Figure 1), a practice that utilizes a range of third-party established formats to transfer between external packages while ultimately forcing all content to be passed through one primary package's exporter before it can be loaded in the engine.

But what if, during production, a superstar programmer develops tech for the engine that will allow you to render all meshes via subdivision surfaces? The programmer tells you that triangulated meshes aren't optimal for this new tech and the system would really prefer quads. You'll now have to update the engine's native mesh structures to support quads, modify the external exporter(s), and—you guessed it—re-export all the mesh assets!

Another more common example is that your project is targeted for multiple hardware platforms. You're going to have to maintain multiple exporters for different targets and at the

very least re-export the assets in a way that suits that target platform, most commonly down sampling the assets in the process. Needless to say, in an export-based pipeline, managing two distinct sets of exported assets and art pipelines can get quite complicated (see Figure 2).

On the other hand, an import-based pipeline shifts the task of native conversion from the export step to the import through a transparent background process. Rather than defining a custom export format, the pipeline makes use of a standardized format (for example, FBX or Collada for models and PSD or TIFF for textures). The asset is saved to this common format and then sourced in the engine. The act of sourcing the "loading" of the asset in the engine triggers a background process that converts and imports the data from the source asset as needed. It might, for example, convert units, de-index vert data, Y-up to Z-up, combine, flatten, resize, recolor, and so forth.

PRESERVATION

The real magic of using an import-based pipeline is that the sourced asset is not modified during the process and therefore can be reprocessed multiple times with different settings and through different processes without its quality degrading.

As an example, if you support the PSD texture format in your engine (import pipeline) and convert it to native DDS in the background, but later down the development timeline decide to switch all textures to a JPG-based SVT solution, you would be able to reprocess and compile the source PSDs easily without any loss of quality of the final texture. However, if you've spent the whole project forcing the production team to convert to DDS, and reference these DDS in engine (export pipeline), then this format change would force you to either:

- recover all source PSDs and convert to the new format (remember these aren't referenced anywhere but in the version control software) or
- convert the DDSs to the new format and face the problems of recompressing the compressed data.

Simply put, an import-based pipeline allows you to retarget assets without wasting many work months or degrading the quality of the final asset (see Figures 3A and 3B).

Saving conversion settings is also a helpful extension to the pipeline. If you store the parameters used to convert the asset as an associated set of metadata, you can dynamically reprocess source content as needed. This metadata is best saved in a shared database to cut down on the cluttered files and file formats. The metadata

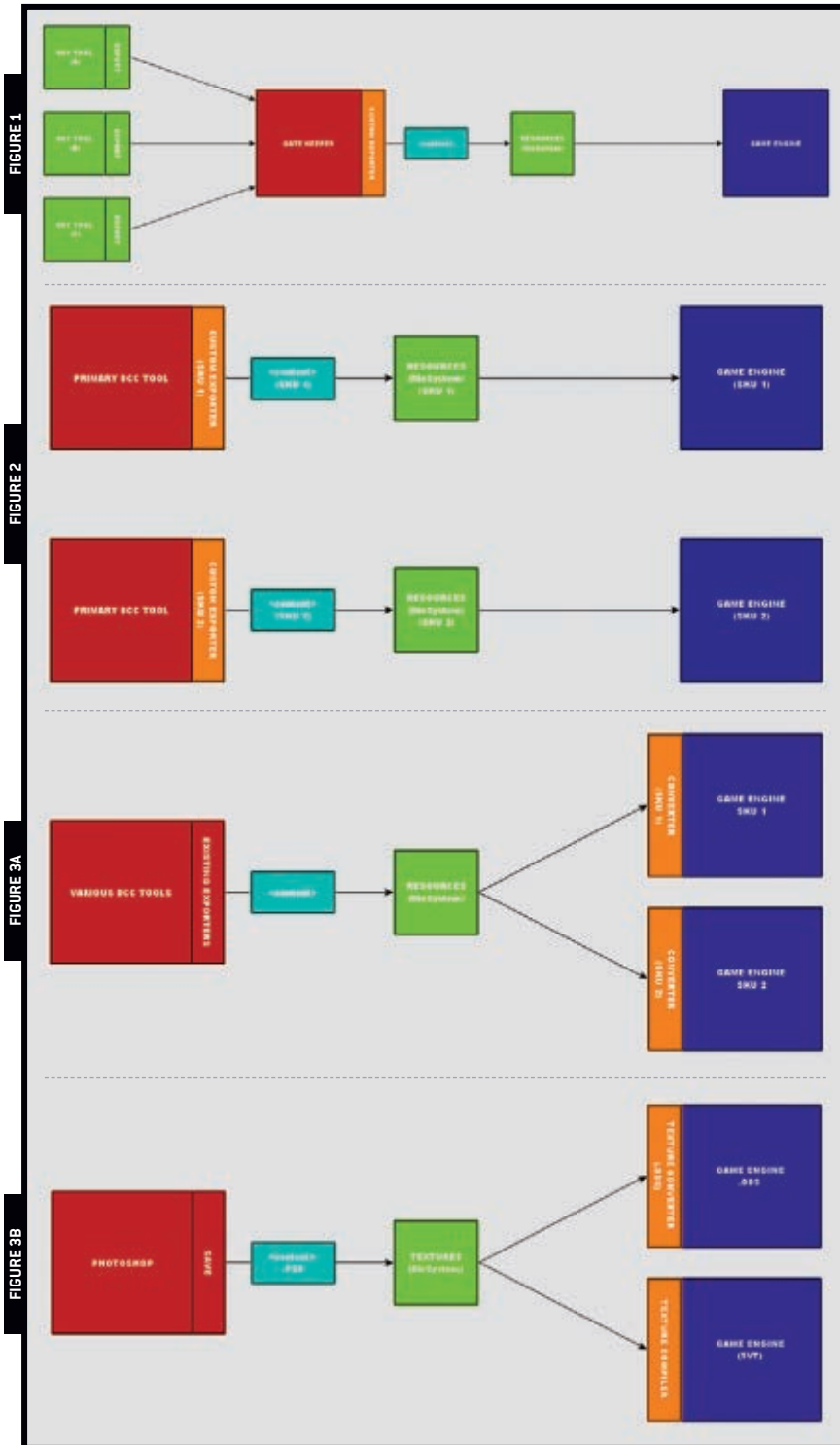


FIGURE 3C

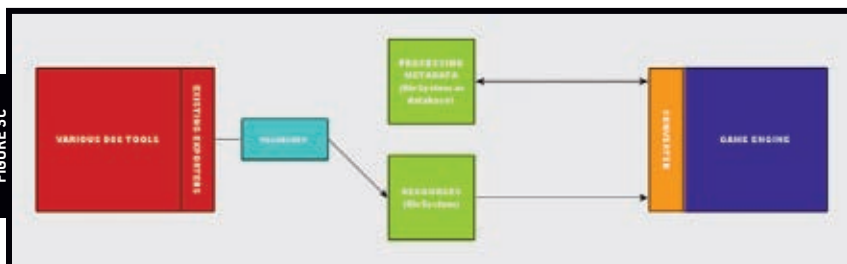


FIGURE 4

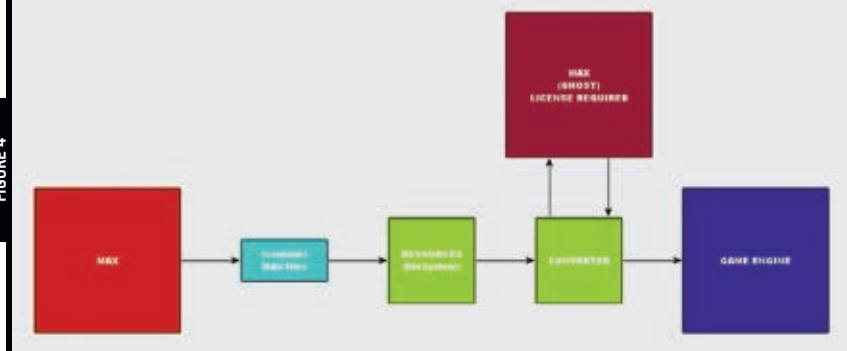
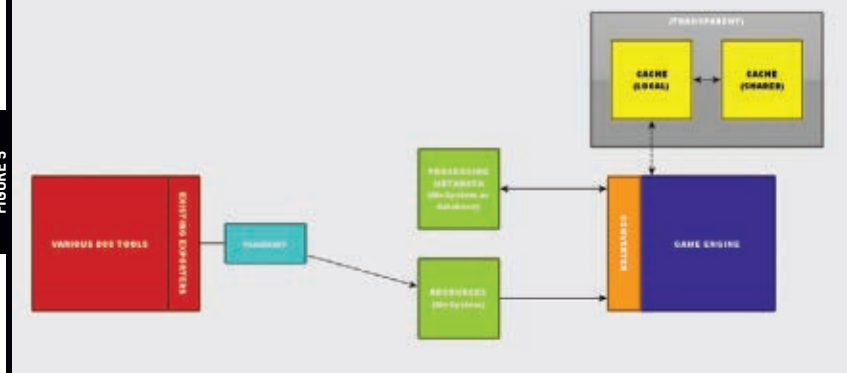


FIGURE 5



can also be used to tweak incoming assets, for example adjusting the levels of a texture, sharpening texture mips, replacing texture mips, recalculating the tangents of a mesh, specifying automatic mesh LOD reduction percentages, and so forth (see Figure 3C).

As a byproduct of not defining your own custom format, the pipeline's ability to support multiple packages becomes trivial as long as you picked a format that's open and supported across these packages, the prime examples being FBX and Collada. It's also helpful that these formats and plug-ins are externally maintained so all version updates are the responsibility of other and much larger teams.

The format you chose is pretty important. I know of a studio that based its pipeline on binary Max files, which seemed like it would achieve the ultimate goal of "import and convert." However, there was one key problem. Max files require the 3ds Max client to load. You cannot open these files without spawning a ghost version of 3ds Max in memory and using 3ds Max-compiled plug-ins to query the data. In other words, they're back to exporting! This is the very definition of a closed format (see Figure 4).

BRINGING THE PROCESS IN HOUSE

It's important that you take the time to research and evaluate all the different formats that are available. I recommend looking for formats that:

- » have the features you need
- » are public (if not open source)
- » have a well-documented SDK
- » are commonly used and frequently updated
- » are ASCII supported, which can help but is not required

One question people may have is, "If we implement an import-based pipeline, doesn't that mean the team will have to convert assets every time it's loaded?"

Yes. However, the amount of conversion can be reduced significantly to only converting new and modified assets—not all assets—using a simple caching system. When an asset is loaded and ready to be processed, the binary results are stored in a local cache directory. If the asset has already been processed and nothing has changed, then the engine will directly load from the cache rather than reprocessing the asset again (see Figure 5). This helps alleviate consistently slow load times. Furthermore, the more componentized your assets are, the less of an impact an updated asset will cost to reprocess.

Consider an animator's workflow. A minor tweak is made to the main character's idle animation. The assets are in a non-componentized format (all animations stored in one asset source), and so any change to any animation will force a complete reprocessing of all the animations for that character and not just the new idle clip.

Two other major features are required to truly take advantage of the cache system. The first is distribution, a pretty simple feature, but an important one. A clean way to use it is to distribute cache via your build-release process (using a build installer, synced from version control or synced from a share). With the cache now shared among the team members, they are able to take full advantage of speed loading the native binary data.

The second major feature that's required is "invalidation," which allows the system to force a flush of existing cached assets. Incorporating a cache version to the cached assets can prevent the engine from sourcing any data that doesn't meet the minimum version requirements. With this simple cache versioning, you can quickly update the build to invalidate any data that might cause instability for the team.

Once a solid caching system is set up, you'll get the advantage of loading native binary assets—speed and robustness—without the limitations of being tied directly to the binary format!

In general, the goal is to shift the technical burden from the artists and designers over to the more capable hands of the tools and pipeline

engineers. I've noticed in my career that some programmers tend to see assets as very explicit sets of source data that should be categorized, formatted, stored, and so on before they want to have anything to do with them. What basically ends up happening is the artists are tasked with organizing and maintaining two very distinct sets of assets. The assets they want to work with (PSD, for example) and the assets the engine wants to work with (DDS, for example).

Over and over again, confusion occurs about what exactly the engine wants. Because of a very distinct disconnect between the source (PSD) and the working format (DDS), the source often gets lost; conversion options are lost, or flattened accidentally—or heaven forbid someone makes a change to the working format (DDS) directly.

TECHNICAL ARTISTS

Slowly over the years, a new group of artists has evolved to take this burden away: the ever-elusive technical artist; someone who knows how the artists want to work and how the programmers want the assets.

Hopefully, with an import-based pipeline, the need for asset wranglers is slowly waning, leaving them more time to focus on other tasks, like writing helpful tools and extensions to assist in other areas of art development. What I suggest via an import-based pipeline is a kind of philosophy whereby the artists and designers are allowed to work how they want, and all the technical stuff in-between is managed for them.

Alas, there is one area where the principle of an import-based pipeline isn't as magical: animation. As you go from textures to models to rigs and animation, the complexity of the problems and the data increases dramatically. At the core every package animates uniquely. The animation systems between art packages are a diverse group and something as simple as IK will evaluate differently across all.

The task of seamlessly transferring animation without baking between packages has been tackled over and over. To date, I don't know of one system that has truly succeeded. The best way I can explain the problem is with a texture analogy. Consider a texture that has been created in Photoshop and is using all the cool features (layers, text, paths, adjustment layers, multiple alpha channels). Now, take that texture and load it into Microsoft Paint. How can Paint possibly expect to be able to edit this texture without actually becoming Photoshop? So how can you work with it? The only way you can is to first flatten the image (rasterize it).

Interestingly enough, Photoshop does this automatically for you when you save. Photoshop will save a PSD file with both flattened and unflattened versions of the same image, a feature that can be turned off, but nevertheless exists.

So like the above example, animation pipelines have to bake animation data (sample data every frame) upon export, which then requires you to maintain multiple versions of the animation, one baked, one unbaked.

I have yet to find a clean solution. Maybe a system like Photoshop's could be added to the 3D world, wherein the animation package, on saving, collapses the animation data into linear, per-frame FK keys. I think for this to enter into the world of the true import pipeline, a standardized animation engine would need to be developed to define the animation systems, as

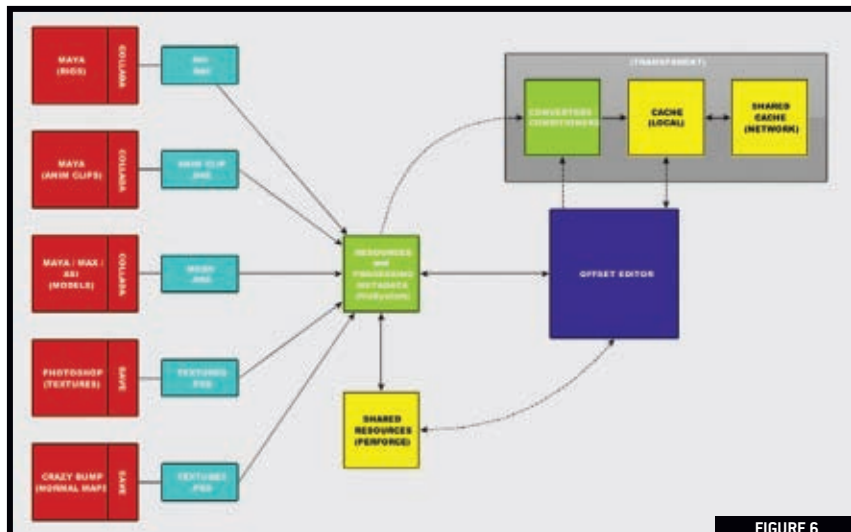


FIGURE 6

well as how they are evaluated each frame (i.e. evaluate constraints, then expressions, then forward kinematics, then etc ...).

This isn't on too many people's radars, and I would bet it will be a while before this problem is solved. For now, animation is the one exception to an otherwise very simple pipeline.

OUR ART HOUSE

Figure 6 shows our current import-based pipeline that we're using for PROJECT OFFSET. We've done our best to make it as simple as possible for the artists, while simultaneously allowing us to upgrade and utilize a full suite of packages for production. For instance, we were able to upgrade to Maya 2009 with little to no impact to the team. Technically, we're able to work in any version of any package that can export Collada content. The maxim we've adopted for the pipeline is, "It just works."

As you can see, we're pretty Maya-centric in our pipeline, but I assure you this is purely based on choice. If required we'd easily be able to switch over to be more focused on Softimage XSI or 3ds Max without much hassle at all.

LONG-TERM SIGHTS

There are definite benefits to switching to an import-based pipeline, with options and simplicity for the artists being the key advantages.

In general, I urge developers to investigate systems and changes to the pipeline that can shift the technical and asset management burden from the art and design teams over to the programming and technical art teams. While the programmers and technical artists will have to work a bit more to keep things running smoothly, the artists will be freed to spend more time iterating and making the art look as good as they can.

Implementing an import-based pipeline requires a bit of heavy lifting at the start, but I guarantee it'll save you many times over down the road. ❖

ROD GREEN is currently the technical art director at Intel where he manages the art and design pipelines for the game engine team group. Previously he was one of the managing directors and COO of Offset Software, the developer behind PROJECT OFFSET. Email him at rgreen@gdmag.com.