UNREAL ENGINE

Fortnite Trailer

Developing a real-time pipeline for a faster workflow

Contents

Fortnite Trailer

Developing a real-time pipeline for a faster workflow

In July 2017, Epic released Fortnite, a video game where players save survivors of a worldwide cataclysmic storm. Players build fortifications and construct weapons from scavenged materials while fighting off monsters and other enemies. As of August 2017, the game had over a million players.

In late 2016, in preparation for the game's release, Epic Games began production on a three-minute cinematic trailer for Fortnite. Epic's goal was to create a real-time animated short with the same quality as a pre-rendered sequence, but which would allow real-time navigation and interaction with the set during the production process.

The purpose of Fortnite's trailer was to showcase its fun and unique art style, and introduce the story's goal as well as gameplay elements such as scavenging, building, and defending.

Figure 1: Scene from Fortnite trailer

This document outlines the process of creating the Fortnite trailer using Unreal Engine, with a pipeline designed to minimize production time while allowing maximum collaboration and creativity.

While Unreal Engine was originally designed as a game development tool, its integration of many animation pipeline tasks—versioning, rigging, animation, editing, editorial review, changelist distribution, and others—make it an ideal tool for animation work as well. Its real-time rendering capabilities contribute to the mix, giving instant feedback for faster work and better results.

Pipeline Development

Pipeline Development

To develop the pipeline for the Fortnite trailer, the Epic team first considered the trailer's goals and the tasks involved.

The trailer contains six sequences comprised of 130 individual shots. Specific goals included:

- Final trailer can be played back in real time in Unreal Engine
- Frame rate of 24 fps
- Length approximately three minutes
- Visual quality of real-time animation at least as high as pre-rendered.

The major tasks required to create a pre-rendered trailer are similar to those for a real-time sequence:

- Environment and character design
- Storyboard
- Determination of tools to use (software, hardware)
- Sequence breakdown
- Assets such as sets, characters, and rigs created or collected
- Rough layout of motion
- Sound and dialogue created or acquired
- Actions/motions created or collected
- Asset editing by multiple team members
- Scene building with multiple assets
- Animation based on rough layout
- Lighting
- Rendering
- Post processing
- Testing output at desired resolution, playback speed, etc.
- Review of work by senior production team members
- Final output

The way these tasks are arranged into a workflow, and the choice of tools to perform these tasks, can have a major impact on man-hours spent, quality of the result, and even whether the project is finished at all.

Epic's goal with the Fortnite trailer was to streamline the workflow to create a high-quality animation in minimal production time (and with minimal stress). To achieve this goal, the team utilized Unreal Engine 4 as the centerpiece of its pipeline.

Linear vs. Real-Time Pipeline

In the early stages of developing the Fortnite trailer, the Epic team considered the best way to improve on traditional pipelines.

Traditional linear animation pipelines take an assembly line approach to production, where tasks are performed sequentially. Newer and more holistic pipelines opt for non-linear and parallel methods to process and distribute data.

While a linear pipeline can produce results, it has certain drawbacks:

- **Limitations to changes.** In any production, animation needs to be broken down into distinct requirements, each of which is critical to the nuances of performance. An improvement for a specific requirement might require a subtle change to an asset's motion, a set's lighting, etc. In a linear workflow, if such a change is required while later steps are underway, the changes can be difficult or time-consuming to propagate throughout the project. Such a limitation can lead to a reluctance to make changes and improvements, which ultimately affects the artistic quality of the production. Since the entire point of the production is to create the best animation possible, such a limitation can defeat the purpose of the project.
- **Post processing required.** Traditional linear animation

studios commonly produce output as layers and mattes to be composited at the end of the project. While this approach does allow for a high degree of control over the result, it adds a great deal to the overall budget and schedule. Post processing is often done outside the creating studio, which further divorces this step from the rest of the production.

To minimize these limitations, Epic needed a new type of pipeline for the Fortnite trailer. With the tools available in Unreal Engine, Epic was able to develop a real-time pipeline that eliminated the problems of a linear pipeline, smoothing the production process while maximizing artistic quality:

- Interactive creative process. The entire production pipeline is represented within Unreal Engine from asset ingestion to animation, effects, lighting, editing, and post production. Reviews can be performed on each part individually or on the project as a whole to determine whether changes are needed. Artists can make iterative changes easily within Sequencer, the cinematic tool within Unreal Engine. The time between making a change and seeing it reflected on-screen is instantaneous, facilitating an interactive and creative process.
- Ease of editing. Sequencer can create, modify, reorder, or even delete shots in real time. Sequencer acts as a production hub, combining aspects of the editorial department with layout and animation in a non-linear timeline editor.
- Faster output. The entire process of post effects can be dealt with inside Unreal Engine, thus reducing the need for external compositing requirements.

Using Unreal Engine as an integral part of a real-time pipeline made additional methods available for improving visual quality and saving time. The team made the decision to utilize several of these options for the Fortnite trailer:

- Leverage existing game assets.
- Deploy an improved facial animation solution through Alembic caching.
- Implement dynamic lighting over baked lighting to maximize visual quality.
- Deploy priority overrides so lighting and objects in each shot could be adjusted individually.

Real-Time Pipeline: A Closer Look

The real-time pipeline utilizing Unreal Engine was developed to address the various and minute issues of animation production, each of them vital to artistic quality, technical considerations, and conversation of artists' time.

To illustrate the components of this new type of pipeline, the following table shows the traditional linear animation workflow and compares it with a real-time pipeline centered around Unreal Engine.

	Linear Animation Pipeline	Real-Time Pipeline
Rendering Method	Linear rendering	Real-time rendering
Primary Creative Tool	DCC	Unreal Engine
Workflow	Less Parallel	More Parallel
Data Organization	Decentralized	Centralized
Developmental Model	Pull	Fetch & Push
Naming Convention	Strict	More Relaxed
Versioning vs Source Control	Manual w/ Symbolic Linking	Atomic Transactions
Output Target	Layered	Final Pixel Output
Asset Considerations	Unlimited Topology	Optimized for RT

Table 1: Comparison of linear and real-time pipeline

Rendering Method

Until recently, the difference in quality between pre-rendered and real-time sequences was noticeable. In recent years, real-time rendering has improved to the point where, if done properly, its visual quality is as high as that of pre-rendered sequences.

A traditional pipeline includes one or more rendering steps. Real-time rendering doesn't just eliminate these steps—it offers a new way of working with all steps in the pipeline.

In a traditional pipeline, the need for a rendering step adds hours or days between changes and review. With real-time rendering, changes are visible almost immediately so work can continue.

Real-time rendering opens the door for the real-time pipeline, where various tasks can be performed instantaneously or concurrently rather than in a linear sequence. The divide created by the rendering step in a traditional pipeline is minimized and, in some cases, eliminated altogether.

Primary Creative Tool (DCC vs Unreal Engine)

DCC (Digital Content Creation) software packages like Maya, 3ds Max, Cinema 4D and Blender are commonly implemented as the central software packages in a linear production pipeline. However, these packages are designed to perform specific tasks in an animation pipeline such as modeling and rigging, not data aggregation and organization.

By contrast, Unreal Engine is designed to bring the production requirements of multiple disciplines and their data into a centralized framework, where artists can work on creative tasks in parallel. Additional tools for aggregation and organization are unnecessary.

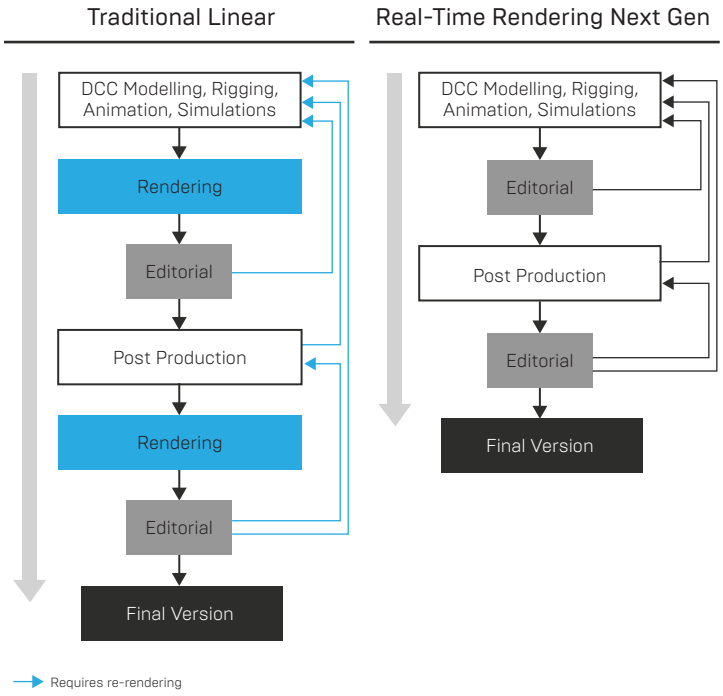


Figure 2: Comparision of pipelines with traditional rendering and real-time rendering

Workflow

Ideally, a pipeline allows for production work to be performed in parallel as much as possible.

This can be accomplished through two primary factors. The first is the usage of levels and sublevels within Unreal Engine. With this functionality, supervisors can divide up the workflow into specific disciplinary stages within the engine. Artists are free to work on their areas while keeping the main project file intact. Once work is completed, it's pushed back up into to the depot and transmitted to everyone working on the project.

The second factor, real-time rendering, makes a parallel workflow possible by eliminating long wait times for renderings, enabling the team to review changes as soon as they're made.

Efficient file management is also part of a parallel workflow. To keep the pipeline moving, artists and programmers can't waste time looking for files or editing out-of-date scenes.

Data Organization

In traditional pipelines, data tends be stored across multiple file servers and various locations with different naming conventions, which can require carefully-designed structures and scripts to point to the proper file. There usually is a "home" directory of sorts, but it can be confusing.

In a real-time pipeline, data is tracked from a centralized depot. For this task, Unreal Engine utilizes UnrealGameSync (UGS), a graphical front end outside Unreal Engine that synchronizes and builds Unreal Engine projects.

Approved versions of files are published to the depot where they become available to other users, and all the files needed for a project can be accessed from this single user interface.

Though the system was originally designed for game development, Unreal Engine's centralized approach to data management lends itself perfectly to animation production.

Developmental Model

During production, artists are constantly saving existing files with new information and/or creating new files that replace older ones. Other artists, in the meantime, need to know such files have been updated, and need access to the updated files.

Due to the nature of this workflow, any pipeline's file management system must be designed to address certain needs:

- When an artist is working on a specific file, a mechanism is needed to inform others that he/she is working on it to prevent double-work and overwriting of each others' work.
- When the change is complete, the fact that the file has been changed (or that a different, newer file is now the one to use) must be recorded.
- Other artists who might use these files must have a way to find out about changed files, latest versions, etc.

Linear animation production typically relies on pull-based systems, where the artist chooses which set of files are required to construct a scene. The artist's selection is typically based on a manually-updated list distributed to artists, or guidance/limitations provided by a custom version control script.

The manual method tracks access and changes through written or verbal reports from artists and managers, with the data stored in a spreadsheet or other application separate from creation tools. Manual versioning is time-consuming and prone to error, as it relies on individual(s) to update the list, and on artists to accurately report work-in-progress and edits to files. While manual versioning can work for small, short projects with two or three artists, in larger teams the errors can quickly compound.

Larger studios sometimes write custom scripts for their DCCs that assist artists in choosing and tracking assets. While superior to manual versioning, this approach has its own drawbacks:

- A new script must be written for every DCC package, and scripts must be updated for new software versions.
- DCC scripting languages are designed to automate native DCC functions such as modeling and animation, not to facilitate version management. A DCC scripting language typically doesn't include, for example, a mechanism to lock a file for editing. A script programmer might find a clever way to control access using filenames, but this means any artist's deviation from the naming convention will cause tracking to fail.

In short, DCC-based versioning, while better than manual methods, remains clunky, limiting, and prone to error.

Some studios use Shotgun, a review and tracking software toolset, as a partial versioning solution. But while Shotgun

excels at production tracking, it has some limitations as an asset and version managing system.

With all these processes, the strict rules of reporting and retrieving are often in direct opposition to the creative process. An improvement gets lost because the edit wasn't recorded, or worse, isn't done at all because of the artist's reluctance to do administrative tasks. (It's worth remembering that artists are hired for their artistic talents, not their project management skills.)

In contrast, the fetch/push system used by the real-time pipeline utilizes the tenets of ACID database transactions to manage access and changes to files. With this method, a user can check out a file from the UGS depot for editing (as in checking out a book from a library). While an item is checked out, other users are automatically locked out from editing it. When the user is done, he checks in the file which again makes it available to others for editing. The UGS records the change and disseminates it via a changelist across the depot.

With such an approach, the three needs identified earlier are completely fulfilled while at the same time eliminating the errors and excess time associated with manual and script-based file management schemes.

Versioning vs Source Control

While traditional linear pipelines typically rely on manual version histories, versionless masters , and/or sophisticated version selection scripts, a real-time pipeline uses a single interface to control all source files including scenes and engine code.

Unreal Engine itself is editable--a programmer can add functionality to the engine to improve performance or customize its behavior. In addition to providing the reliability of a fetch/push system for asset management, UnrealGameSync (UGS) provides synchronization and dissemination tools for any code changes to the engine.

Source control for all files is provided by Perforce P4V, a leading enterprise version management system. UGS acts as a wrapper for Perforce to provide a seamless experience in managing data for Unreal Engine.

UGS is designed to facilitate low overhead and fast iteration times between designers, programmers and artists. Specific features offered by UGS:

- New changes can be disseminated across the entire depot, allowing other users to synchronize to the data. Developers can sync to a changelist as soon as it's submitted, and locally compile the source code matching that change. They can add comments to each change, or flag builds as good or bad for other developers.
- Compiled editor builds can be zipped, synced and decompressed automatically for artists.
- Engine version files are updated to the synchronized changelist separately from data, so developers can make alterations to the engine code without altering the assets and their usage in the project. Artists can sync to update engine code without being forced to download all the assets again, making synchronization a quick process.
- Individual engineers can flag the rest of the team when they're working on fixing a broken build.
- File activity logs can be displayed alongside the list of submitted changelists.
- Build steps can be customized to utilize project-specific tools and utilities.

[1] A *versionless* master is a copy of a specific version of a file, or a symbolic link that points to a version somewhere else. The point of the versionless master is that the artist can simply load a scene and all the references to versionless masters in the scene point to the appropriate files or links. Its intent is to prevent artists from having to worry about selecting a version—whenever they open their scenes, they'll always get the latest version. While versionless masters are a possible solution to file management during production, they still need to be maintained and updated with a system of some kind.

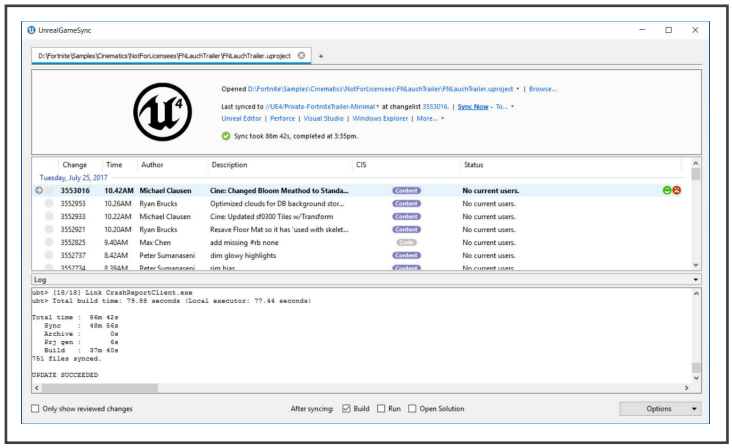


Figure 3: Source control with UnrealGameSync

Output Target

Traditional linear animation studios commonly take the approach of rendering their output as layers to be assembled in compositing with other elements, such as background matte paintings and effects. While this approach does allow for a great degree of control over the result, it adds to the overall budget and schedule.

By contrast, a real-time pipeline embraces a final pixel output philosophy. The entire production pipeline, from asset ingestion to animation, effects, lighting, editing, and post production, is aggregated and represented in a single module or interface. Whether changes are made upstream or tweaks are made downstream, the result is updated and displayed in the WYSIWYG interface as soon as the changes are made available.

Unreal Engine is designed to support final pixel output. The real-time aspect of the program's performance means users can make iterative changes easily within Sequencer without overly impacting departments upstream. The entire process of physics-based simulations and post effects can be dealt with inside the engine thus reducing the need for external compositing requirements.

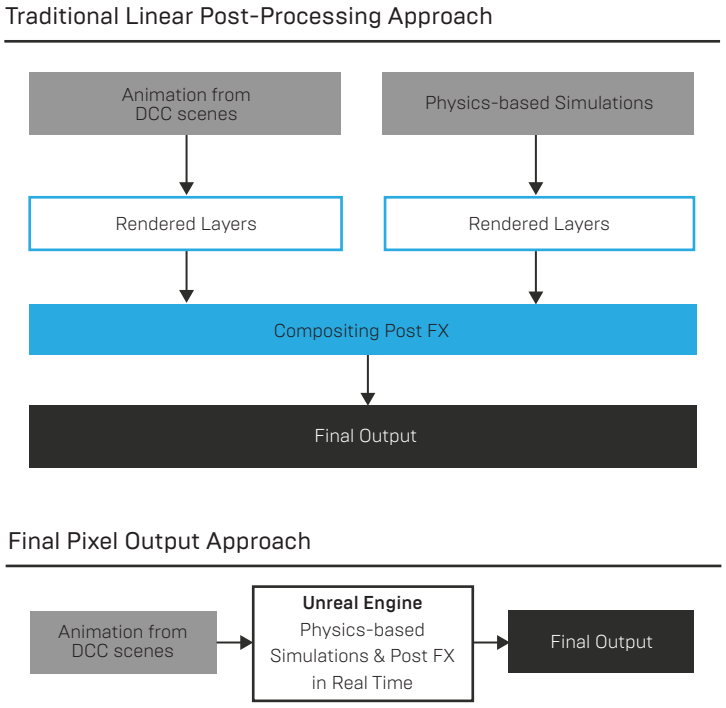


Figure 4: Traditional vs. Final Pixel Output pipeline for post processing

Production Concepts

Animated film production borrows many concepts from live action movie production. One concept is the linear division of a project into pre-production, production, and post-production activities. For live action films, the division is clearly centered around the shooting process—shooting the film is production, anything done beforehand is pre-production, and anything done afterward is post production.

One of the reasons for this clear-cut division of activities is the cost of assembling a crew for a live shoot. If the sets aren't fully prepared (a pre-production step) and the film crew and actors are standing around waiting, money is being wasted. In the same vein, having to reassemble a cast and crew to reshoot a single scene during the post-production phase can add a great deal of cost.

For similar reasons, traditional linear animation pipelines follow a clear-cut division of pre-production, production, and post-production activities centered around the animation process as the production phase. Set and character design fall into pre-production, while scene assembly and finishing are part of post production. Any changes that don't follow this linear structure, such as a set change during post production, adds time and cost.

While this linear approach does result in finished work, it puts a great deal of pressure on each of the teams to "get it right" before moving their work on to the next phase.

Conversely, a real-time pipeline is not as rigid—steps are no longer in a sequential order but can be performed simultaneously, and some steps that artists are accustomed to thinking of as production or post-production steps can be moved earlier to save time. Real-time rendering and a centralized approach to data storage/retrieval allow for this fluidity, where any step in the process can be placed wherever necessary to save time and improve the quality of the finished work.

Sequence Breakdown

Another concept that animation production borrows from live film is the *sequence breakdown*.

Before shooting begins, the director of a live action films breaks the script down into a series of smaller structures to help organize story flow and action specifics. This approach to live action films translates well to the animation process.

Sequence - The script is broken into large chunks called sequences, with each break occurring at a logical interval. A common sequence break point is a switch to another location or subplot.

Beat - Each sequence is broken down into smaller chunks called beats. A beat is a major action or mini-conclusion to the action, a switch to a different environment or point of view, or some other natural division in the sequence.

Shot - Each beat is broken down into individual shots. A shot includes a specific camera setup (still or moving) and action from some part of the script. If the same action is shown from a different camera setup, that's a different shot. If a different part of the script is acted out on the same camera, that's a different shot.

Scene - The environment where action takes place. The same scene might be used for several shots in different sequences.

Take - Duplicate versions of the same shot. The same camera setup is used and the same action is performed, but with small (or large) differences in motion, emotion, etc.

Cut - The term *cut* has several meanings. In live filming, it is common for the beginning and end of a shot to be trimmed off during the editing process because they contain items that aren't part of the performance, such as the director saying, "Action!" or an actor entering from offscreen. A trimmed shot is called a cut, and the action of trimming the shot is called cutting. The term cut also refers to an edited, final version of the film. Examples of this usage include G-rated and R-rated cuts of the same film for different audiences, or the "Director's Cut" version of a major motion picture available only on DVD.

Breaking down sequences is an administrative task that can take place anytime during pre-production to aid in organizing work.

Pre-production Steps

Animation projects include these steps at the outset, in roughly the order listed.

Design - At the start of a project, artists get to work designing the environment, characters, weapons, and other scene elements.

Storyboard - Next is a *storyboard*, a representation of the entire animation as a series of pictures. A storyboard can be likened to the comic book version of the animation, with each image representing an individual action or piece of dialogue that moves the story forward. Often the storyboard is a set of hand-drawn cards physically tacked to a wall, where the cards can easily be moved around during the review process to refine the story. Software can also be used for storyboarding provided that images can be easily moved around during review.

Story Reel - Using editing software, the storyboard images are arranged into a *story reel*, a movie sequence with each image held on screen for the approximate time the scene will take in the final movie. Sound or dialogue might be added to aid in the review process. Other names for the story reel include *cinematic* or *animatic*.

Sequence Breakdown - As described in the previous section, the script is broken into sequences and shots to organize work.

Rough Layout - A *rough layout* is rudimentary movie that replaces each image in the story reel with approximate motion, sound, etc. By the end of the production process, each section of the rough layout will have been replaced with a polished animation sequence to produce the final output.

Each step—design, storyboard, story reel, and rough layout—is reviewed by the director and other team members before going on to the next step.

Linear vs. Real-Time Pre-production

While the overall pre-production steps remain the same for linear and real-time pipelines, there are differences that make the real-time pipeline a more efficient approach.

In a linear pipeline, designs must be polished and complete before pre-production can be considered finished. In a real-time pipeline, a rough design is sufficient to move through pre-production, and design can continue in parallel with production until final, polished designs are achieved.

In a linear pipeline, a rough layout is created through a laborious process of manual keyframing followed by several rounds of reviews and corrections. Sequence Breakdown must be done before rough layout so the keyframing work can be assigned to artists. In addition, a lot more keyframing is required during the production phase.

In a real-time pipeline, motion capture (mocap) can be used to produce an interactive rough layout that can be reviewed and approved on the fly. The script is broken down into sequences before the mocap session, but breakdown into shots is done after after the director has chosen the mocap takes for the rough layout. This approach also provides animation keys for motion that has already been approved by the director, saving review time during the production phase.

Fortnite Pipeline

A number of tools were used to create the Fortnite trailer, all of which needed to be incorporated into the production pipeline with Unreal Engine:

- Autodesk® Maya® – Low-poly modeling, UVs, animation
- Autodesk 3ds Max® - Character and hard surface modeling
- Pixologic ZBrush® - Sculpted details
- Modo™ - Retopology tools
- Adobe® Photoshop® and Allegorithmic® Substance Painter - Texture painting
- Allegorithmic® Substance Designer - Materials and textures
- xNormal™ - Baking normals for environment assets
- Autodesk Motionbuilder® - Motion capture data processing (data captured with Vicon motion capture hardware)
- Blender® - Dynamic destructive simulations
- Adobe Premiere® – Animatic (offline) editing, final editing
- Autodesk Shotgun™ - Production tracking
- Vicon® Blade™ - Motion capture

File formats used for data transfer:

- Autodesk FBX® - Transfer of environment meshes and body rigs/animation
- Alembic - Transfer of facial rigs/animation

Tools for data sequencing and aggregation:

Unreal Game Sync (UGS) - Asset aggregation and version control

Sequencer - The production hub, combining aspects of the editorial department with layout and animation in a non-linear timeline editor.

Sequencer takes the place of several external applications by providing editing tools, animation controls, etc. UGS and Sequencer, when used together, recreate a "studio" like atmosphere within Unreal Engine itself.

The following diagram shows the production pipeline developed for the Fortnite trailer.

Even though the entire trailer could be played in real time within Sequencer, daily reviews were more easily accomplished in a dedicated screening room. To this end, each night the entire sequence was rendered to 1920x1080 PNG files (the final output resolution) via Sequencer, and dailies created for review by both the creative and editorial teams. This gave supervisors and artists the ability to review individual shots in detail and capture notes for creative feedback. The editorial team would conform the full sequence in Premiere from rendered shots so the team could review the entire sequence with regard to shot lengths and pacing. Due to the time to save files to disk, the rendering/saving process took about an hour and the conforming process took 2-4 hours.

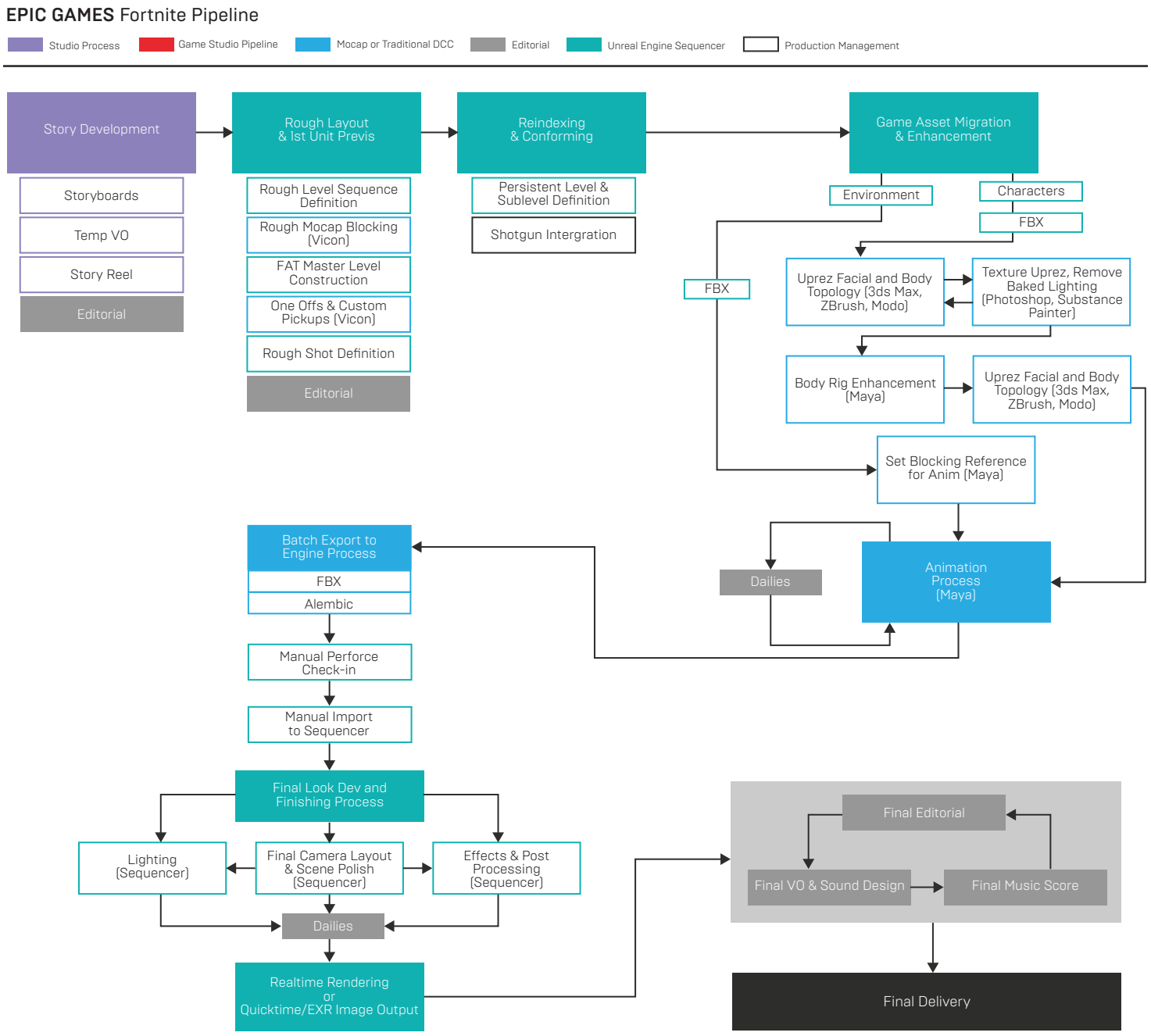


Figure 5: Pipeline for Fortnite trailer

Pre-production

Pre-production

For the pre-production process, Fornite had the advantage of game level assets and rigs that were already in production for the Fortnite video game. Additionally, animation libraries for the husk characters (enemy monsters) were available for use by the animation team along with completed scene assemblies where action took place. As a result, the conceptual design phase, typically required by any type of production, wasn't required and progress could move forward directly into story development and layout.

Story Development

Fortnite started with traditional storyboards to figure out the story content of the piece. These storyboards were scanned and imported into Premiere Pro to construct an offline edit and story reel. The story reel served as an initial shot breakdown and timing resource for the rough layout workflow. New concept art was developed to establish the mood of the scene and detail the look and feel of the set.

Figure 6: In-game character assets pulled for Fortnite trailer

Figure 7: Storyboards

Sequence Breakdown

In breaking down the Fortnite trailer, the pre-production team looked at the animation portion as a three-minute short film, and broke it up into sequences, beats, and shots accordingly.

The animation takes place in three locations (scenes) with two different activities at each location, which made the change of location and activity a logical breakpoint for dividing up the animation into six sequences.

Sequences were not divided into beats or shots at this stage. Instead, the division of shots and beats came naturally through the rough layout process, once the motion was roughed out with mocap.

Rough Layout

To speed up the layout process, the Fortnite team bypassed traditional approaches to rough layout and instead implemented motion capture to block out the rough action, giving give more creative control to the director and cinematographer and saving overall production time.

In a traditional rough layout process, the story reel would be supplied to the Layout Supervisor, and then multiple 3D artists would work within one or more DCCs (digital content creation packages) to block out character movement and camera placement using stepped keyframe animation. Artists normally work for days, perhaps weeks, constructing rough layout shots that are then sent back to editorial for integration into the sequence, with each keyframed sequence replacing one or more storyboarded shots. These shots are then reviewed by the director, and any sequences that require changes are sent back to artists.

The use of Unreal Engine for rough layout presented an opportunity to shortcut this laborious process. Using the Sequencer, motion capture sequences could be quickly imported, reviewed, edited, and re-captured until the director was satisfied with the motion. The time to create and iterate an approved sequence for rough layout was shortened from days/weeks to hours.

The Fortnite team called this approach First Unit Previs, combining filmmaking terms for the principal photography team (first unit) and methods for visualizing complex action prior to actual filming (previsualization or previs).

All motion was captured with Vicon Blade using Motionbuilder, then cleaned up and validated in Maya and exported into Unreal Engine Sequencer.

First Unit Previs

Though more technically intensive, First Unit Previs upends the slower, traditional rough layout approach of manual keyframing. Because the director and cinematographer are free to conceptualize the movie by their own hands, the result is a more natural cinematic approach to the layout process.

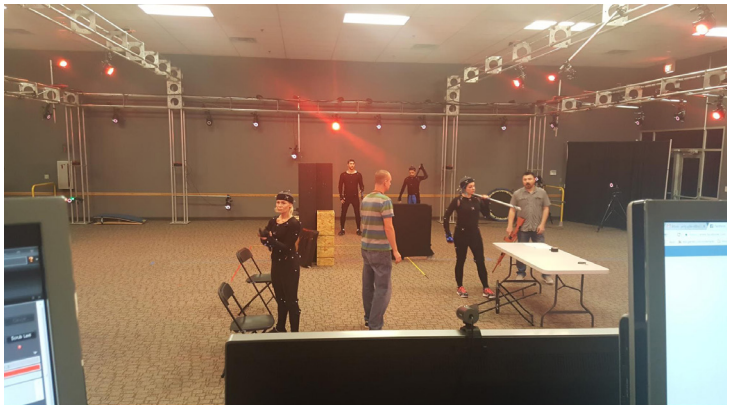


Figure 8: Motion capture session

Prior to the motion capture session, a rough version of each scene was built with placeholders for environmental elements such as the ground and buildings. The storyboard called for a new, crumbling version of the existing Durrr-Burger restaurant, so the initial asset was modified to help describe and inform the narrative.

[2] Block or block out: To determine an actor's movements for a performance. The term originates from early live theater, where actors' stage directions were sometimes worked out by moving around blocks of wood on a miniature stage model, with each block representing an actor. The act of determining actors' movements, and the resulting plan itself, both came to be known as "blocking".

Each scene also included low-resolution versions of game characters and their rigs for testing the mocap action. Having these elements in place at the start kept the time and expense of the motion capture session to a minimum.

For the Fortnite First Unit Previs session, actors acted out the script and their actions were captured through multiple, long master mocap takes. Each take's mocap was put on a character rig within the environment so the performances could be reviewed.

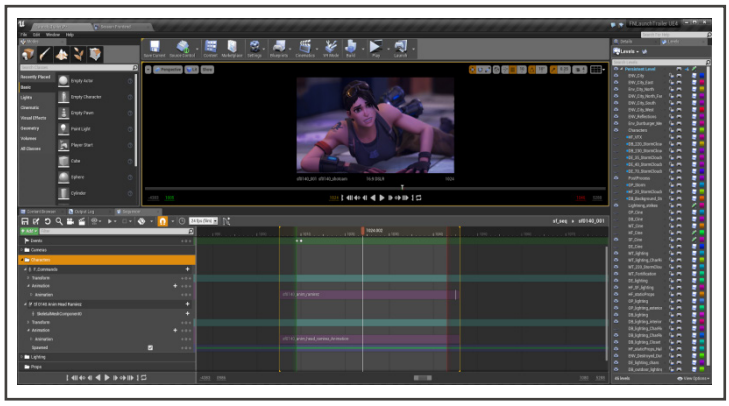


Figure 9: Mocap data on in-game character for analysis

These performances were analyzed to determine which ones were best for the purposes of the trailer. The best long takes became the basis of the interactive rough layout, where the director and others could review, re-capture, and swap out motions until they were satisfied with the result.

Sequencer Steps

Each sequence is imported as a Level in the Sequencer. For game design, a Sequencer Level is designed to hold what you'd expect—an individual level in a game, separate from all other levels. Each level can then be reviewed and edited as its own entity, while still giving access to data and assets from other levels for editing and fine-tuning.

These same tools are ideal for animation production. With each sequence as a separate Level in the Sequencer, sequences can be reviewed and edited separately while still allowing access to data from all levels.

The work sequence for Fortnite First Unit Previs was as follows:

- Mocap actors perform an entire sequence.
- The mocap is put on a character in the rough environment and reviewed by the director until he feels there is a decent take of the entire sequence.
- The best take of the entire sequence, as selected by the director, is imported into the Unreal Engine Sequencer as a Master Level sequence. This long master take is referred to as the Fat Sequence.
- While the motion capture session is still active, the director reviews the Fat Sequence and determines whether portions of it need to be replaced or additional one-offs added.
- For portions that need to be replaced, either portions of other versions of the sequence are selected, or the mocap actors perform the required retakes.
- The replacement take is imported into the Sequencer

The review/replace process was repeated for each sequenced level until the director and the team were satisfied with the rough layout. From this process, shots and beats naturally emerged as the director crafted the sequences.

Note: Epic captured camera positions during Mocap, but camera positions can also be blocked in Unreal Engine with cine cameras if they aren't captured during the Mocap process.

Rough Layout Cleanup

During motion capture sessions and director review, levels and sequences were created quickly and loosely to keep up a brisk pace and finish motion capture as quickly as possible. Very little attention was given to finer administrative tasks during this fast-paced step.

Once rough layout was approved, a second step of organization was required: re-indexing shots, redefining levels, and conforming sequences into a more orderly fashion.

Defining Levels

In Unreal Engine, every object that the viewer sees or interacts with resides in a Level. A Level is made up of a collection of meshes, volumes, lights, scripts, and more all working together to bring the desired visual experience to the viewer (or player, if creating a game). Levels in UE4 can range in size from massive terrain-based worlds to very small levels that contain just a few elements.

Levels are created and organized within the Sequencer's Level Editor. By default, the Level Editor always contains a Persistent Level. This is the base level for the sequence—anything placed directly on this level will persist through all sublevels.

It is common to leave the Persistent Level itself empty, and establish sublevels under it for scene assembly environments, specific sequence- or shot-based effects, routinely used characters that are common across sequences and shots, environments and props, light rigs, Blueprints, and post-processing devices.

Sublevels could be thought of as performing the same function as layers in a DCC. When a sublevel is made current, all work done is added to that sublevel. In addition, sublevels can be hidden or unhidden at any time.

Levels have the additional feature of providing a master animation track that can be referenced into a DCC such as Maya.

Fortnite's level organization included:

- Persistent Level
  - Environmental sublevels
  - Character sublevels
  - Cine sublevels
  - Lighting sublevels
  - Blueprint sublevels[3]
  - Post Processing sublevels

For additional information on defining sublevels, adding or moving actors, visibility settings and streaming methods, please see Managing Multiple Levels in the Unreal Engine documentation.

[3] In Unreal Engine, a Blueprint is a container for a script created with visual tools (node relationships). Blueprints are used to control in-game or cinematic action without the need for coding. For example, in a game, a Blueprint might hold a trigger to open a door when a character approaches.

Level Sequences

Motion and animation within a Level is defined through a Level Sequence made up of Tracks. A single Track could contain just the character animation, transformations (object movement), audio, or any number of other discrete elements of a sequence. A Track can also contain an entire shot.

In Unreal Engine, a Level Sequence is treated as an individual object within its Level. A Level Sequence can be likened to a container for a cinematic scene—basically a movie editor contained inside an object.

Figure 10: Add Level Sequence option

Because Level Sequences are self-contained assets, a Level Sequence can be embedded within another Level Sequence.

By default, a Level Sequence is built from the top down—tracks lower in the Level Sequence hierarchy take precedence. This allows filmmakers to build a pipeline they are accustomed to, where adjustments at the shot level override the Sequence they are contained in.

Fortnite Level Sequences

The Level Sequences for the Fortnite trailer followed this general structure:

- Top Level Sequence for entire trailer
- Sequence 1 Level Sequence
  - Shot A Level Sequence
    - Tracks for the shot (camera, lighting, characters, shot-specific audio, etc.)
  - Shot B Level Sequence
    - Tracks for the shot
  - (additional shots and tracks)
  - Visibility track for Sequence 1
- Sequence 2, 3, etc. broken out in the same way
- Audio main track
- Fade track
- Visibility track for Top Level Sequence

Note that the top level sequence has a visibility track, as does each sequence Level Sequence. In addition, Audio and Fade tracks are under the top level to allow control of these elements throughout the entire trailer.

The structure for the Fortnite trailer's Level Sequences is shown in the following diagram, showing the first sequence and its first shot broken out into tracks.

A master FILM Level Sequence named Launch Trailer contains all six of the trailer's sequences, each with its own Level Sequence. Each of these Level Sequences contains shots, with a Level Sequence for each shot. Inside these shot Level Sequences are tracks for character animation, VFX, lights, camera, etc.

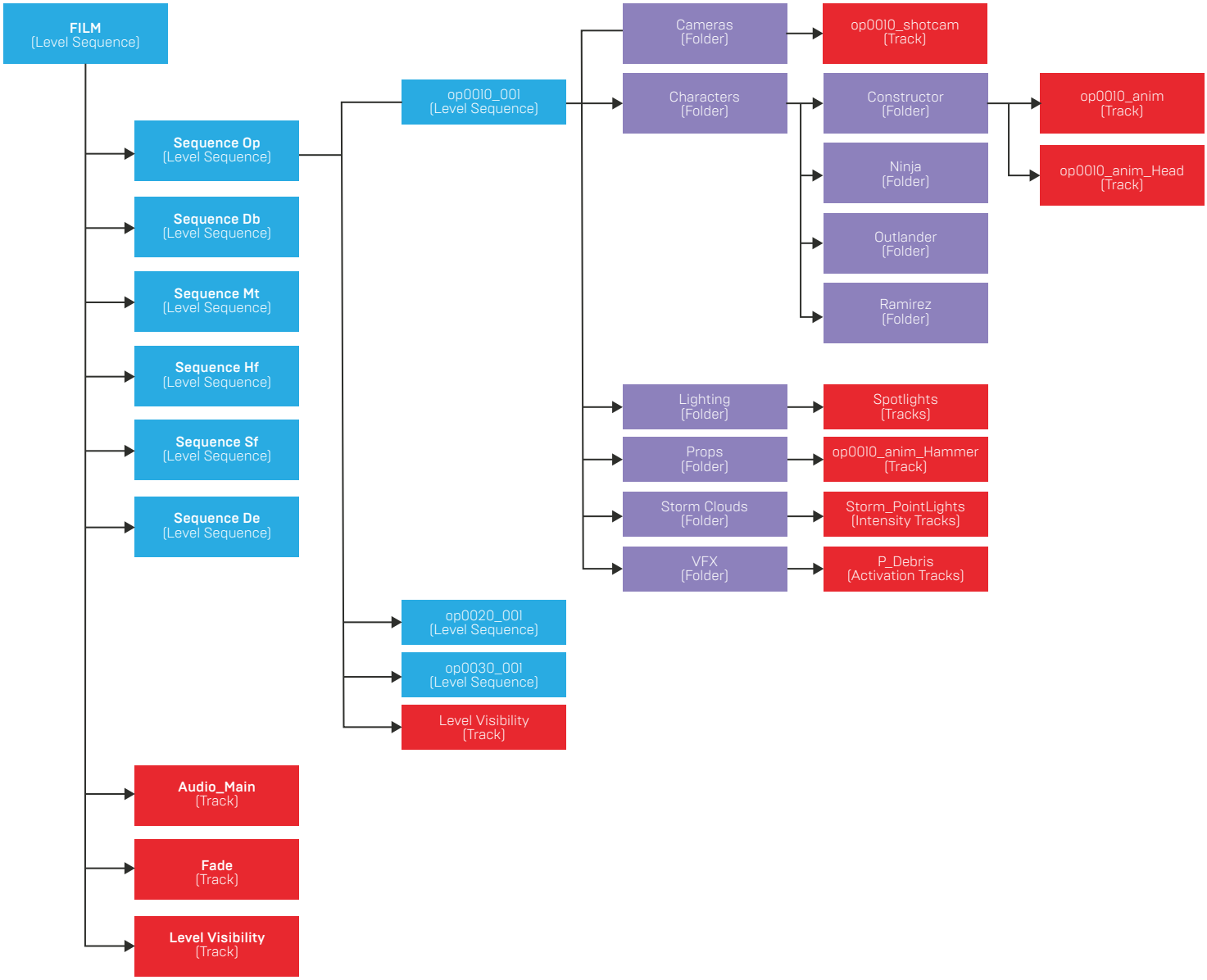


Figure 11: Level hierarchy for Fortnite trailer

File and Folder Organization

File organization goes hand in hand with setting up tracks. Each track should have at least one corresponding folder to hold all the data for that track.

The folder structure Epic used for Fortnite's tracks is shown in the diagram. While many tracks utilized a single folder, the animation tracks accessed several character folders.

Naming Conventions

- Top Level Sequence (FILM): *Launch Trailer*
- Sequence Level Sequences: *OP_Seq, HF_Seq*, etc.
- Shot Level Sequences: *OP_0010, OP_0020*, etc.

A naming convention that utilized short names made it easier to manage shots and sequences in Unreal Engine.

Another useful method of ensuring organization of the trailer was to assign official shot numbers to the trimmed motion capture takes and their respective level sequences. The shot numbers followed a traditional film style approach by combining a two-letter sequence designator with a four-digit shot number followed by take number, for example *db0010_001*.

Level Visibility

Level visibility is an important concept within Unreal Engine. As shown in Figure 8, the FILM level sequence has a Level Visibility track, and each Sequence also has its own Level Visibility track within that Level Sequence.

External scenes are referenced directly through the Level Visibility track. When a level becomes visible, this triggers the loading of these external scenes and any shots under that Level Sequence.

Although a Level Visibility track can be used in any place where tracks can be added, Epic opted to use them only at the FILM and Sequence levels. In general, shots don't need Level Visibility tracks, as they should always be visible within the sequence. If something within a shot needs to appear or disappear, a per-object Level Visibility track would be used.

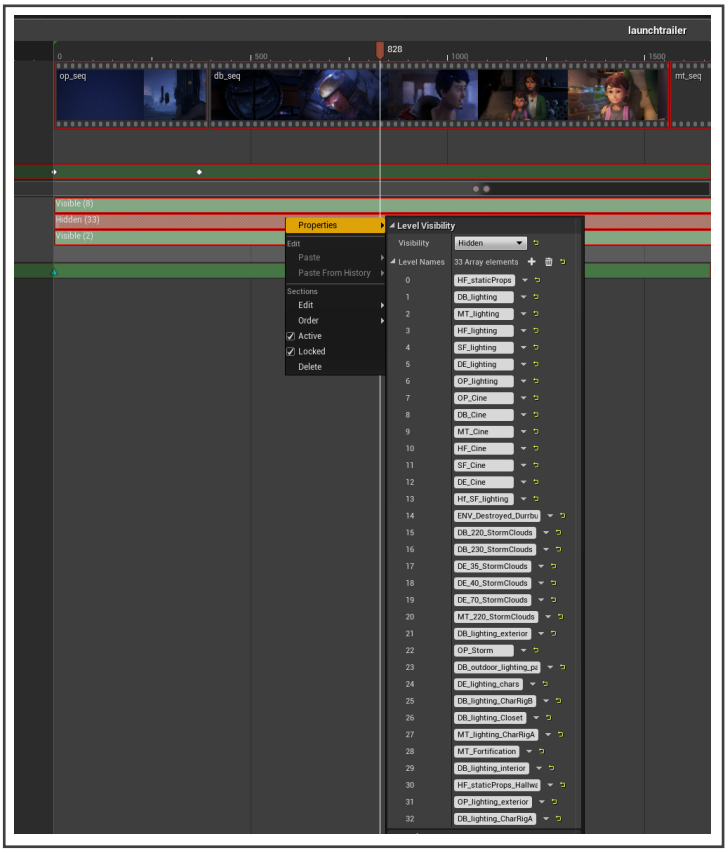


Figure 12: Level Visibility for current sequence

Level Management

The level location for an asset can be checked from the World Outliner window by clicking the arrow on the top right.

If the Sequencer window is open, the World Outliner will also show which Sequencer(s) the object is associated with. The World Outliner shows all the items in a scene in a hierarchical view.

While playing in the FILM Level Sequence; objects might disappear and reappear in the World Outliner. This is normal operation as it reflects the state of Level Visibility.

For additional information on Level Sequence and the Sequencer Editor, see Sequencer Overview in the Unreal Engine documentation.

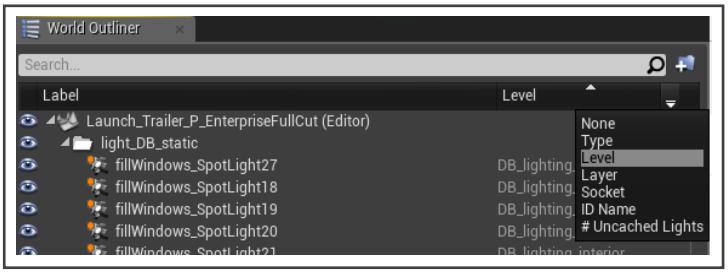


Figure 13: World Outliner

Production

Production

Scene Assembly

While modeling, rigging, and animation for each asset in the Fortnite trailer was done with a DCC package, the final assembly of each scene was done in Unreal Engine. This approach differs from a traditional CG pipeline, where environments are assembled within the DCC package itself. By finalizing each asset separately within a DCC package and aggregating them in Unreal Engine, the team was able to work on characters and environmental elements in parallel.

Transfer of data between DCCs and Unreal Engine would be accomplished with either of two file formats.

- FBX – For transfer of models and editable/animatable rigs to the Unreal Engine Animation & Rigging Toolset (ART)
- Alembic – For transfer of complex animation as baked morph targets

While both Alembic and FBX can be used for exchanging information between DCCs and other pipeline programs, they store data differently. The Unreal implementation of an Alembic file stores data as "baked" information—the animation of a character, for example, is stored as a series of vertex positions, basically a set of morph targets with no accompanying rig. Conversely, FBX retains information about rigs, their associated models, and skinning settings. In other words, a rig transferred to Unreal Engine via FBX can still be edited in a DCC, while Alembic files don't retain this functionality.

Improvements to In-game Assets

Given the importance of making a significant improvement in quality between the Fortnite game and trailer, Epic needed to make improvements to existing in-game models, textures, and rigs to improve the quality of the final animation. At the same time, the final version needed to remain lightweight enough memory-wise to allow real-time playback.

Animation Tests

As a first step in determining the improvements needed, the Fortnite team tested existing in-game models and rigs against the mocap animation supplied by First Unit Previz. These original rigs had been created with Unreal Engine's Animation & Rigging Toolset (ART). ART works in conjunction with Maya, operating within Unreal Engine but calling various rigging functions from Maya as needed.

In reviewing the animation tests, the team quickly found that improvements were needed in two areas:

- Body rigs – Model resolution would need to be increased to support body deformation. The existing rigs for in-game characters would provide a basis for the trailer characters' rigs, but would need to be augmented in ART.
- Facial rigs – Existing facial models and rigs would be insufficient for the level of fidelity the animators wanted. In addition to rebuilding character heads at a higher resolution, and a different rigging/animation workflow would need to be implemented to support real-time playback with the higher quality level. Facial rigs would be created directly in Maya to leverage Maya tools not available with ART, then exported via Alembic to utilize the greater per-vertex control that Alembic offers.

Model Resolution

Epic determined that for the Fortnite trailer, they would need to use higher-resolution models than would normally appear in a game world, not only for characters but for environmental elements. While higher-resolution models improve the appearance of an animation, increasing model resolution introduces additional considerations:

- While tools exist to increase resolution automatically, new polygons must be adjusted and tweaked manually to get the maximum benefits.
- Texture resolutions need to be increased to support the new geometry.
- Increased resolution also increases the amount of memory required to play back the animation. Care must be taken not to increase resolution so much that playback would suffer.
- Character rigs must be adjusted or replaced to leverage the new polygons and improve animation fidelity.
- Deformation of high-resolution characters during the trailer animation could be quite different from in-game deformation, meaning rigs and keys that worked on the low-resolution model might need to be adjusted.

Modeling

To create the models used for the Fortnite trailer, existing in-game assets were exported to 3ds Max, ZBrush, Modo, and Maya for further work.

Character Models

To begin work on character models, artists separated each asset into separate meshes. Technical Animators advised on how the characters should be broken down to best facilitate rigging and animation.

Resolutions of all body model parts were increased to help with deformations and physics.

Figure 14: Separating character head, body, clothing

Because of the new rigging and animation approach for the heads, character head geometry was swapped out with high-resolution universal topology. Although each character retained its unique appearance, the topology itself was consistent from one character to the next. This approach saved time in rigging and animation, as similar rigs could be used for all characters.

Asset resolution for each character was around 185,000 triangles.

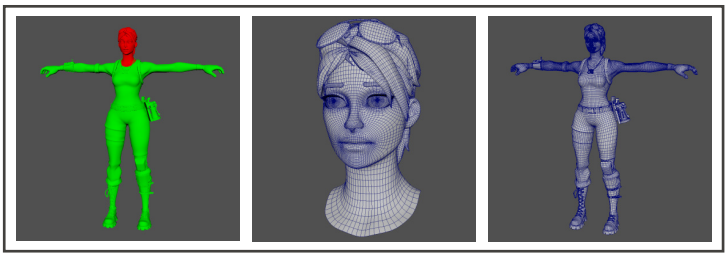


Figure 15: High-resolution characters and heads

Environment Models

Epic's objective for this project was to expand on the world and environments of Fornite while ensuring the piece would still run in real time.

For items in the environment, the Fortnite team started by increasing model resolution of simple assets to define the level of fidelity required, and to inform the direction for the remaining environmental assets.

Environment assets were exported back to Unreal Engine via FBX.

Materials and Textures

In-game materials and textures were exported along with models via FBX. Some textures included baked lighting, which had to be removed for the Fortnite trailer.

Many textures needed increased resolution to work with the improved models. The Fortnite team used Substance Designer to improve existing textures based on the new topology.

Substance Designer provides a vast improvement over the traditional technique of creating multiple unique 2D bitmaps. Substance Designer can use both bitmaps and vector graphics as a base for materials, and includes tools for importing mesh information, adding ambient occlusion, and propagating changes at the base level throughout several materials.

Once the textures were complete, body textures were exported from Maya via FBX format. Because the facial rig and animation required Alembic export, facial textures were exported via the Alembic format.

Materials and textures must be set up in a certain way in Maya to export properly into Unreal via Alembic. In Maya, the material shading group name must be assigned to the geometry's component-level faces to match the material name in Unreal Engine.

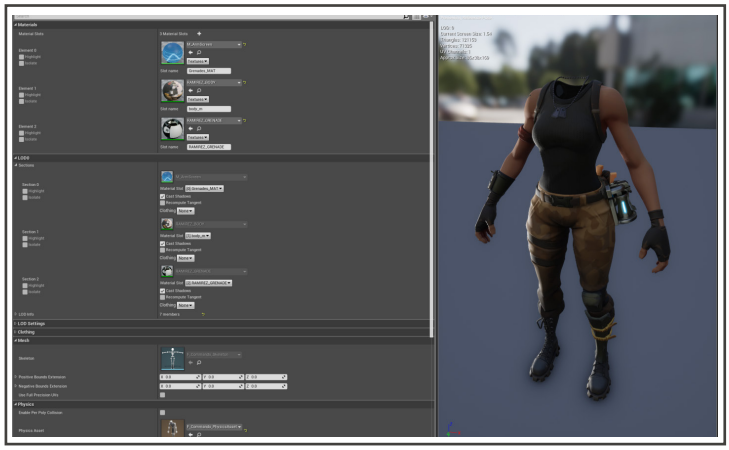


Figure 16: Assigning materials to a character's body

For FBX export, the Maya Material Node is assigned to object level geometry.

Figure 17: Matching names of shading group nodes with materials in Unreal Engine

For both Alembic (Shader Group) and FBX (Material Node), the material must be assigned at the face level in Maya rather than the mesh level.

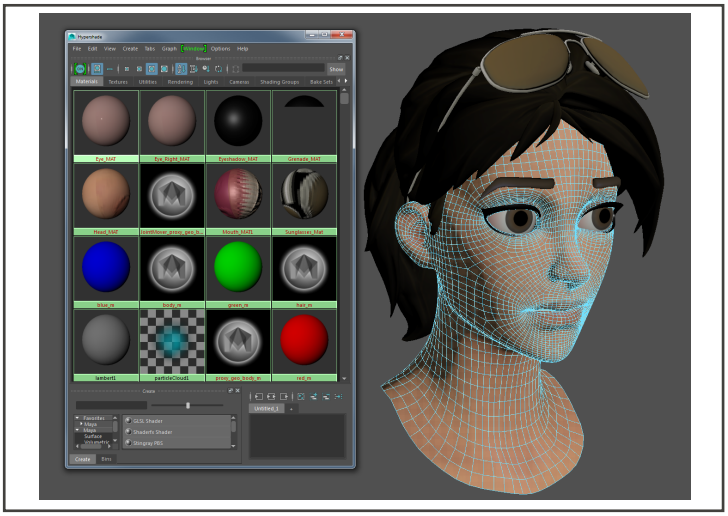


Figure 18: Facial materials in Maya

Dealing with Real-Time Considerations

Throughout the environment, Epic needed to be conscious of not going overboard on triangles and textures so the Fortnite trailer could maintain a solid real-time performance. With the combination of new high-resolution assets and the world continuing to be built out to support the creative process, the Fortnite team quickly became bound by the number of objects in the scene.

The team optimized the set by removing any objects from the initial game environment that didn't contribute to the trailer's narrative. In addition, once final cameras were in place, they removed backs of buildings and anything else that wasn't in a direct field of view.

These creative and technical decisions allowed the Fortnite team to produce a stunning visual environment that helped drive and support the stylized narrative. In doing so, it enabled Epic to push the boundaries of what's possible in a real-time format.

Rigging and Animation

Because of the decision to use a Maya/Alembic workflow for facial rigs rather than ART/FBX, different workflows were used for body and facial rigging.

Body rigs were created in-house with ART. Due to time constraints, facial rigs were outsourced to a third party who created custom rigs per character. They imported the ART body rigs, added the custom face rigs, and sent the new, combined rigs back to Epic.

Once the combined ART and custom face rigs arrived back in-house, they were shipped to an outside vendor for animation in Maya. The vendor received both the rigs and the layout files of the shots in the cinematic from Epic.

When the animated sequences were complete, their Maya files and animation movies were shipped back to Epic and imported into Unreal Engine. As the Fortnite team reviewed each sequence, in-house animators added and edited motions based on the changing cut.

Body Rigs

While the original in-game rigs were mostly sufficient to support the needs of the Fortnite trailer, leaf nodes were added to the body rigs to provide more complex dynamics and help with deformations.

With a rig generated from ART, the joint/deformation hierarchy is automatically separated from the rig control structure, making it easier to export the rig and its animation to Unreal Engine afterward. This general design can also be implemented manually within Maya, even if ART is not being used for rig construction.

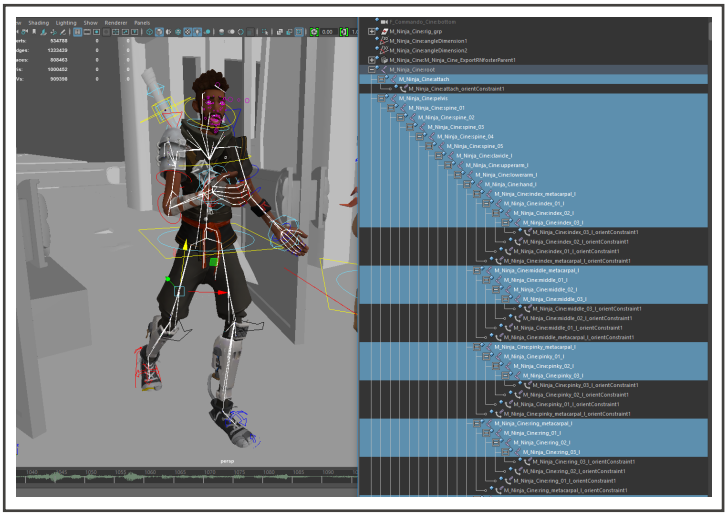


Figure 19: ART rig with separate joint and control hierarchies

Body Animation

Body performance was a combination of mocap from the rough layout stage and keyframed animation.

The environmental blocking placeholders constructed during First Unit Previs were exported out of Unreal Engine as FBX files. When imported into Maya, the result was a simple, gray reference geometry that provided animators with a visual guide to animate against.

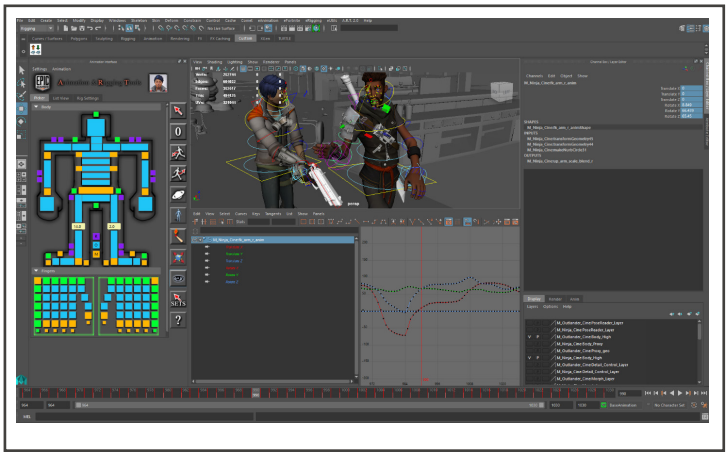


Figure 20: Reference geometry in Maya

Animators used standard animation techniques to refine body animation for their shots, using the ART tools to help simplify their workflow. When animation was complete, the animation was exported back to Unreal Engine via FBX.

Animation & Rigging Toolset (ART)

In addition to an interface for skeleton creation, skeleton placement, and rig creation used to generate the initial rig, Unreal Engine's Animation & Rigging Toolset (ART) is a full suite of animation tools and user interface including a body picker, options for importing mocap or animation data and exporting to FBX, pose tools, mirroring options, space switching settings, and more. Below are descriptions of the three tabs used by animators as well as the side gutter of buttons for various tools.

Body Picker

The **Picker** tab displays a body picker where an animator can quickly select a body part to get access to that part of the rig. Animators at Epic use the ART body picker to help them animate the various characters on all of Epic's internal projects.

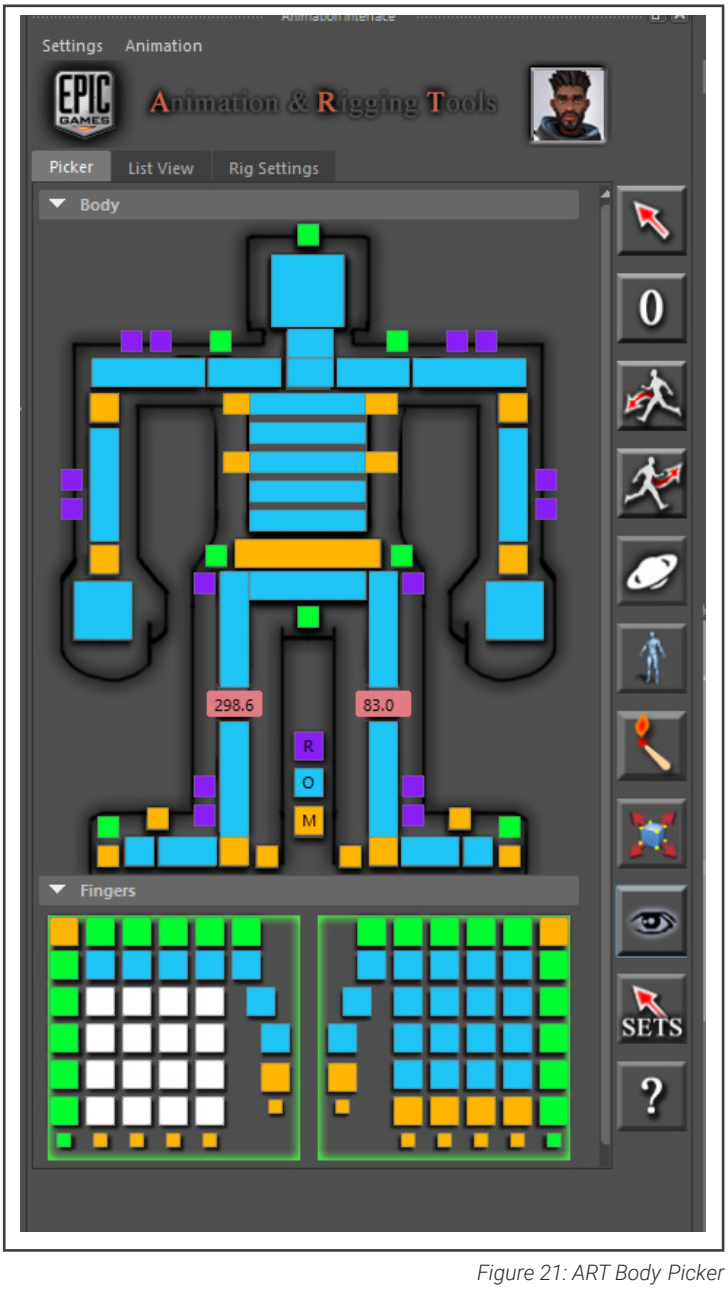


Figure 21: ART Body Picker

List View and Rig Settings

The **List View** and **Rig Settings** tabs include tools for selection, import/export, space switching, rig visibility and set creation utilities.

Figure 22: ART List View and Rig Settings

Pose Editor

The **Pose Editor** saves and loads custom poses.

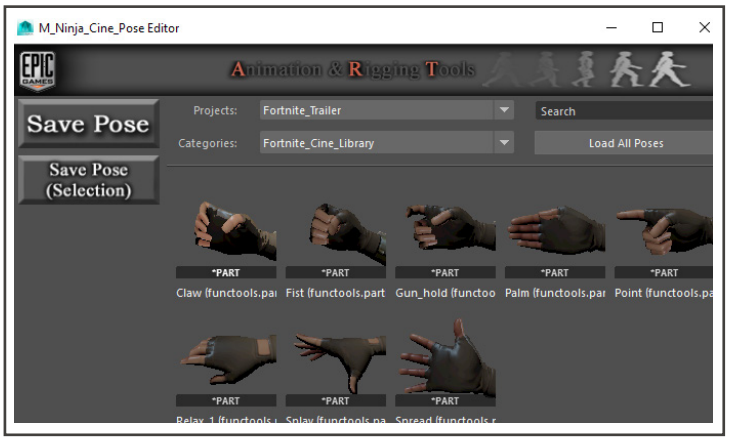


Figure 23: ART Pose Tool Window

Facial Rigs and Animation

During the animation tests, the Fortnite team determined that a new workflow was needed for facial rigs and animation. The team decided to use Maya rather than ART for rigging, and to export the data via Alembic rather than FBX.

For the Fortnite trailer, the team used Maya lattices for facial deformation. A lattice is a virtual cage around a set of vertices. The points of the lattice can be pushed and pulled to deform the lattice into different shapes, and the vertices inside the lattice follow along to a greater or lesser degree depending on their proximity to the part being deformed.

Lattices are useful for animating groups of vertices in a subtle and natural way. Since ART doesn't support lattices, the team needed to use Maya directly to implement this approach.

The team also wanted to use blend shapes, another Maya feature not supported by ART. Blend shapes are different versions of the same model, in this case several versions of a character's head with different facial expressions. When the rig is animated, the facial shapes blend from one expression to the next.

The final facial rig was a combination of 201 joints, lattices, and blend shapes. All characters' heads had universal topology and the same facial rig, making it possible to share rigs between characters.

Blend shapes can be exported/imported via FBX, but deformers, including lattices, cannot. Then the team encountered another challenge that informed their decision to not use FBX export for facial animation. When setting up a rig, each vertex can be influenced by several joints, blend shapes, and lattices. At the time of the Fortnite trailer's production, FBX format was able to export up to eight influences per vertex, but the team needed more than that to get the facial performance they wanted. Alembic can export unlimited influences per vertex, making it a better choice for facial rig and animation export.

The face rigs consisted of weighted joints constrained to

nulls (handles) which have blend-weighted set-driven keys, giving animators the ability to control eye movements and facial muscle groups. An upper layer of tweak controls grouped the individual controllers for easier animation of broad motion, such as opening the jaw.

The facial rig provided on-surface controls for animators to move as needed, as opposed to a virtual control board.

All facial performance was manually keyframed in Maya.

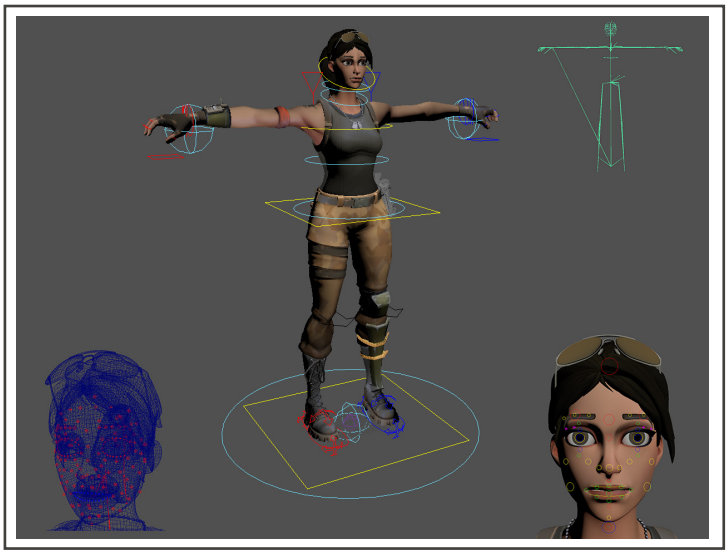


Figure 24: Facial rig in context with body rig

Figure 26: Facial performances

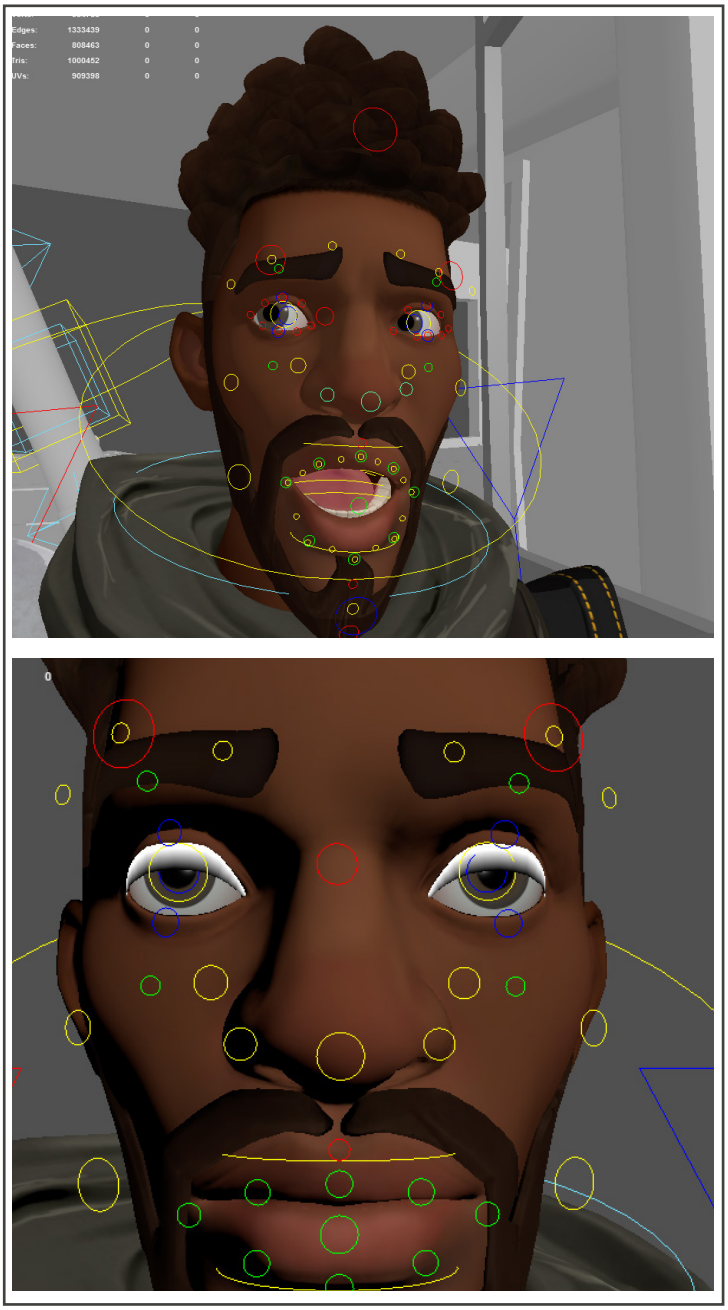


Figure 25: On-surface facial controls

Exporting to FBX and Alembic

During the process of fine-tuning animation, each time an in-house animator made an update to a body or prop animation in a Maya scene, an FBX file encompassing the changes was exported. If a change was made to facial animation, the changes were exported via Alembic format.

There are a few ways to perform the export using Maya's internal commands or the FBX exporter in ART tools.

FBX Export from Maya

Even though FBX export is an option in ART, this demonstration shows the actual Maya settings, and will use standard Maya operations to better illustrate what is happening under the hood.

Before FBX export, animation must be baked on to joints to remove the controls and constraints so only the joint animation is passed to Unreal Engine. To do this, select all the joints for the character and choose **Edit > Keys > Bake Simulation** from the Maya menu. The separate rig/deformation and control hierarchy mentioned earlier makes this step easier.

Click **Bake** to bake the joints.

Select the joints and export to FBX over the desired frame range.

For more information, see the FBX Animation Pipeline topic in the Unreal Engine documentation.

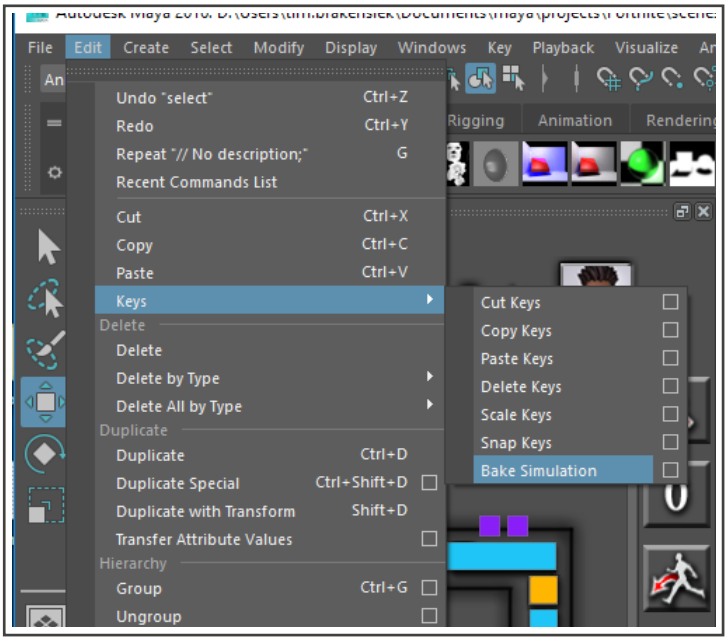


Figure 27: Bake Simulation menu option

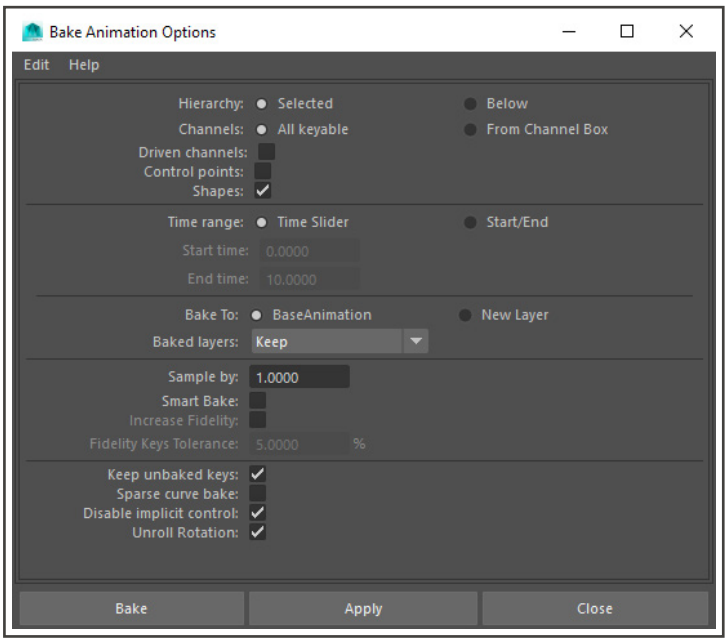


Figure 28: Bake Animation Options dialog

FBX Export from ART

This demonstration uses the shot DB0137 from the Fortnite trailer to visually demonstrate the steps for FBX export directly from ART.

With the scene and rig loaded, click the **Export Motion** button. This opens the Export Motion dialog.

On the Export Motion dialog, select options and click **Export FBX**. This automatically creates a copy of the rig and bakes all the motion to it before creating the FBX file, performing the same steps that are required manually when exporting from Maya.

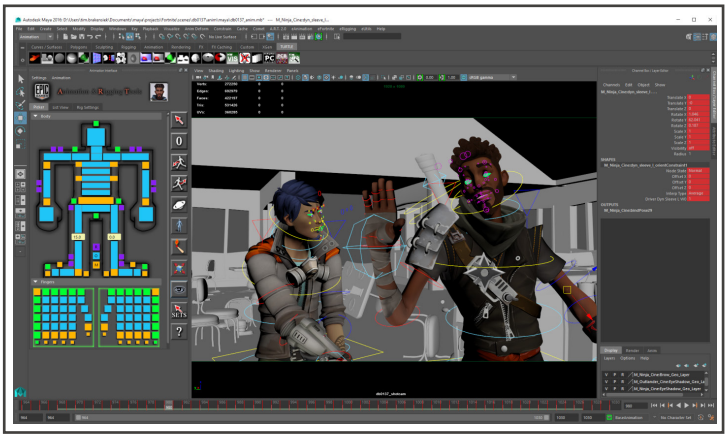


Figure 29: Scene loaded in ART

Figure 30: Export Motion button

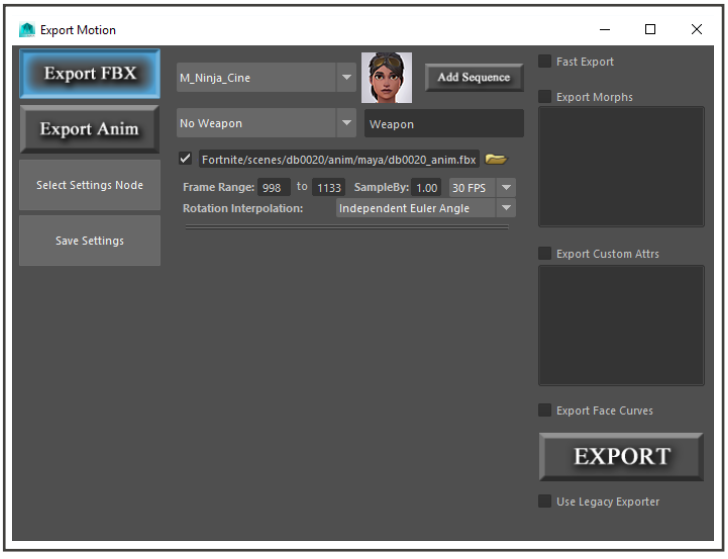


Figure 31: Export Motion dialog in ART

Alembic Export from Maya

To export the deforming facial animation in Maya to Alembic for use in Unreal Engine, the facial meshes first had to be triangulated in Maya with the Triangulate tool under Mesh.

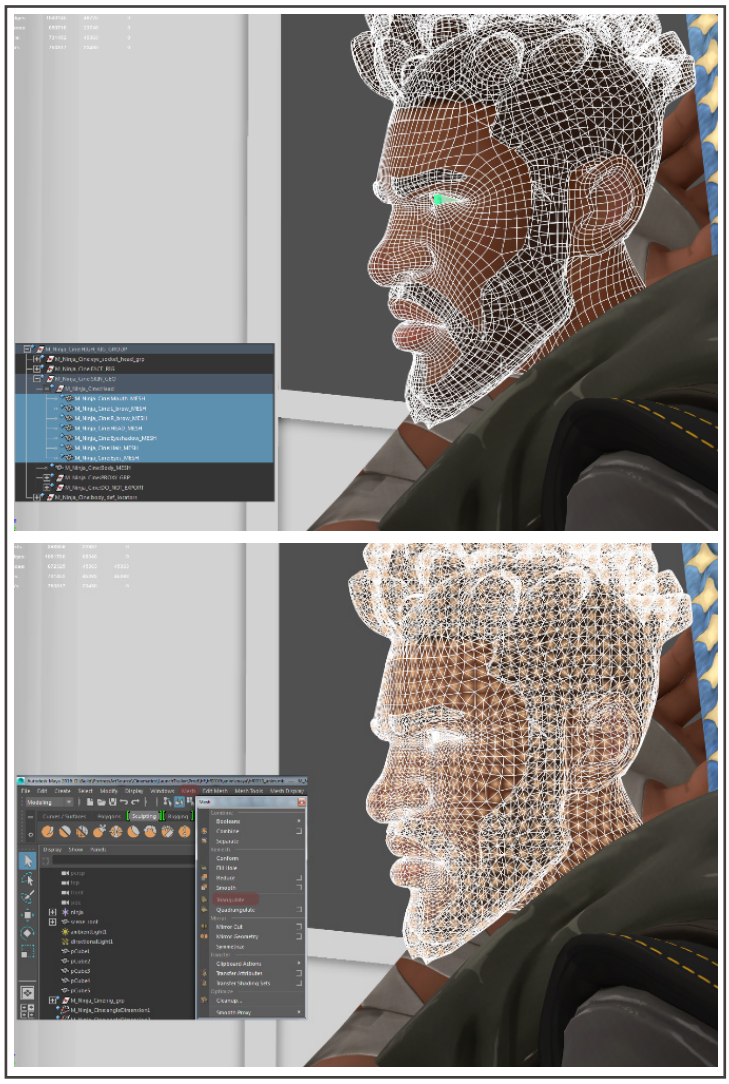


Figure 32: Facial mesh before and after triangulation

To export to Alembic format, select the meshes to export and choose **Cache > Alembic Cache > Export Selection to Alembic** from the Maya menu.

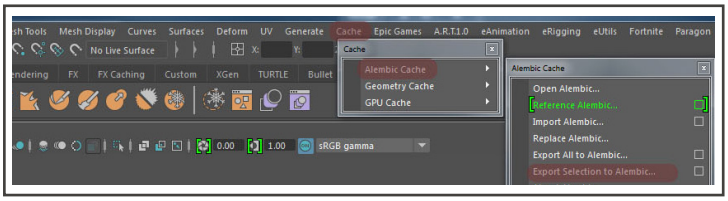


Figure 33: Export Selection to Alembic

This opens the Options dialog with export settings.

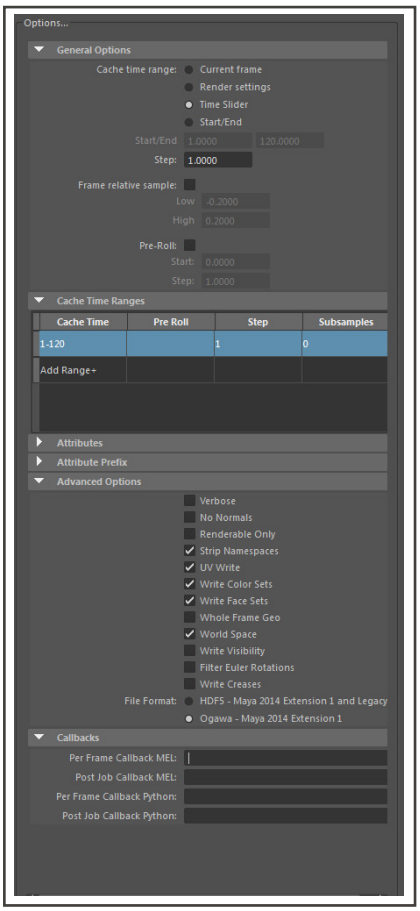


Figure 34: Options dialog for Alembic export

Custom Alembic/FBX Export Tool

Having to export to two different formats quickly became cumbersome for the animation team. To solve this problem, Epic developed a tool to export from Maya to both FBX and Alembic simultaneously from a single interface.

This tool, called Cinematic Exporter, is not currently included as part of the ART toolset. However, Epic was able to develop this custom tool quickly in the heat of a fast-paced production, further shortening the Maya-to-Unreal pipeline. It is included here to illustrate how a small custom solution based on a pipeline's specific needs can leverage Python within Maya to shorten production time.

Cinematic Exporter launches from a pull-down menu in Maya. The left side of the interface populates a list of all the exportable nodes it finds in the scene. To export, select the nodes for which data needs to be exported.

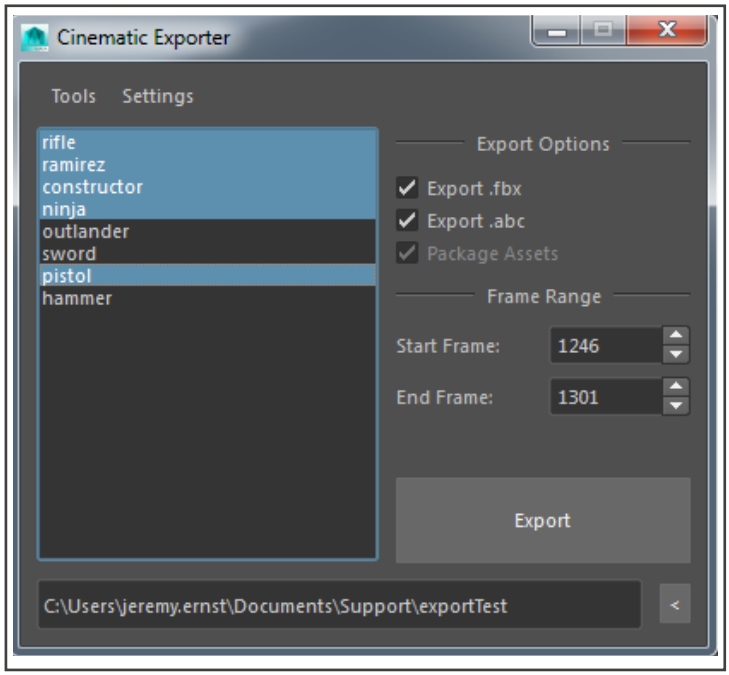


Figure 35: Cinematic Exporter node selection

By default, both FBX and Alembic (ABC) files are exported, but either file type can be selected alone. The frame range is automatically set to the timeline's current active frame range, but can be changed.

An output path has to be set for the export to initialize. For the Fortnite workflow, this was a location in Perforce.

Clicking the **Export** button launches a Maya process to export the data using the settings described in the FBX Export from Maya and Alembic Export from Maya sections. The process runs in the background, meaning the artist can still use Maya while export is taking place.

The exported data is placed in named folders within the output folder for easy location for import.

To give Cinematic Exporter the ability to recognize exportable rigs, the technical animators/riggers at Epic created a scripted node, a dummy object with export data attached to it, and placed one of these nodes on all the scenes that included rigging. When Cinematic Exporter is launched, it populates the list with exportable heads, characters and props based on the export nodes it finds in the scene. Export nodes were designated as specific to Alembic (for heads) or FBX (for bodies/props) using a custom tool.

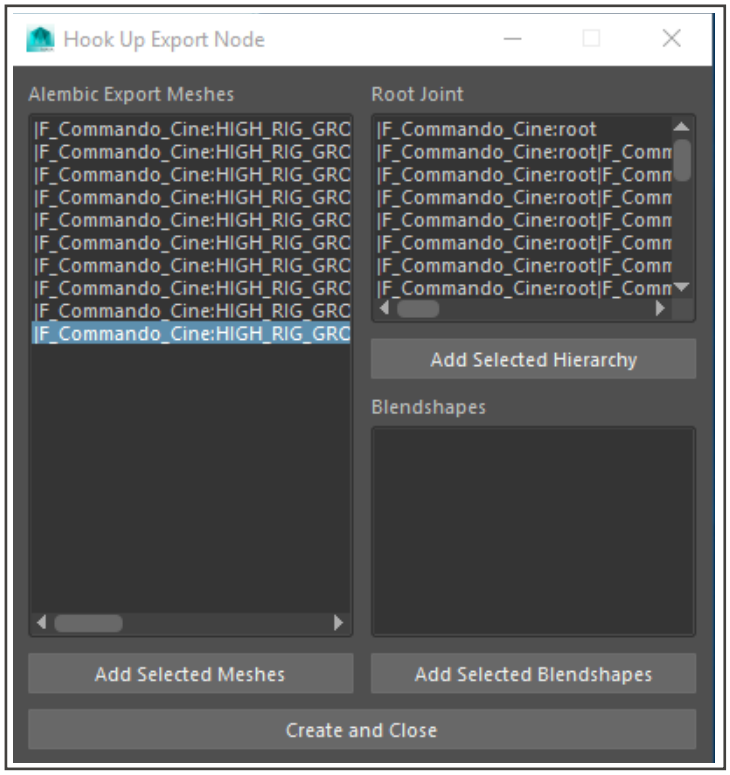


Figure 36: Utility to set Alembic and FBX export nodes

The exported data is organized into folders for easy import.

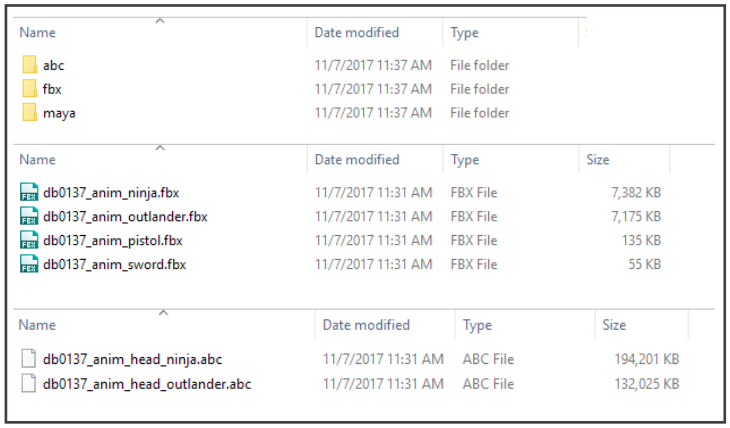


Figure 37: Organization of exported files

Importing to Unreal Engine

The final step is to import both the FBX and Alembic files (ABC) into Unreal Engine.

FBX Import to Unreal Engine

FBX files for body and prop animations were imported to Unreal Engine using its FBX import tool.

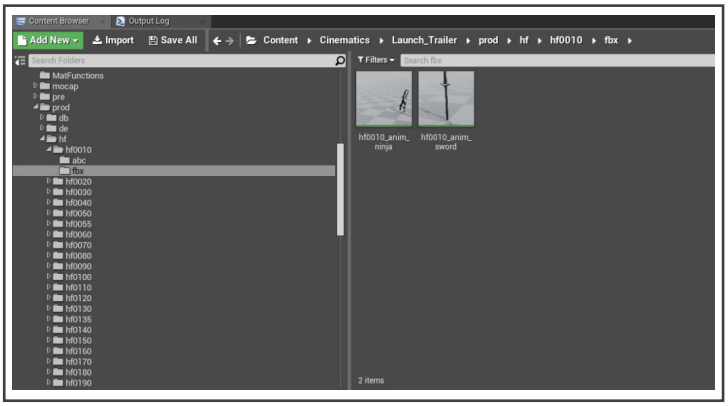


Figure 38: Highlight FBX folder before choosing Import option

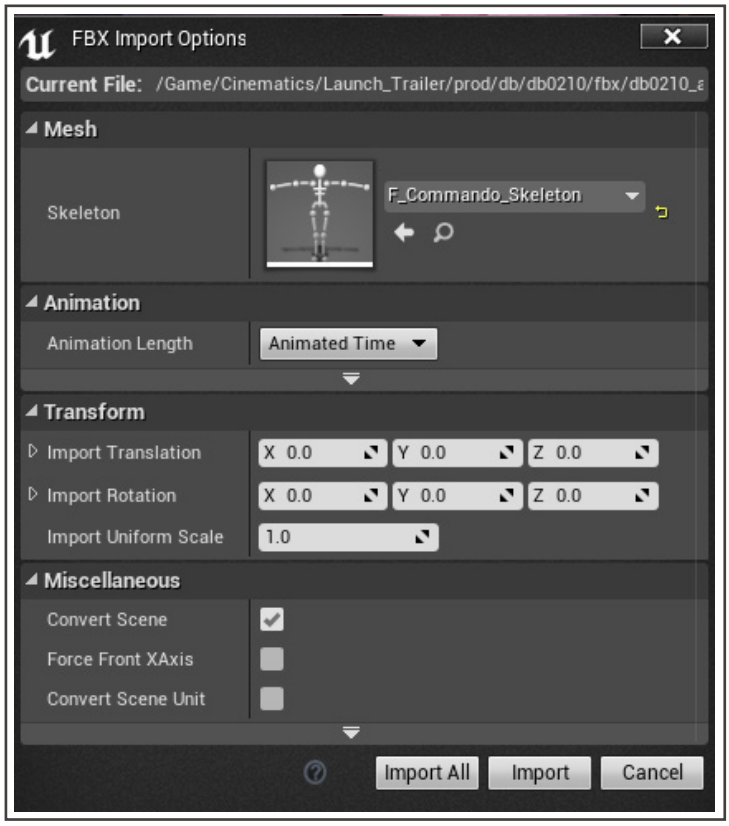


Figure 39: Unreal Engine FBX Import Options dialog

Alembic Import to Unreal Engine

Alembic files for facial animation were imported to Unreal Engine using its ABC import tool. The data is imported as morph targets using PCA compression.

Alembic files were imported with the Skeletal option.

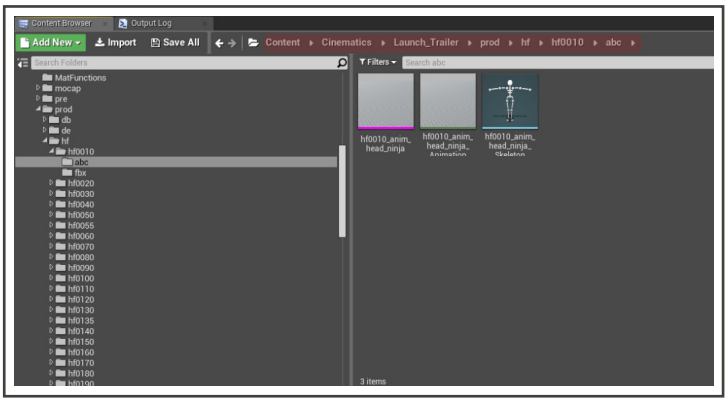


Figure 41: ABC folder selected before starting import process

Figure 40: Imported animation applied to character

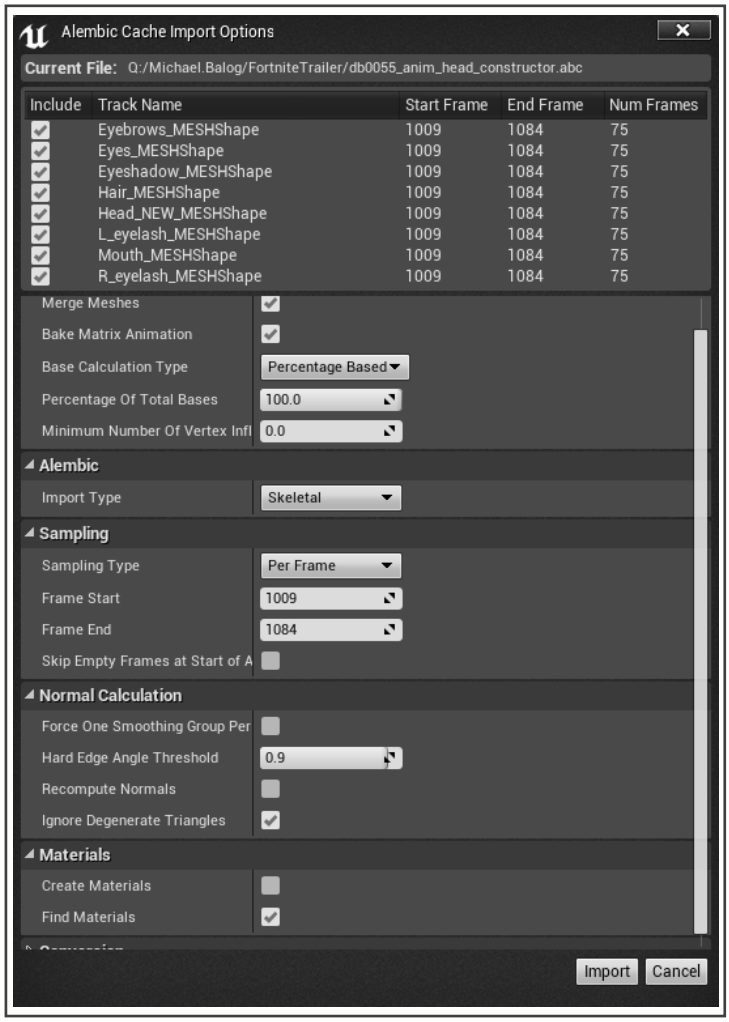


Figure 42: Alembic import options

The morph targets generated by the Alembic importer can be viewed in the Morph Target Preview window.

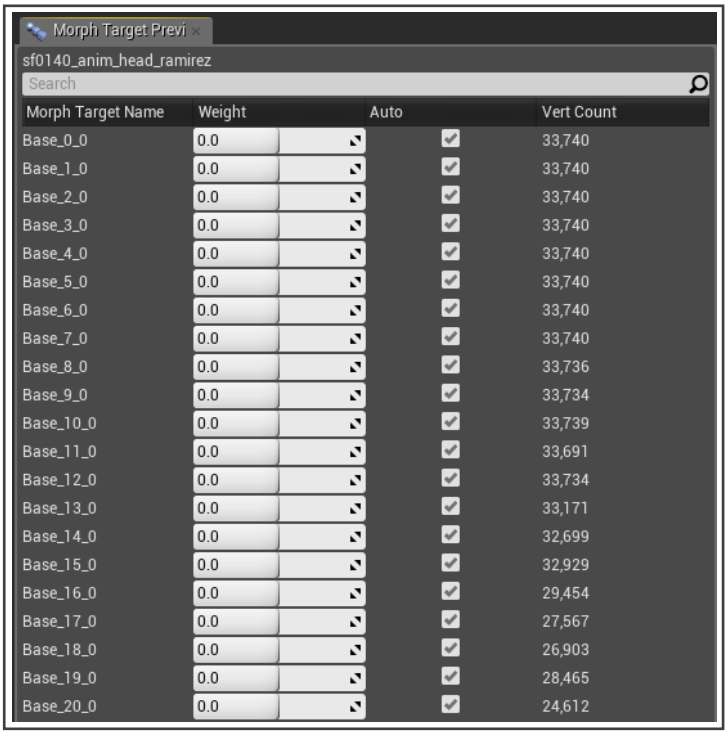


Figure 43: Morph Target Preview window

Morph target playback is an intensive operation that works best when GPU processing is used. In preparation for using the morph targets in real-time playback, the **Use GPU for computing morph targets** option should be turned on. To access this option, from the File menu choose *Edit > Project Settings > Rendering > Optimizations*.

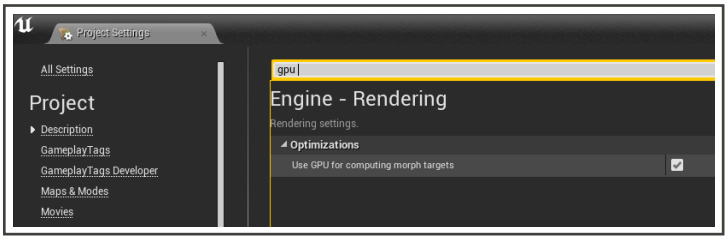


Figure 44: Use GPU for computing morph targets option

PCA Compression

When an animation is exported via Alembic, the ABC files stores the position of every vertex on every frame. For a model with over 150,000 triangles in a two-minute animation, the amount of data can add up very quickly. Analyzing and playing back such a large volume of data in real time isn't feasible.

Instead, Unreal Engine imports and stores Alembic animation data using Principal Component Analysis (PCA) compression to reduce the amount of data yet still keep animation quality high. During import, poses are extracted and weighed to determine the "average" pose and other poses' differences from the average. These differences are analyzed to determine which frames would make the best morph targets, and only the data from these frames is stored. In effect, the process distills the enormous amount of vertex-based animation data to a smaller, more manageable data set.

During playback, Unreal Engine loads the morph target data into memory and blends them per-frame in real time. In this way, the Fortnite facial animation exported via Alembic format could be played back in real time in Unreal Engine. AnimDynamics, another UE feature, was also used to enhance the characters' secondary animation detail such as hair and clothing movements, all in real time.

Lighting

Originally, Fortnite intended to implement a baked lighting solution to increase the playback performance of the trailer. Many of the existing game assets that were utilized and enhanced for the trailer already had a baked lighting solution within their texture maps. However, baked lighting proved to be too restrictive for directors who wanted to adjust lighting more intuitively.

Fortnite chose to forego baked lighting in favor of the increased flexibility of dynamic lights with cast shadows. This allows shot-by-shot adjustments of lighting values and set dressing placement.

Figure 45: Real-time playback of Alembic facial motion in Unreal Engine

Priority Overrides

Unreal Engine can implement attribute priority overrides to objects and lights within the scene. This allows artists to effectively establish a lighting rig on Level Sequence and then override those settings (both transitional and lighting attributes) on a per-shot basis.

Priority overrides simplify the management of customized placement of objects and lights, whether to address specific placement issues in the shot layout or to modify lighting on a case by case basis.

Light Rigs vs Spawnables

Fortnite implemented both light rigs and spawnable lighting solutions to light the sequences within the trailer. Both approaches proved performant, but creating a light rig provided a better sense of organization for light planning. Ultimately, the best approach would be to construct a dedicated light rig for overall sequence lighting, and then supplement the rig with spawnable lights when required.

Distance Field Ambient Occlusion

To reduce overhead for real-time playback, the Fortnite team pre-computed soft shadows for non-animated environment objects.

To aid in the computation of soft shadows, Unreal Engine can determine distance fields from mesh objects. A distance field describes the surface of a mesh object as a series of XYZ distances from an origin point, resulting in a point cloud to represent the object. The lower the resolution of the distance field, the lower the number of points recorded and the "softer" the definition of the object.

Distance fields lend themselves well to calculation of soft shadows. Producing soft shadows from hard surfaces is computationally intensive, but shadows generated for distance fields are naturally soft as they are cast on these softer versions of objects.

Ideally, a distance field is computed with a resolution high enough to represent the object but low enough to retain softness and also keep computing time to a minimum. In Unreal Engine, the resolution of a mesh's distance field is controlled by its volume texture resolution.

For the environment in the Fortnite trailer, Distance Field Ambient Occlusion was routed into various objects' material settings to provide a sense of shadowing by non-shadow casting and indirect lights. These soft shadows were pre-computed, which reduced overhead for real-time playback.

For more information on distance fields and their uses, see Mesh Distance Fields in the Unreal Engine documentation.

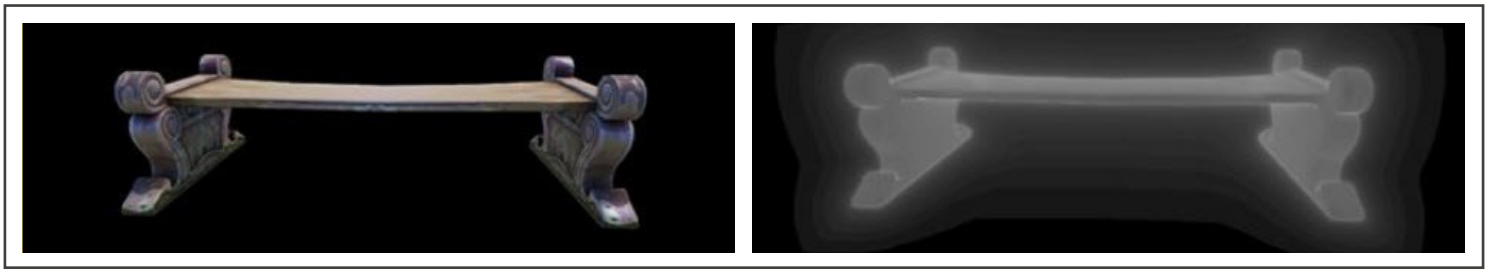


Figure 46: Original polygon mesh (left) and distance field representation (right)

Effects & Post Processing

Effects and Post Processing

Traditional animation pipelines rely on complicated render farms to calculate visual effects and post-processing enhancements as separate render passes that must be composited in a dedicated compositing program like After Effects or Nuke. While the use of these kinds of programs have their place, Unreal Engine eliminates a number of these steps by providing those capabilities within the engine.

For the trailer, the Fortnite team used Unreal Engine tools FFT Bloom to enhance glows and light effects and Cinematic Tonemapping for color correction.

The smoke-like storms and enemy deaths required special attention. In Fortnite, enemies always emerge from the storms themselves. A primary objective for the Fortnite trailer was to reinforce the connection between the enemy characters and the storms.

Figure 47: Monster emerging from the storm

To help achieve that integration, real-time volumetric rendering methods were used to keep all the FX live and adjustable in Unreal Engine, rather than import them from an offline package.

Enemy Deaths

The trailer required more than 25 unique husk (monster) death effects. Due to Entertainment Software Rating Board (ESRB) restrictions, the effects had to be non-violent. Epic decided each monster would quickly dissipate into a cloud-like material when it died, reducing perceived violence while helping to bridge the story's storm/monster connection.

While this type of effect can be externally generated and rendered through simulation software, such simulations would ordinarily be generated early in the pipeline and then would have to be reworked and re-exported every time a camera or animation was changed. Instead, Epic decided to use a fully real-time fluid simulation in post processing, again taking advantage of the non-linear workflow Unreal Engine is designed to support. Cameras and animation could be refined as much as the artistic process required without the need to regenerate a simulation from an external package.

Each death needed to be lit to match the current scene, and also needed to respond to scene forces. By simulating the effect in real time, the results were more changeable.

Figure 48: Monster death from hammer swung in a circular trajectory

The enemy monsters were voxelized using a simple trick of converting character skeletal meshes into "smoke

monsters" to seed the density of the volume textures. Mesh motion vectors were also captured during voxelization and added to the simulation during advection stages. Artists used velocity with a combination of curl noise and normals from the skeletal meshes.

To save on performance, the single-scattering method was used to light the volumes. With single scattering, just the light that hits the volume is reflected back. Multi-scattering, which calculates the light bouncing within the volume, was deemed too expensive to implement for a real-time solution. However, the effect can be approximated by blurring the results of single scattering.

By using a color instead of a scalar for the shadow density, the light hitting a volume could be made to change color with shadow depth, allowing a wider range of artistic effects. This "extinction color" gave more stylized color controls of the purple and pink smoke the monsters emitted when killed.

Each sim was controlled via the Sequencer. Sims were triggered one frame early to allow capture of motion vectors. Afterwards, the mesh contribution (density and velocity) could be keyframed. This allowed for very rapid iterations when compared with the wait times incurred by traditional offline rendered simulations.

Pressure iterations were run at half resolution as an optimization. Simulations were done using a GPU-accelerated Navier-Stokes-based fluid solver. The monster characters themselves were the emitters, contributing their densities and velocities to real-time simulations. These contributions could then be controlled by tracks in the Sequencer which made it easy for final tweaks to be made.

Blueprints were also created that could convert any sim into a Flipbook[4] from the specified camera. Flipbooks were used as a fallback plan for any shots that could not afford simulation— for example, if a large number of monsters were being killed in a single shot.

In key shots, invisible meshes were used as emission sources for art-directable motion and velocity. Volume textures were stored as pseudo-volumes and laid out as slices in a standard 2D texture.

The 3D simulations were rendered using a standard raymarch shader.

Storms

The shapes of the storms were defined via procedural distance field layers that were animated using 3D Flowmaps, a type of animated texture in Unreal Engine. The Flowmaps were hand-painted inside of virtual reality using custom Blueprint tools.

Painting the clouds' Flowmaps in VR was a more natural experience than using a mouse or stylus. Clouds were also animated to grow and spread by animating thresholds for the Distance Field layers.

All volumetric storm clouds were rendered using a raymarch shader similar to the one used for the enemy deaths, but with 3D Flowmaps instead of full fluid simulations. To track long, curly wisps of clouds without distortion, an iterative process was used to generate the Flowmaps.

Figure 49: Storm shape in Fortnite trailer

[4] In Unreal Engine, a Flipbook is a series of 2D images played back at a specified speed. Playing back a Flipbook is faster than playing a fluid simulation in real time.

Shapes were a mix of procedural layers and hand sculpting in real-time in a virtual reality environment. The velocities used for the Flowmaps were also a mix between curl noise and hand-sculpted velocities in VR. Procedural shapes were used more for shots showing lots of growth, as artists controlled the growth of the procedural shapes parametrically over the life of a shot.

Volumetric Fog

Some shots called for cloud effects to evoke the feel of the impending storm, but didn't require the custom shapes required by enemies and storms. For such shots, the Fortnite team made use of Unreal Engine's built-in Volumetric Fog.

Because Volumetric Fog uses inverse-squared distribution of voxels, detail farther away from the camera is lost. This makes Volumetric Fog more suitable for overall scene effects that are relatively soft, as opposed to specific shapes. A custom per-object raymarch shader was used to generate more custom effects and maintain detail in the distance.

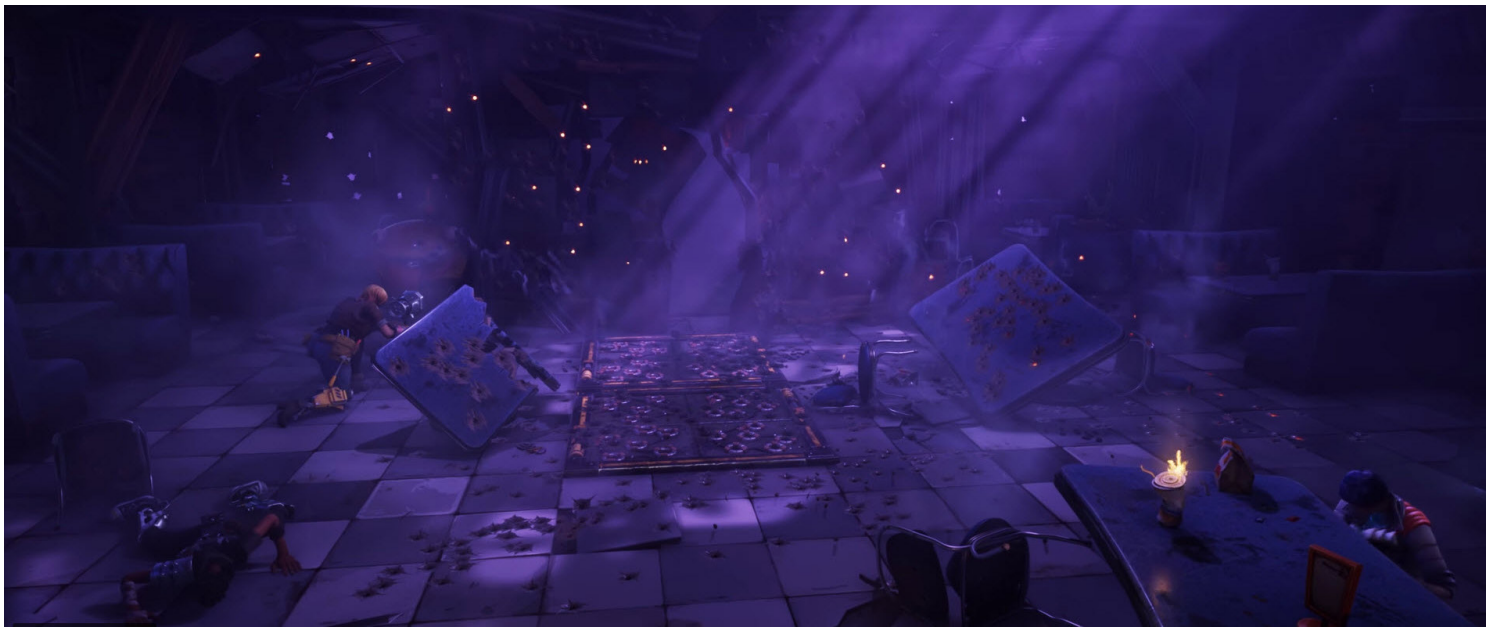
The fog in a level could receive shadows and was assigned custom materials to add detail. The team used the same pseudo-volume technique from the storms and sims to pass the 3D texture data. Volumetric fog effects were generated in real time during playback.

Unreal Engine allowed Epic to push the boundaries by showing film-quality simulations are possible in real time. Creating all these visual effects in real time allowed them to achieve a quality and integration that would not have been possible within a standard offline workflow.

Final Output

When all production was complete, the frames were exported from Unreal Engine as uncompressed 1920x1080 PNG files and converted to QuickTime (MOV) using Adobe Media Encoder and the PNG video codec. Adobe Premiere CC was used for editing with audio.

The final output was delivered as Quicktime (MOV) encoded with the PNG video codec.



Product Data & Information

Project Data & Information

Production Schedule

- Development start: August 2016
- Game asset enhancements start: Nov 2016
- Pre-production start: January 2017
- Production: Feb-Apr 2017
- Base sequence lighting start: January 2017
- Shot lighting start: March 2017
- Effects start: April 2017
- Final adjusted due date: May 2017

Team Composition

- Michael Clausen - Sr. Cinematic Designer and Co-Director
- Gavin Moran - Creative Cinematic Director and Co-Director
- Andrew Harris - Studio CG Supervisor
- Michael Balog - Director Animation Technology
- Ryan Brucks - Principal Technical Artist
- 3 Senior Cinematic Artists
- Approximately 7 to 10 Unreal Engine lighting, effects, and technical artists
- External Animation Team - Steamroller Studios, FL
- Editorial Team

Target Platform

PC Configuration:

- PC - i7 (7th Gen processor or higher)
- 64 GB RAM
- SSD Hard Drive
- 1080 GTX / 1080 Ti / P6000

About this Document

Authors

Brian J. Pohl
Tim Brakensiek
Simone Lombardo

Contributors

Michael Clausen
Gavin Moran
Michael Balog
Brian Brecht
Andrew Harris
Ryan Brucks
Sebastien Miglio

Editor

Michele Bousquet