# ICI Laflaque

## Broadcast quality within a tight timeline

# Contents

# ICI Laflaque

## Broadcast quality within a tight timeline

The production team for *ICI Laflaque*, an animated weekly political satire show, has worked with a real-time pipeline since its inception in 2004. This workflow was developed out of necessity. Each 30-minute episode includes brand new scripts based on breaking news that need to be produced within a 7-day turnaround time, from voiceover to animation to final output.

From the start, real-time rendering (or near-real-time rendering) was needed to meet this timeline week after week. For 14 years, Vox Populi Productions produced ICI Laflaque on this aggressive schedule by using near-real-time rendering straight out of Autodesk MotionBuilder as the final output.

After recognizing that Unreal Engine could give them a great boost in visual quality, Vox Populi decided to switch to Unreal Engine for the 2018-2019 season. This document is about the changes Vox Populi made to the show's pipeline to use Unreal Engine for superior real-time output while keeping intact the parts of their workflow that were already optimized.

In addition to much higher visual quality for the show, the production team also discovered other ways Unreal Engine could speed up their work and leave more time for creativity.

# The Show

## The Show

*ICI Laflaque* is an award-winning French-language political satire show. The show, which airs on the ICI Télé Radio-Canada network, features a large cast of 3D animated characters, both fictional and reality-based.

The star of the show is Gérard D Laflaque, a fictional news anchor.

Each episode follows two storylines. The first is Laflaque's current affairs TV show, also named "ICI Laflaque", where he comments on the news of the day and interviews famous political figures, and the second weaves in various fictional stories that revolve around public personalities. Many of the segments refer to real-life news that happened just a week before.

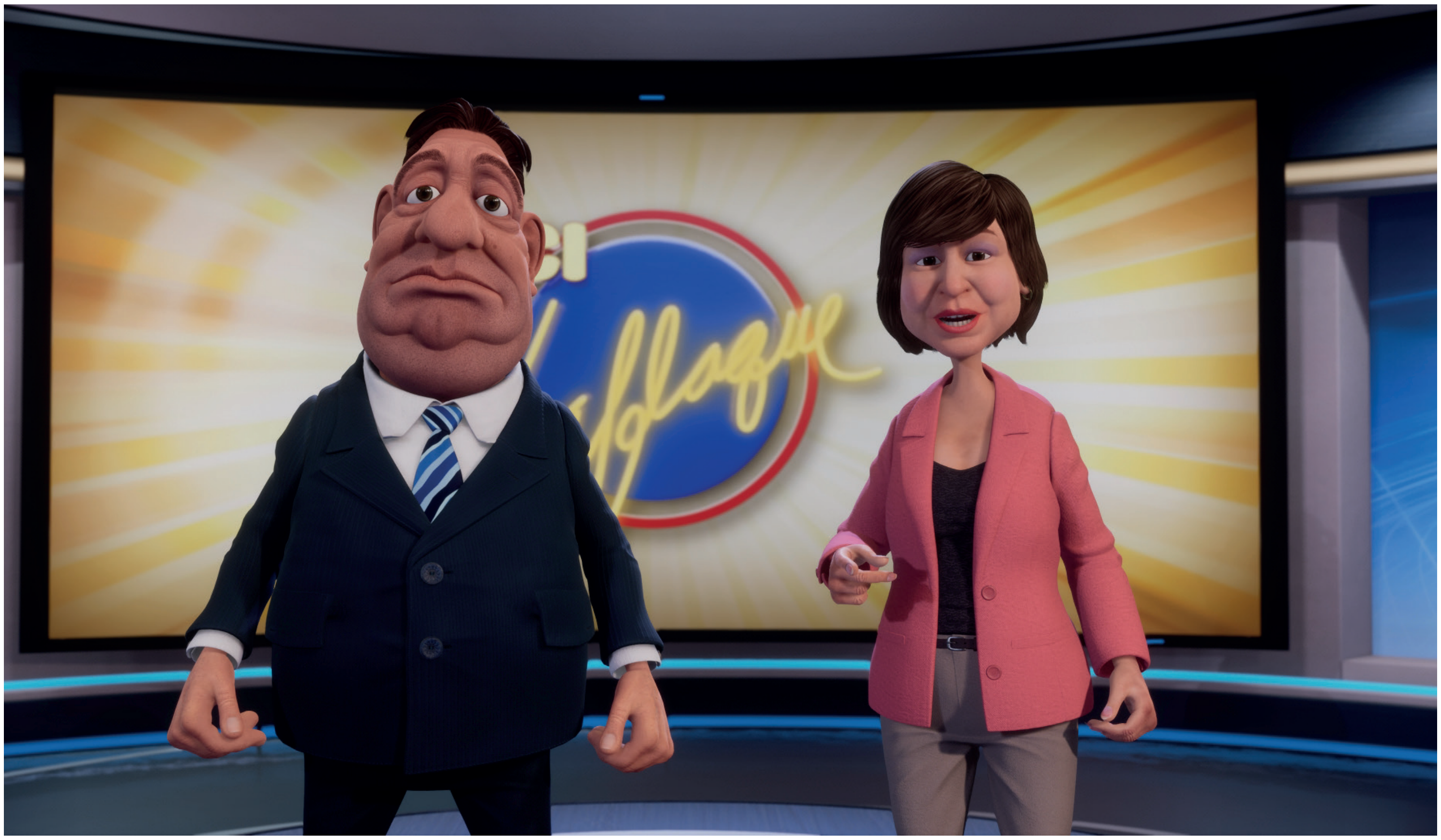


Figure 1: Gérard D Laflaque on the set of his TV show with Radio-Canada news anchor Céline Galipeau

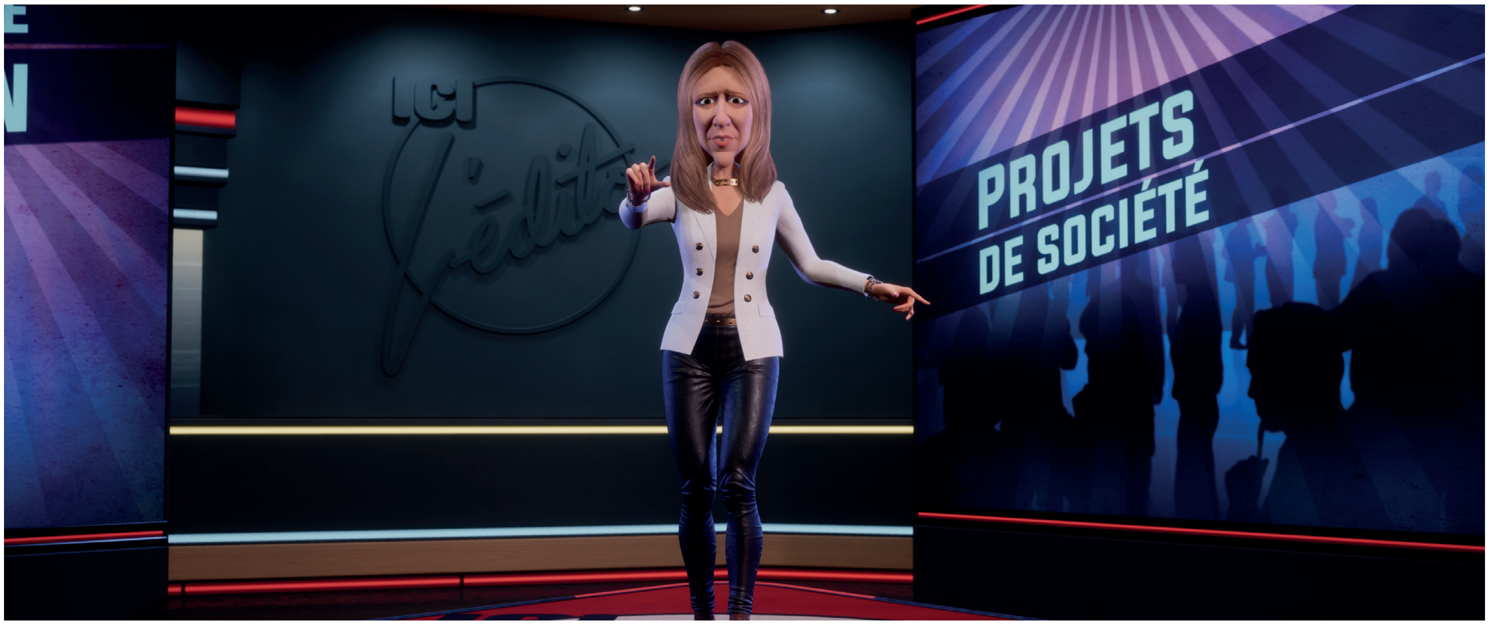


Figure 2: Céline Dion makes an editorial speech

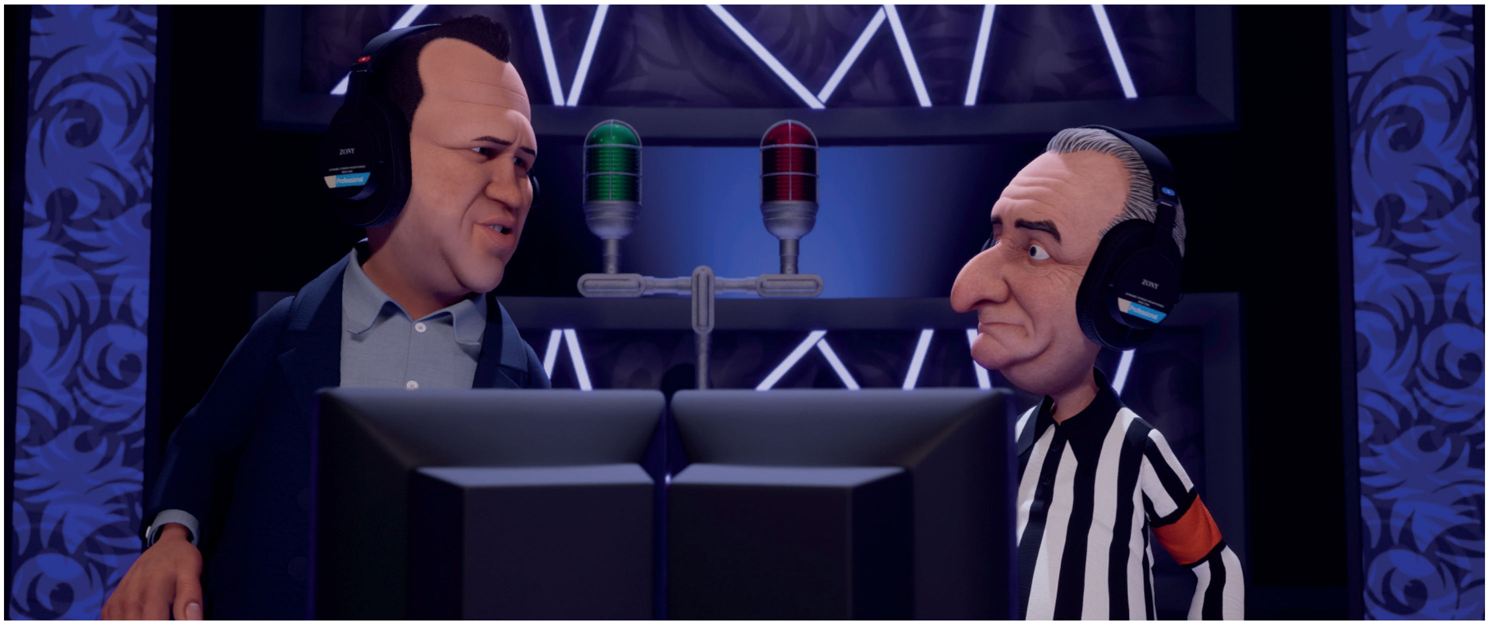


Figure 3: Former Montreal Canadiens player Dave Morissette and referee Ron Fournier discuss hockey

Throughout his tenure as a political TV show host, Gérard D Laflaque has interviewed a variety of guests including Donald Trump, Barack Obama, Hillary Clinton, Kim Jong-Un, President of France Emmanuel Macron, and a number of Quebecois and Canadian politicians.

The show's scripts are irreverent and witty, and reflect a Quebecois point of view. When Laflaque asks USA President George W. Bush to explain the USA's electoral college voting system, Bush replies that it's simple: the candidate with the most money wins. During an interview with then-Canadian Prime Minister Stephen Harper about his stance on certain issues, Laflaque examines the PM's body with an X-Ray machine and proceeds to show that Harper has no heart.

Other recurring sketches include the happenings at a local liquor store named FLAQ, and parodies of Quebecois TV game shows.

## History

Gérard D Laflaque made his first public appearance in 1982 as a latex marionette. Back then, French-language public television station Radio-Québec aired hilarious 75-second vignettes of Laflaque's commentary in the aptly named *La minute et quart à Gérard D Laflaque (The minute and a quarter of Gérard D Laflaque)*. Laflaque's name is inspired by Gérard D. Lévesque, a prominent Quebec politician, but with a twist—"la flaque" is French for "the puddle".

The show's concept, and Gérard D Laflaque himself, were conceived by award-winning Quebecois political cartoonist Serge Chapleau. After creating puppets for television, and he turned his craft to creating the first marionettes for these segments, including the original Gérard D Laflaque puppet.

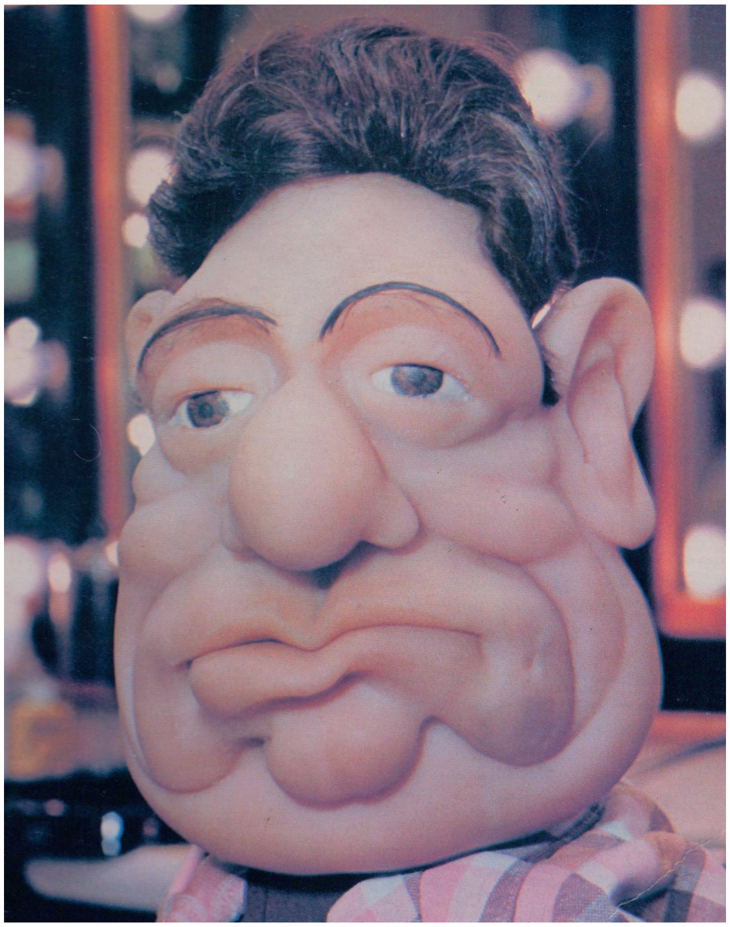


Figure 4: Gérard D Laflaque as a latex puppet

In time, the world of Laflaque got its own 30-minute show on Radio-Canada with the name *Et Dieu Créa... Laflaque (And God created... Laflaque)*, which was eventually renamed *ICI Laflaque (Here is Laflaque)*. All the character design for the show is based on Chapleau's drawing style. Chapleau is still active as a content producer for the show—he continues to supervise model sheets for new characters, and also approves show scripts.

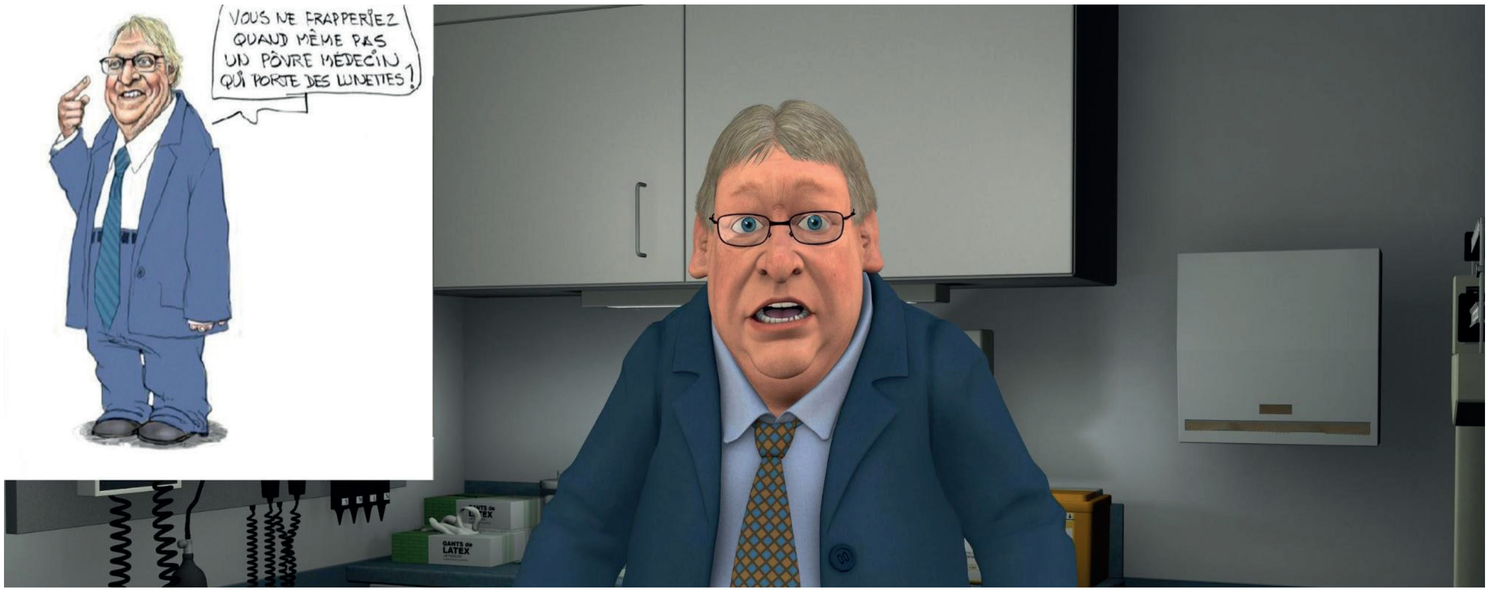

VOUS NE FRAPPERIEZ QUAND MÊME PAS UN PÔVRE MÉDECIN QUI PORTE DES LUNETTES ?!
GANTS de LATEX

Figure 5: Serge Chapleau caricature of Canadian politician Gaétan Barrette

When the Laflaque show started production in 2004, Vox Populi developed a pipeline using motion capture and MotionBuilder to animate and render the show, and OpenGL for real-time rendering. The team used real-time previews during production, and final output from MotionBuilder was rendered in near real time.

The pipeline worked sufficiently well for producing a new 30-minute episode every 7 days. But after 14 years, Vox Populi decided to upgrade the show's look by switching to Unreal Engine.

## Production Schedule

In 2018, the Laflaque show is entering its 15th season. The production team has always followed an aggressive schedule for the 30-minute show. During the week, writers are still producing scripts about news that happened a few days or even hours before, and finished productions of these scripts are included in the footage delivered to Radio-Canada a few days later.

The annual production schedule follows this pattern, regardless of the pipeline used:

- May-August: No episodes are produced. The team writes and produces short sketches on "timeless topics", ones that traditionally repeat in Quebec and around the world: global warming, family values, unemployment, and so on. The production team also works on new sets and characters that might be needed, for example a newly elected head of a country or a Canadian politician rising to prominence.

- September-April: Production of weekly episodes comprised of short sketches. Roughly half the sketches included in the weekly show are "timeless topics" previously produced, while the other half, based on recent news, are newly produced that week. Each season is 20-30 episodes (the number varies from year to year).

For the 2018-2019 season that started in September 2018, the production team aimed to create an Unreal Engine project containing 26 episodes, 100 characters, 50+ backgrounds, and more than 1000 props and specials[1]. Each episode would be 30 minutes long, and produced on a 7-day schedule.

On Monday through Thursday of each production week, the team meets to discuss the new scripts and whether they can be produced by Friday. Action that would require a new, complex 3D model might be rewritten to exclude such a model. If a new minor character is introduced, the team discusses whether existing models can be quickly repurposed or whether the character could be represented as a cardboard cutout.

As part of this challenge, the Vox Populi team sometimes has to deal with "script emergencies" during the week that leave them with only three days to produce a finished sketch. Because of the show's short turnaround time, having a pipeline that's responsive to these challenges is very important.

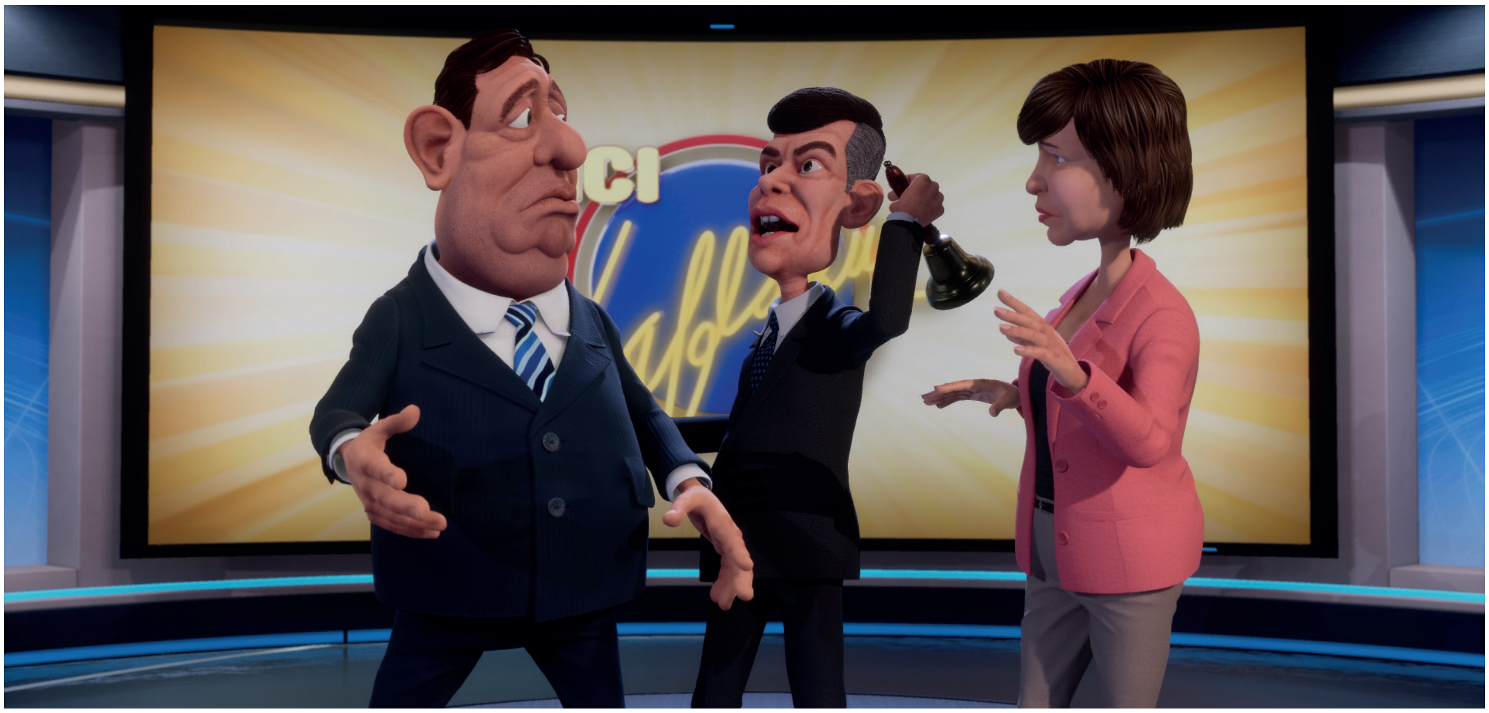


Figure 6: Political commentator Mario Dumont rings the bell in protest of cannabis legalization

[1] Vox Populi uses the term "specials" to refer to clothing, accessories, and props selected for a specific character in a specific sketch. For a recurring character, the team has many previously-modeled items to choose from: outfits, hats, eyeglasses, etc. as well as props like pens, cell phones, clipboards, and briefcases. Most of these items have their own rigs. Choosing the specials for the character means fully outfitting him/her for the sketch, including all accessories and hand props.

## Switch to Unreal Engine

## Switch to Unreal Engine

Vox Populi had been keeping an eye on Unreal Engine for a few years as a possibility for a real-time rendering solution, and was impressed by The Mill's Blackbird project and ILM's use of Unreal Engine for final renders on Rogue One.

In 2018, Vox Populi partnered with Epic Games to explore an Unreal Engine workflow that would replace their existing real-time solution and give a boost to the show's visual quality.

Vox Populi envisaged a switch to an Unreal Engine pipeline in two phases:

- Phase 1: Motion-capture and keyframe data (or just animation data) sent to MotionBuilder, rendering in Unreal Engine
- Phase 2: Live feedback of motion capture data in Unreal Engine via the MotionBuilder LiveLink plugin

This document relates to the Phase 1 change. Phase 2 will be performed at a future date.

For Phase 1, the team wanted to keep the show's overall look the same, but with an upgrade in visual quality—the show would look better to viewers while still being recognizable as the same show. With Unreal Engine, the team would be able to use PBR (physically-based rendering) textures, which are optimized to work with real-time lighting. Such textures generally result in richer colors and a cleaner, smoother look for renderings.

In addition, Vox Populi needed to minimize changes to their existing workflow to avoid disrupting the entire show with R&D. The goal was to develop a basic (and solid) UE4 pipeline to replace their existing real-time rendering process.

To ensure the new pipeline was robust, Vox Populi enabled a side-by-side production with old and new workflows.

Comparisons of final output appear in the Rendering section of this document.

### Original Pipeline

In the original pipeline, Vox Populi already had several steps worked out for maximum efficiency including modeling, motion capture, and animation.

Throughout the show's history, the team has modeled characters and sets in ZBrush and 3ds Max, set up basic rigging in 3ds Max, used a PhaseSpace system for motion capture, and finished up rigging and animation in MotionBuilder. There was no need to change these parts of the pipeline to use Unreal Engine for real-time rendering. And since the team was already very familiar with these workflows, the switch to Unreal Engine wouldn't impact the time needed to perform these tasks.

The team had been using MotionBuilder OpenGL output for previews and also final pixels. While previews could play back in real time at 30 fps, final output could be generated only at a less-than-real-time speed of 5-10 fps. By using Unreal Engine, Vox Populi hoped to improve the quality of previews and also the speed of final output.

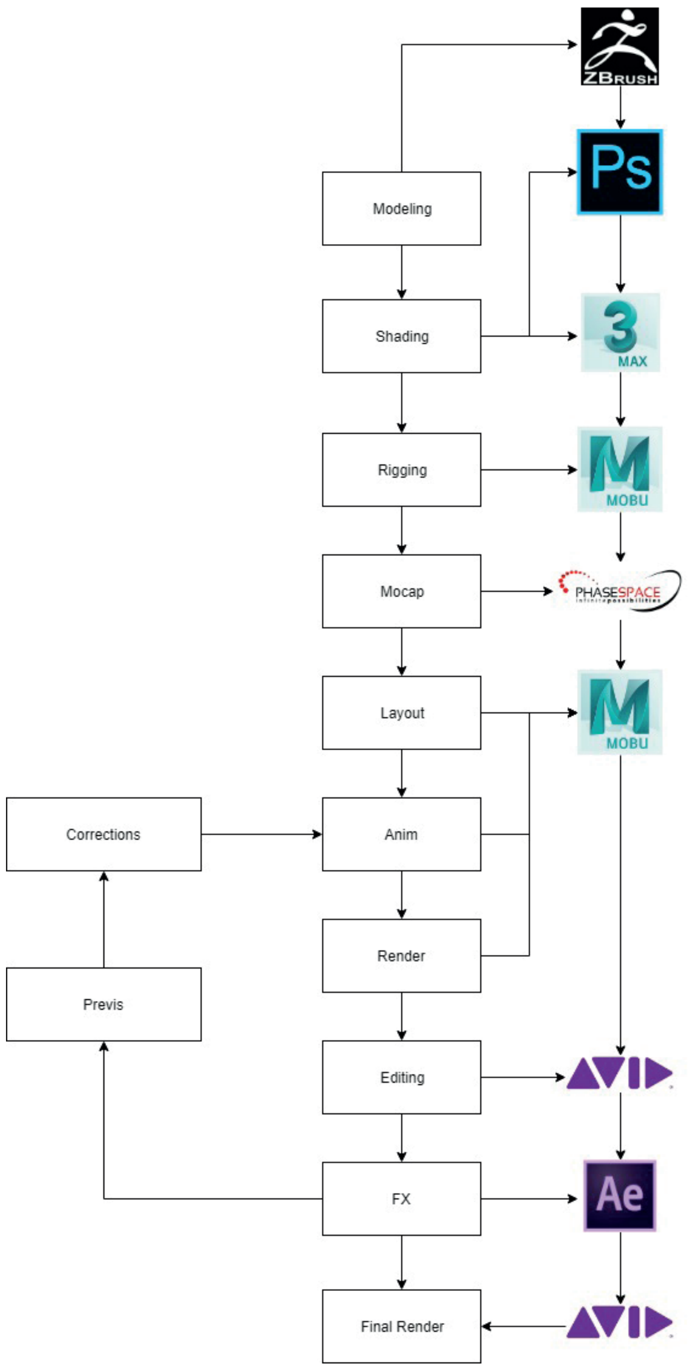


Figure 7: Linear MotionBuilder pipeline

## Unreal Engine Pipeline

IIn the first phase, the team aimed to make the following changes to the pipeline:

**Shading** - Shading would be done in UE4. In the original pipeline, all shading was accomplished with textures painted or edited in Photoshop, and materials were assigned in 3ds Max and MotionBuilder. The production team wished to streamline this process by using Unreal Engine materials as much as possible.

**Rigging** - The rigging process in MotionBuilder would remain largely the same, with some tweaks to work efficiently when imported into Unreal Engine.

**Lighting** - Since rendering would be performed in UE4, it made sense to do all lighting in UE4.

**VFX** - In the show's existing pipeline, VFX were created separately in After Effects. With Unreal Engine, VFX could be done concurrently with animation in UE4.

**Compositing** - By using the tools in Unreal Engine's Sequencer, compositing steps in the original pipeline could be done directly in Unreal Engine.

**Rendering** - Animation previews are still done in MotionBuilder, but the final rendering would be performed with Unreal Engine instead.

The team also wished to easily swap out specials and characters in UE4, and have access to the Camera Switcher tool from MotionBuilder when importing to the Sequencer camera options.

The Unreal Engine pipeline looks quite different from the original pipeline, with a more parallel workflow between all the production steps.

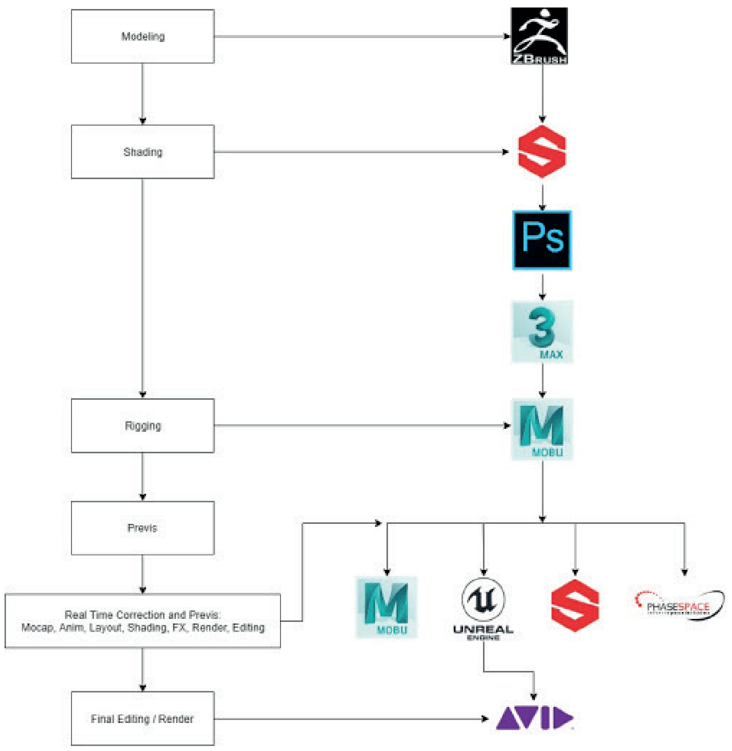


Figure 8: Real-time UE4 pipeline

Other benefits the Laflaque team realized from switching to an Unreal Engine pipeline:

- Vox Populi was able to use the existing MotionBuilder assets without much correction.
- In the original pipeline, the team used MotionBuilder's real-time preview capabilities to review animation in its initial stages. Unreal Engine provides a real-time GPU system for viewing, offering a much higher-quality preview.
- Perforce could be used as an integrated tool to manage the assets and split the work within the team.

## Tools Used

### Original Pipeline

Autodesk® 3ds Max® - Modeling, UVs, morph targets for clothing and facial animation, rigging, skinning

Pixologic® ZBrush® - Character sculpting

Adobe® Photoshop® - Texture painting

Autodesk MotionBuilder® - Motion capture data processing and classic animation, real-time previews and output

PhaseSpace® Motion Capture - Motion capture for body motion

Adobe After Effects® - VFX, post processing, infographics

Avid® Media Composer® - Sound, final edits, credits, deliverables

### Unreal Engine Pipeline

Most of the tools of the original pipeline are still used in the real-time pipeline, but a few are added or switched out to maximize efficiency with Unreal Engine. Tools that are new to the Unreal Engine pipeline, or those that have different functions in the new pipeline, are listed below.

- Adobe Photoshop and Allegorithmic Substance Painter and Designer - Material painting and authoring
- Autodesk MotionBuilder - Motion capture data processing and classic animation
- Adobe After Effects - VFX, infographics
- Unreal Engine - Previews, post processing, final output

## Production Steps

## Production Steps

While some steps stayed the same with the new pipeline, some steps changed substantially. The following table summarizes the changes needed to optimize the original pipeline for the new Unreal Engine pipeline.

| Step | Original Pipeline | Unreal Engine Pipeline |
|---|---|---|
| Motion Capture | PhaseSpace into MotionBuilder | <no change> |
| Layout | MotionBuilder | MotionBuilder & Unreal Engine |
| Modeling | ZBrush & 3ds Max | <no change> |
| Shading | 3ds Max & MotionBuilder | Substance Painter in UE4 |
| Rigging | 3ds Max & MotionBuilder | <no change> |
| Animation | MotionBuilder | No change to body/facial animation process. Texture switching now done in UE4. |
| Transfer to Unreal Engine | <N/A> | FBX |
| Lighting | MotionBuilder | Unreal Engine |
| Rendering | MotionBuilder OpenGL | Unreal Engine |
| Compositing and VFX | After Effects | After Effects & Unreal Engine |
| Editing and Delivery | Avid Media Composer | <no change> |

Table 1: Summary of pipeline changes

## Motion Capture

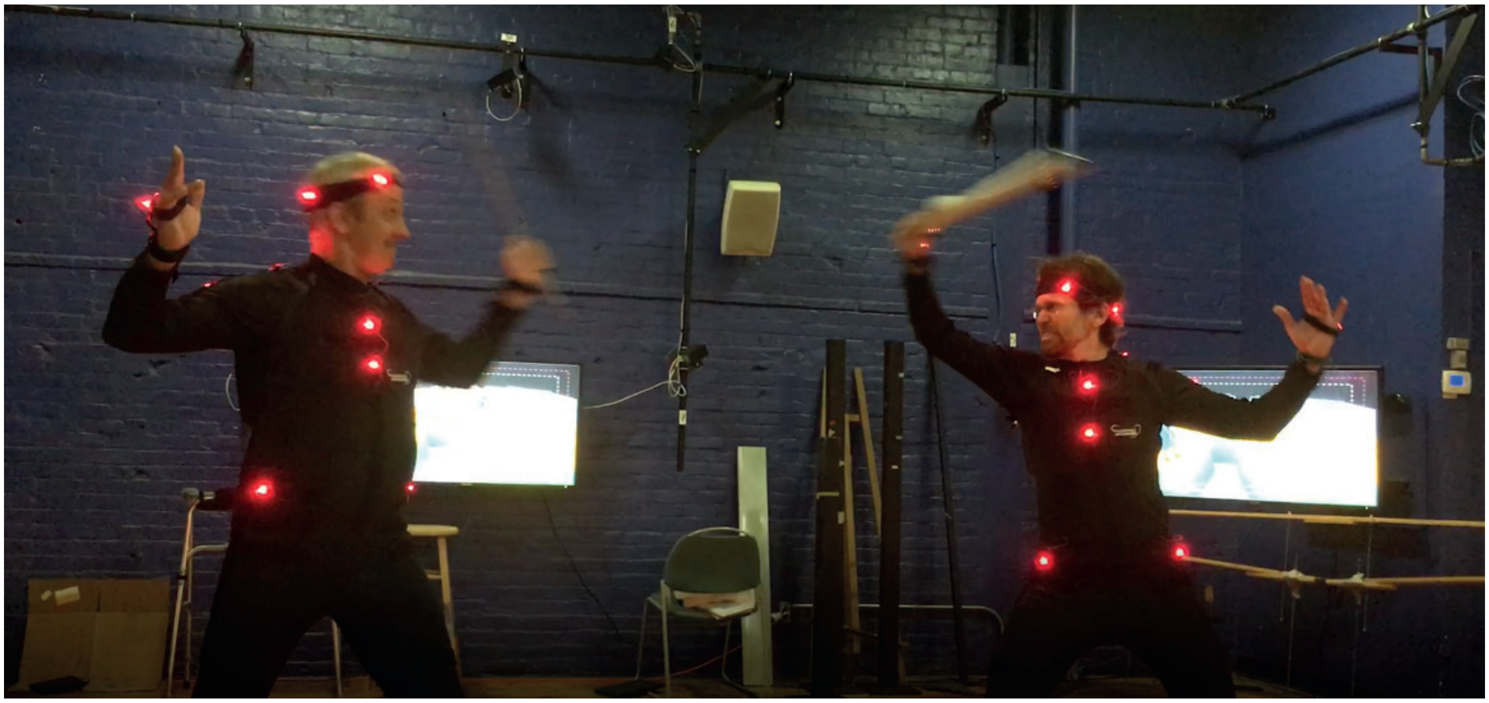


Figure 9: Laflaque Mocap Scene

Vox Populi uses PhaseSpace as its motion capture system for the Laflaque show. The system uses 34 Fujifilm X-E2 motion capture cameras with active markers.

The system was chosen for its balance of low cost and accuracy—the Laflaque show features a lot of broad motion as opposed to subtle movement. While the capture data contains a small amount of noise that needs to be corrected in MotionBuilder, the system amply serves the production's needs.

## Layout

In the original pipeline, during the motion capture session, the mocap data was streamed into MotionBuilder where it was applied to the characters in a mocap layout scene that included most of the actual final environment. This gave the director the means to review everything in context.

The director then reviewed the action in this MotionBuilder scene, setting up cameras and using the camera switcher to delineate the shots. After the cameras were set up, remaining assets (accessories) were added to the scene.

For the production, it was essential to be able to use the MotionBuilder cameras, since all animation is done in MotionBuilder. For this reason, the team needed a very accurate conversion of MotionBuilder's cameras to UE4's cameras.

The team used the camera import tool and some custom adjustments to keep a similarity with the MotionBuilder cameras.

## Importing Camera Data

In MotionBuilder, when creating a camera, a point of interest (look-at point) is generated automatically. This point, a null attached to the camera, manages the camera's rotation and field of view (FOV).

When exporting/importing FBX to UE4, the point of interest isn't converted, meaning the camera's rotation is lost. The solution was to plot each camera's rotation using a custom Python script in MotionBuilder before export, and to set a custom FBX property in UE4 to hold the focus distance value:

- In Editor Level Sequences / Track Settings / FBX Setting:
    - Add --- FBX Property: FocusDistance
    - Add --- Component Name: CameraComponent
    - Add --- Property Name: FocusSettings. ManualFocusDistance

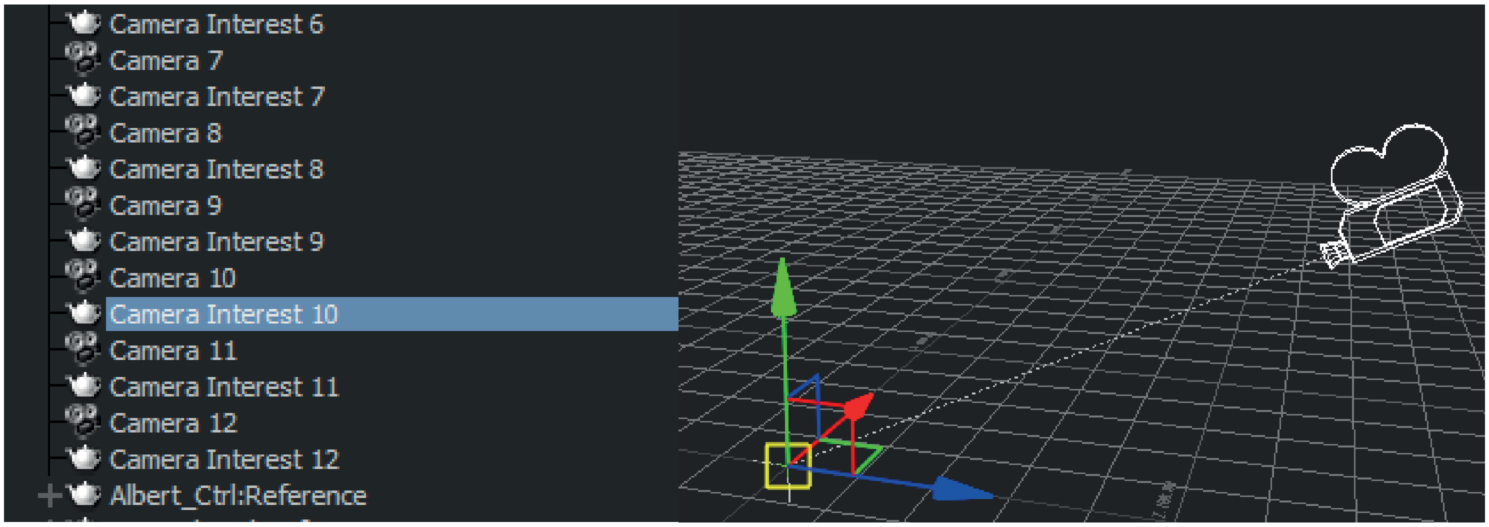
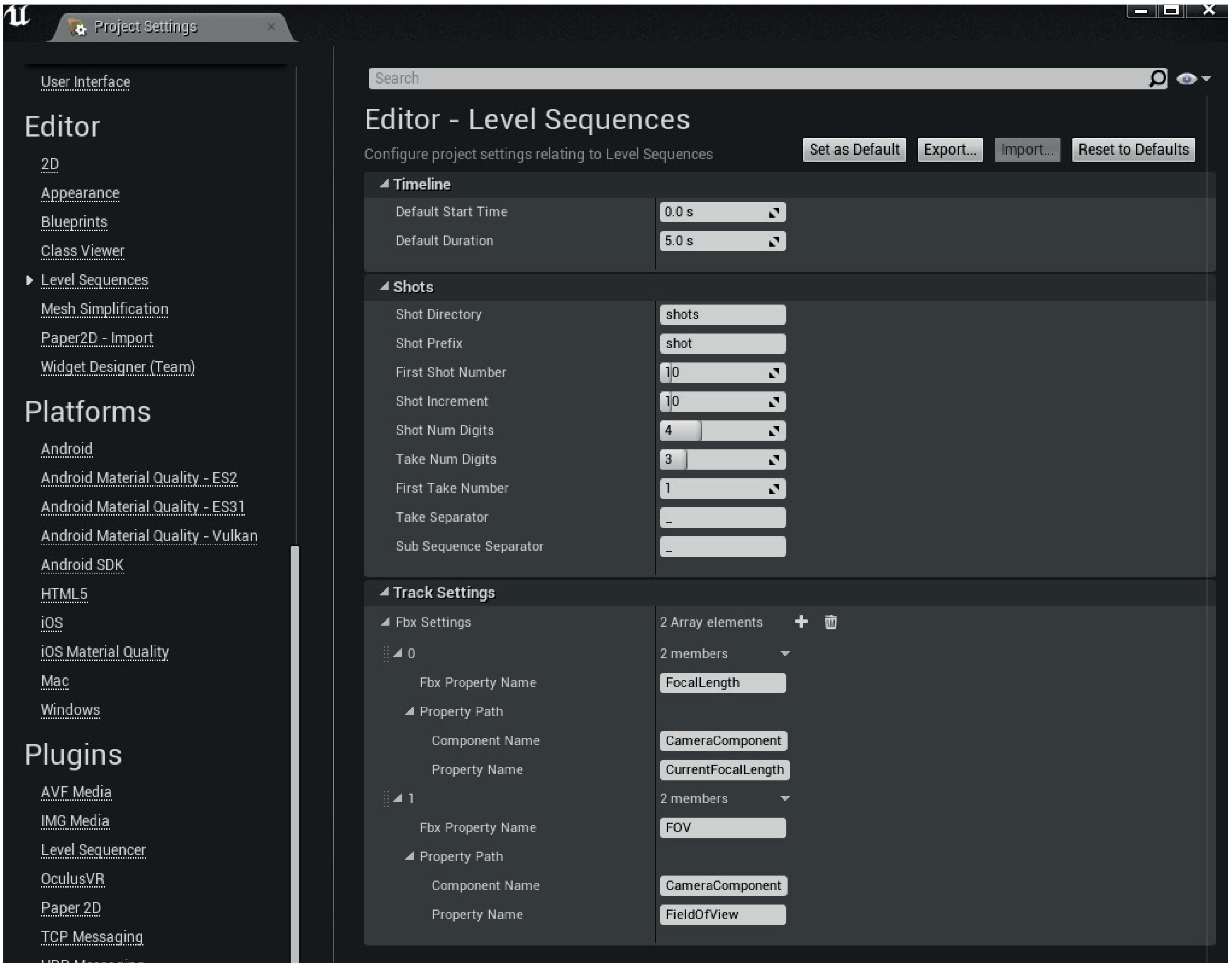


Figure 10: Camera Interest in MotionBuilder



Editor - Level Sequences

Configure project settings relating to Level Sequences

**Timeline**

| | |
|---|---|
| Default Start Time | 0.0 s |
| Default Duration | 5.0 s |

**Shots**

| | |
|---|---|
| Shot Directory | shots |
| Shot Prefix | shot |
| First Shot Number | 10 |
| Shot Increment | 10 |
| Shot Num Digits | 4 |
| Take Num Digits | 3 |
| First Take Number | 1 |
| Take Separator | _ |
| Sub Sequence Separator | _ |

**Track Settings**

| | |
|---|---|
| Fbx Settings | 2 Array elements |
| 0 | 2 members |
| Fbx Property Name | FocalLength |
| Property Path | |
| Component Name | CameraComponent |
| Property Name | CurrentFocalLength |
| 1 | 2 members |
| Fbx Property Name | FOV |
| Property Path | |
| Component Name | CameraComponent |
| Property Name | FieldOfView |

Figure 11: FBX Export Setting

This setup takes the Focus Distance from the camera parameter in MotionBuilder, which then gets imported into the UE4 Sequencer.

## Camera Switcher

MotionBuilder's camera switcher and story tools were an important part of the original pipeline. In switching to the new pipeline, one of the biggest needs was to convert the Camera Switcher data in MotionBuilder to work with the Camera Cuts feature in UE4.

The Unreal Engine developer team integrated the option to use MotionBuilder Camera Switcher data and update the data in the Camera Cut Track in Sequencer. By allowing Unreal Engine to understand camera IDs from MotionBuilder, the team could have an alignment of camera setups between packages where Sequencer will reflect MotionBuilder's Camera Switcher.

Maintaining the sequence cut from MotionBuilder's Camera Switcher simplifies the reconstruction of all the shots in UE's Sequencer tool.

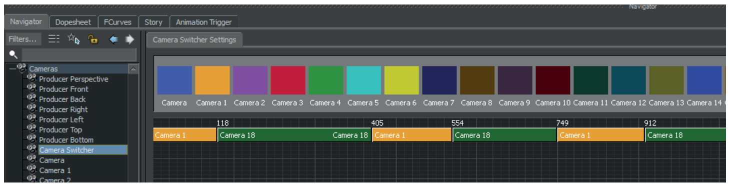


Figure 12: Motion Builder's Camera Switcher

## Modeling

The team models characters with ZBrush and 3ds Max, then uses 3ds Max to separate body parts such as teeth and clothing. Each character consists of around 160,000 triangles.

As for the sets, the main set is the ICI Laflaque studio where Gérard performs his interviews. Other sketches use interiors like a liquor store, a grocery store, an office, and the Parliament building in Quebec City, along with exteriors such as parks, parking lots, a downtown street, and a back alley. Many of theses sets were already built over the show's lifetime.

In the new pipeline, the models are created in the same way as in the original pipeline. The caricature drawing (model sheet) is used to make a ZBrush version and the model is then finalized in 3ds Max.

New characters are produced year-round, and the team tries as much as possible to repurpose existing elements, particularly clothing and hands. Existing characters are updated by adding new hairstyles, textures, props, etc.

## Shading

In the original pipeline, the team assigned textures created or edited in Photoshop to models in 3ds Max and MotionBuilder. As part of the upgrade to an Unreal Engine pipeline, the team switched to using the Allegorithmic Substance Tools.

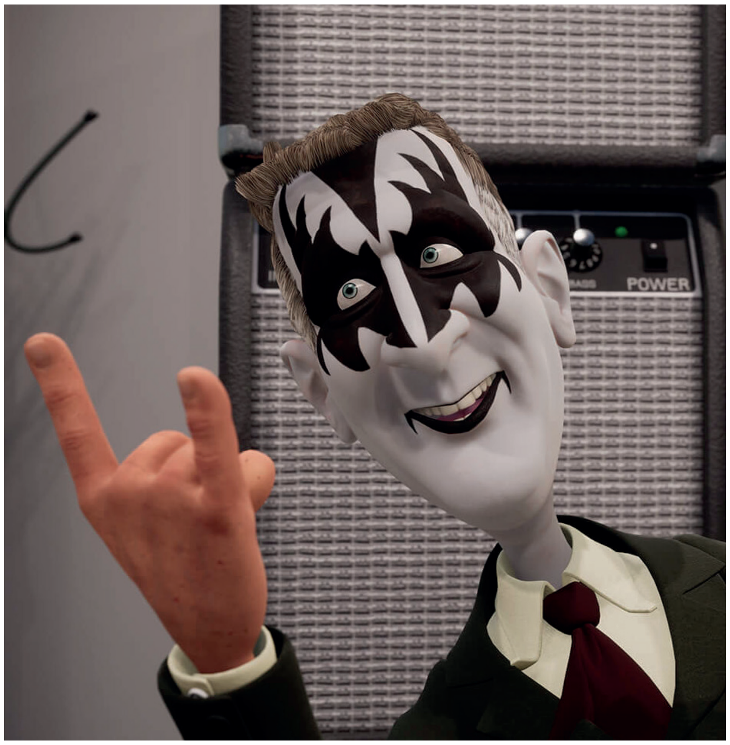


Figure 13: Textures created with Allegorithmic Substance Tools

Substance allows the team work on PBR materials as a whole rather than separate textures, most of which didn't exist in the previous pipeline. The Substance materials are then exported to individual textures for Unreal Engine to use.

During the initial tests, the team realized it would need to reprocess certain textures to conform to Unreal Engine's formats. File types, resolutions, and new shading possibilities needed to be taken into account.

Transparent objects like windows and eyeglass lenses were initially created using a translucent shader with refraction, but using refraction caused some rendering issues. The team instead used shading tricks to emulate a refraction effect.

The team also made use of texture switching in Unreal Engine for some of the animation, which is described in the Animation section of this document.

## Rigging

Characters are rigged and skinned in 3ds Max, then exported to MotionBuilder.

Character bodies are rigged with a combination of bones and Dummy objects (nulls). Character heads are given around 50 morph target shapes. Some of the accessories also have morph target shapes to facilitate animation.

The FBX stack for a character mesh exported from 3ds Max to MotionBuilder contains the geometry and the MeshSmooth, Morpher, and Skin modifiers. In MotionBuilder, the skeleton rig is a standard Control rig.

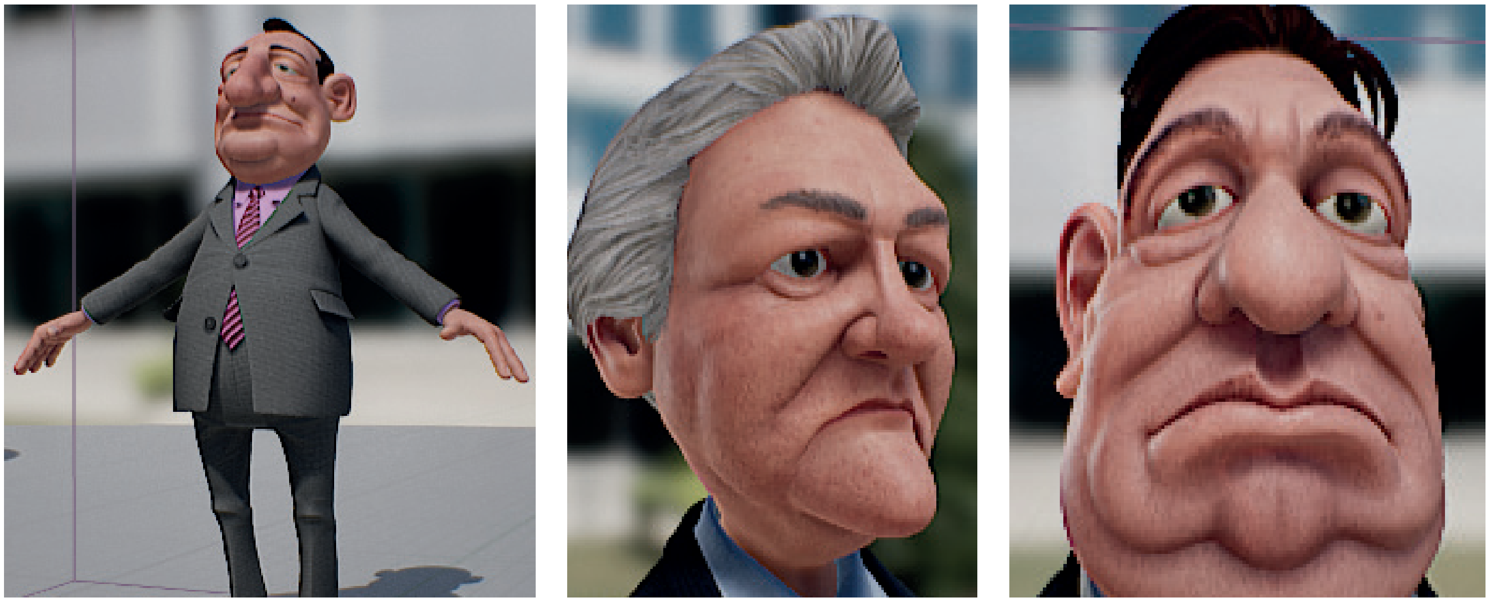


Figure 14: Laflaque character shading in UE4

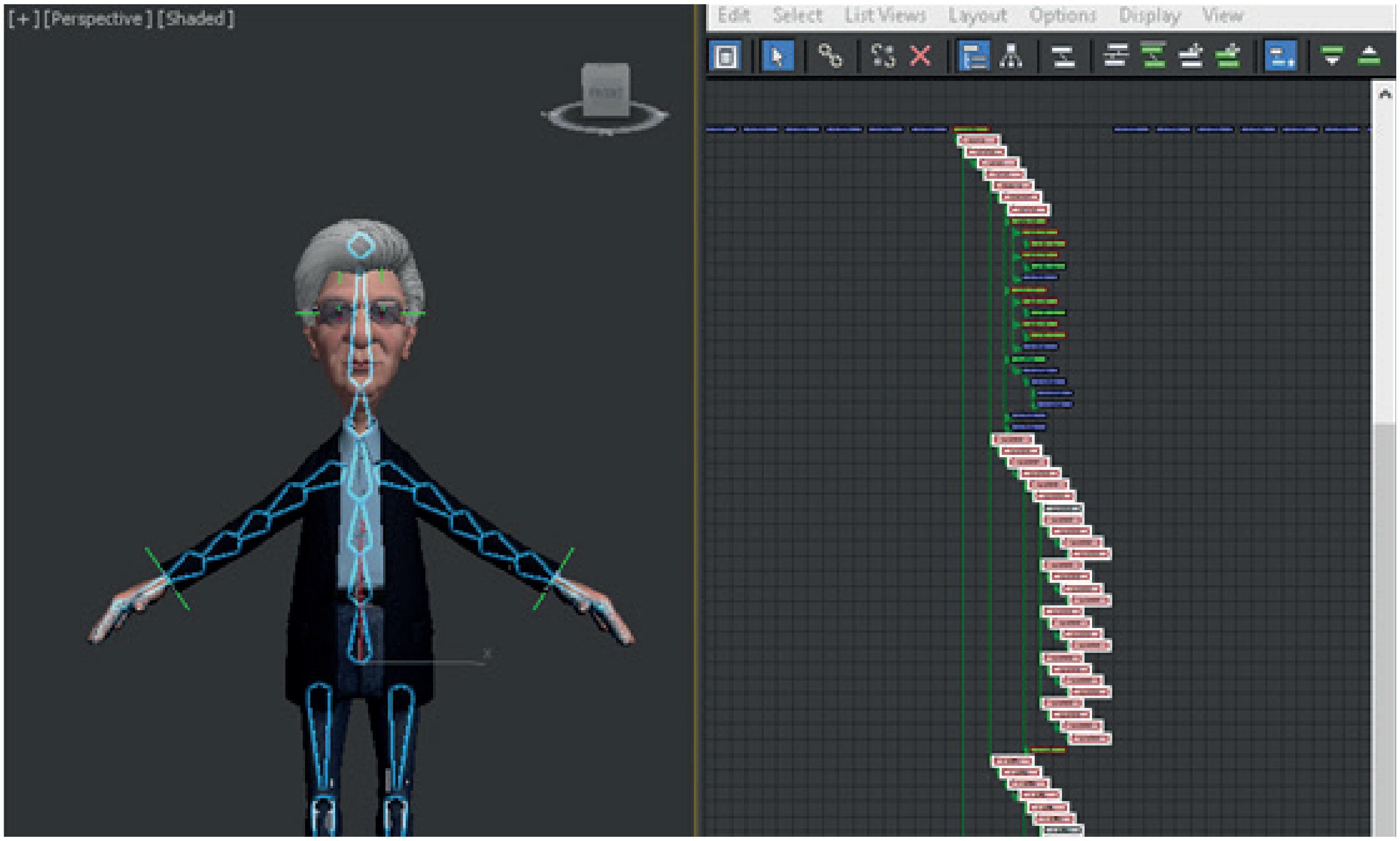


Figure 15: Laflaque character hierarchy in 3ds Max

The character facial rig includes shapes and clusters. Markers are added in MotionBuilder as needed to control secondary animation (effectors for IK channels, etc.). Several types of constraints for the animation of the eyes, clothes, and other secondary elements are also added.

Each of the characters use the same hierarchy and nomenclature, but each has its own unique MotionBuilder Namespace.

## Extra Bones

One of the defining factors of the Laflaque characters is the thousands of specials (props and accessories) that can be added to the basic character templates. The secondary motion from these specials is part of the personality and humor of the Laflaque show.

One example is the turban and beard added to the Pierre character. The bones added to these specials are parented to the character, giving the opportunity for secondary motion.

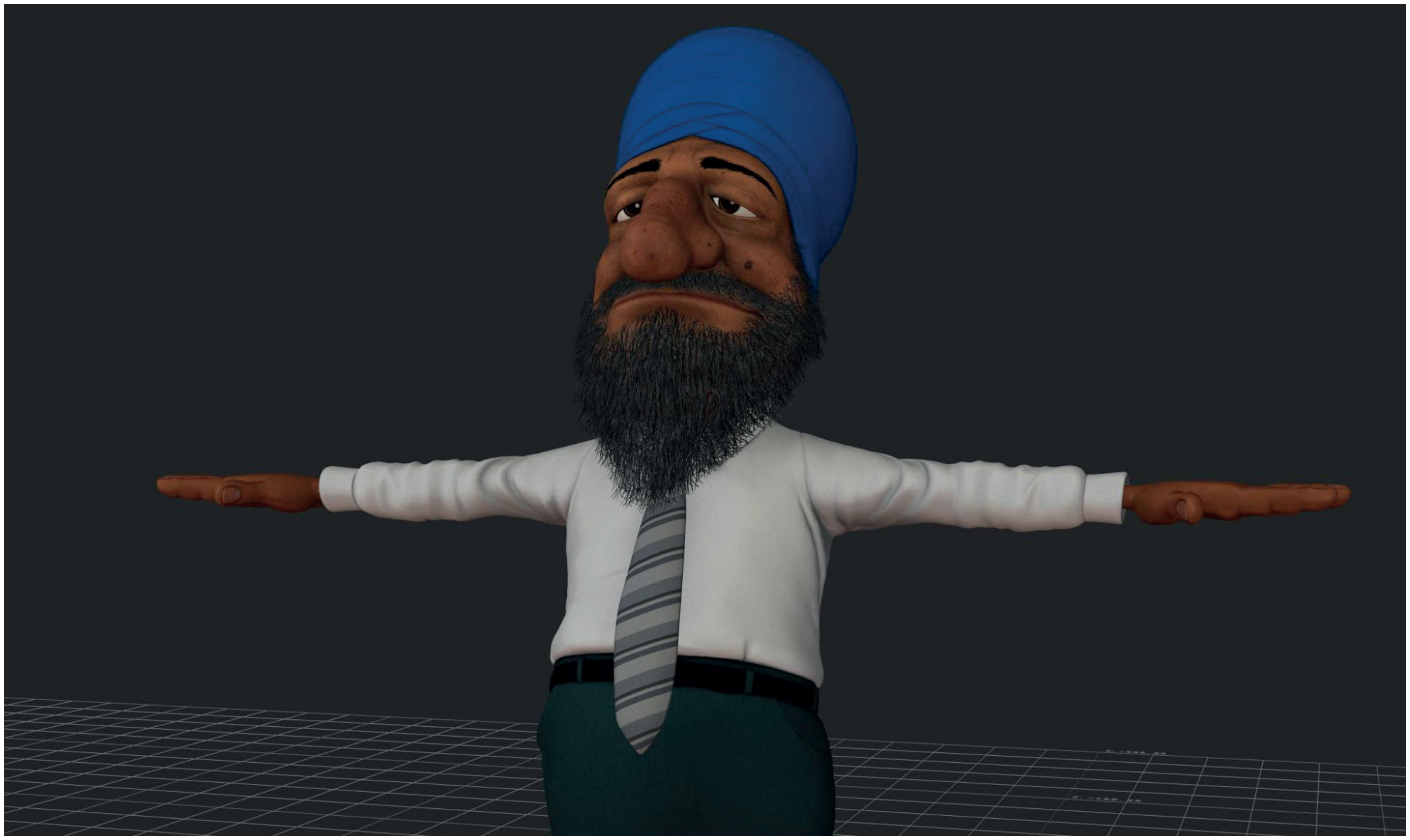


Figure 16: Pierre character base plus beard and hat specials with extra bones for animation

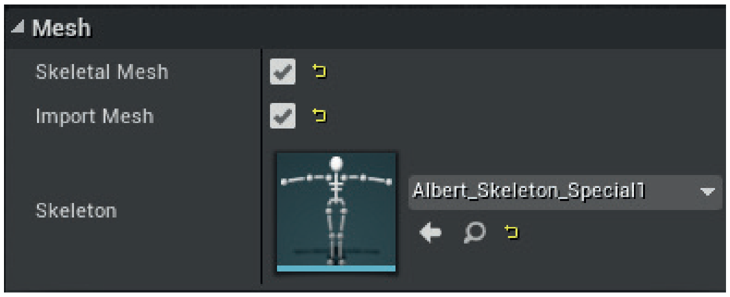
Over the 15-year CGI development of the show, hundreds of versions of characters have been created and thousands of specials have been added to them. In addition, several versions of any one accessory or prop might exist.

A high-poly version of a prop would be developed for close-ups and a low-poly version for distance shots or quick action, but the rigs of each version might not match. When MotionBuilder was the only application used for animation and rendering, these variations caused no particular difficulties. However, these variations brought unique complications to the new UE pipeline and to the management of different skeletal rigs. If the character and special each have their own rig, both these rigs had to be linked to the character rig before export to UE.

Vox Populi's original approach was to import the character and prop hierarchies separately, then use the Set Master Pose Component in Unreal Engine to set up a parent/child relationship between the character's and prop's hierarchies using Blueprints.

However, the connection in a Blueprint with Set Master Pose didn't work for this purpose as the extra bones became invalid. To use the Set Master Pose Component, the same hierarchy of bones was needed across all skeletal meshes.

The team needed to re-export the skeletons with the added bones from MotionBuilder with the animation, and import them as new Skeletal Meshes in Unreal Engine.



Mesh

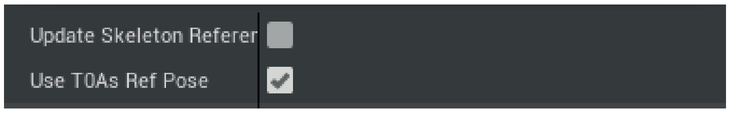
| | |
|---|---|
| Skeletal Mesh | ✓ |
| Import Mesh | ✓ |
| Skeleton | Albert_Skeleton_Special1 |

Figure 17: Select Skeleton in import FBX settings

## FBX Export Issues

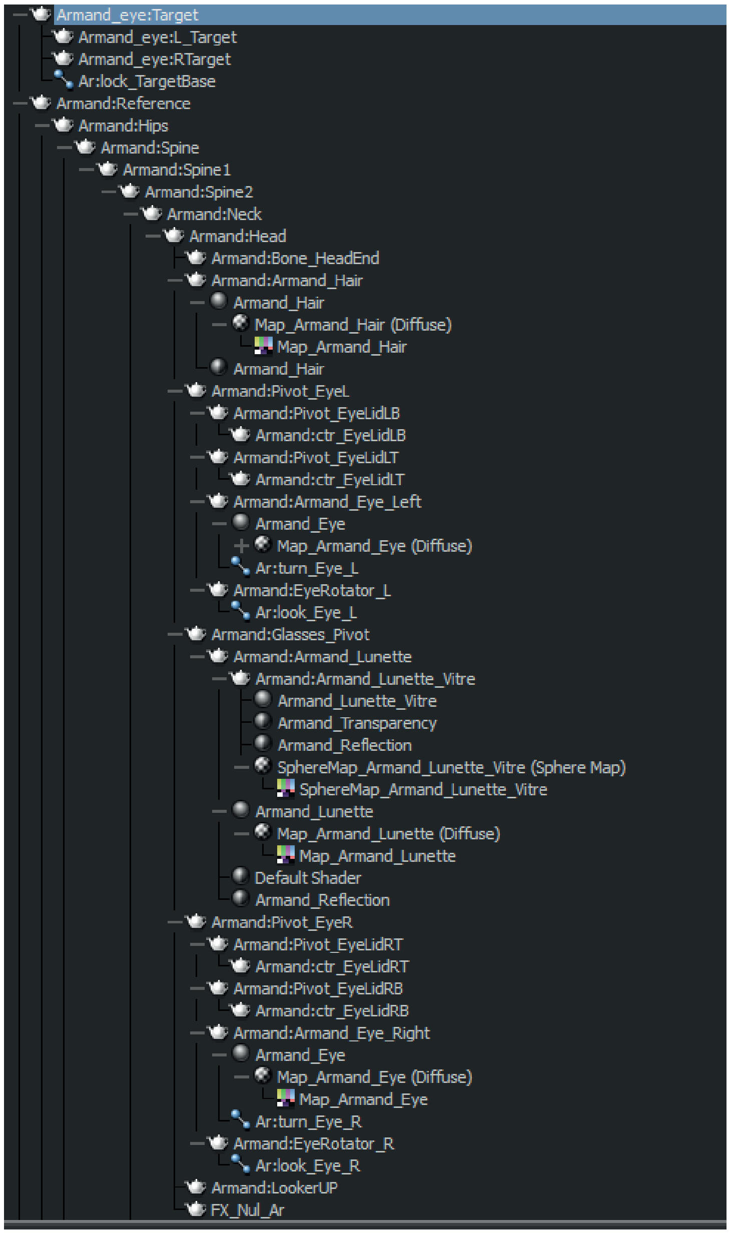
Combining older assets with new caused issues with FBX transfer from MotionBuilder to UE. Sometimes, UE would report an error on import and the arms would appear to be disconnected from the rest of the hierarchy. Vox Populi was able to resolve these problems by putting each character in T-pose and/or setting a TRS keyframe on all nodes before exporting from the DCC.

In MotionBuilder, the best solution is to set up the T-pose manually. In some cases, the option Use **T0As Ref Pose** in the UE4 importer tool can automatically correct the problem.



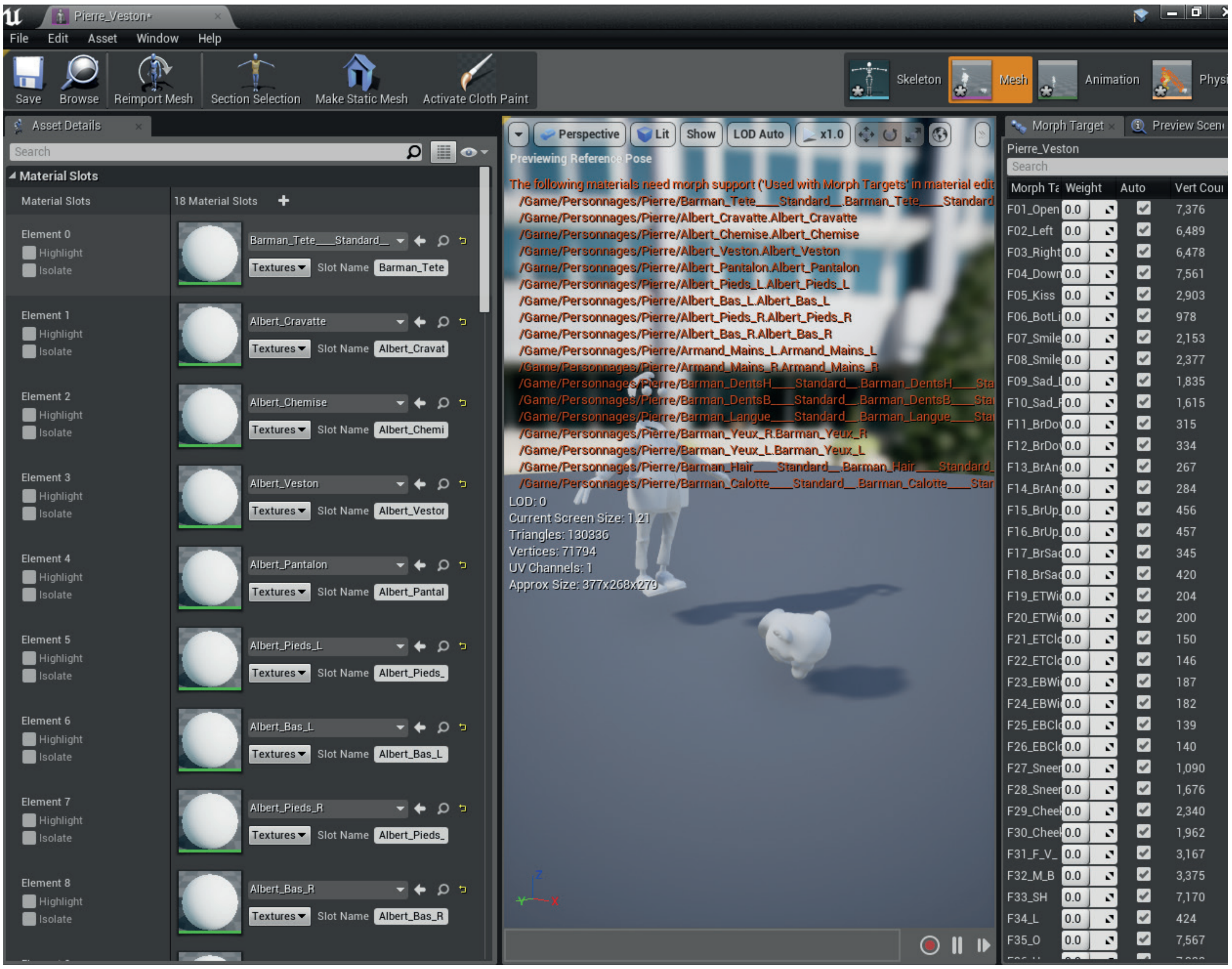
| | |
|---|---|
| Update Skeleton Referer | |
| Use T0As Ref Pose | ✓ |

Figure 18: Select Use T pose in import FBX settings



```
Armand_eye:Target
  Armand_eye:L_Target
  Armand_eye:RTarget
  Ar:lock_TargetBase
Armand:Reference
  Armand:Hips
    Armand:Spine
      Armand:Spine1
        Armand:Spine2
          Armand:Neck
            Armand:Head
              Armand:Bone_HeadEnd
              Armand:Armand_Hair
                Armand_Hair
                  Map_Armand_Hair (Diffuse)
                    Map_Armand_Hair
                Armand_Hair
              Armand:Pivot_EyeL
                Armand:Pivot_EyeLidLB
                  Armand:ctr_EyeLidLB
                Armand:Pivot_EyeLidLT
                  Armand:ctr_EyeLidLT
                Armand:Armand_Eye_Left
                  Armand_Eye
                    Map_Armand_Eye (Diffuse)
                  Ar:turn_Eye_L
                Armand:EyeRotator_L
                  Ar:look_Eye_L
              Armand:Glasses_Pivot
                Armand:Armand_Lunette
                  Armand:Armand_Lunette_Vitre
                    Armand_Lunette_Vitre
                    Armand_Transparency
                    Armand_Reflection
                    SphereMap_Armand_Lunette_Vitre (Sphere Map)
                      SphereMap_Armand_Lunette_Vitre
                  Armand_Lunette
                    Map_Armand_Lunette (Diffuse)
                      Map_Armand_Lunette
                  Default Shader
                  Armand_Reflection
              Armand:Pivot_EyeR
                Armand:Pivot_EyeLidRT
                  Armand:ctr_EyeLidRT
                Armand:Pivot_EyeLidRB
                  Armand:ctr_EyeLidRB
                Armand:Armand_Eye_Right
                  Armand_Eye
                    Map_Armand_Eye (Diffuse)
                      Map_Armand_Eye
                  Ar:turn_Eye_R
                Armand:EyeRotator_R
                  Ar:look_Eye_R
              Armand:LookerUP
              FX_Nul_Ar
```

Figure 19: Character's bone hierarchy in MotionBuilder



Material Slots

| Material Slots | 18 Material Slots | | |
|---|---|---|---|
| Element 0 | Barman_Tete___Standard_ | Textures | Slot Name Barman_Tete |
| Element 1 | Albert_Cravatte | Textures | Slot Name Albert_Cravat |
| Element 2 | Albert_Chemise | Textures | Slot Name Albert_Chemi |
| Element 3 | Albert_Veston | Textures | Slot Name Albert_Vestor |
| Element 4 | Albert_Pantalon | Textures | Slot Name Albert_Pantal |
| Element 5 | Albert_Pieds_L | Textures | Slot Name Albert_Pieds_ |
| Element 6 | Albert_Bas_L | Textures | Slot Name Albert_Bas_L |
| Element 7 | Albert_Pieds_R | Textures | Slot Name Albert_Pieds_ |
| Element 8 | Albert_Bas_R | Textures | Slot Name Albert_Bas_R |

```
Previewing Reference Pose
The following materials need morph support ('Used with Morph Targets' in material edit
  /Game/Personnages/Pierre/Barman_Tete___Standard__.Barman_Tete___Standard
  /Game/Personnages/Pierre/Albert_Cravatte.Albert_Cravatte
  /Game/Personnages/Pierre/Albert_Chemise.Albert_Chemise
  /Game/Personnages/Pierre/Albert_Veston.Albert_Veston
  /Game/Personnages/Pierre/Albert_Pantalon.Albert_Pantalon
  /Game/Personnages/Pierre/Albert_Pieds_L.Albert_Pieds_L
  /Game/Personnages/Pierre/Albert_Bas_L.Albert_Bas_L
  /Game/Personnages/Pierre/Albert_Pieds_R.Albert_Pieds_R
  /Game/Personnages/Pierre/Albert_Bas_R.Albert_Bas_R
  /Game/Personnages/Pierre/Armand_Mains_L.Armand_Mains_L
  /Game/Personnages/Pierre/Armand_Mains_R.Armand_Mains_R
  /Game/Personnages/Pierre/Barman_DentsH___Standard__.Barman_DentsH___Sta
  /Game/Personnages/Pierre/Barman_DentsB___Standard__.Barman_DentsB___Sta
  /Game/Personnages/Pierre/Barman_Langue___Standard__.Barman_Langue___Sta
  /Game/Personnages/Pierre/Barman_Yeux_R.Barman_Yeux_R
  /Game/Personnages/Pierre/Barman_Yeux_L.Barman_Yeux_L
  /Game/Personnages/Pierre/Barman_Hair___Standard__.Barman_Hair___Standard_
  /Game/Personnages/Pierre/Barman_Calotte___Standard__.Barman_Calotte___Star
LOD: 0
Current Screen Size: 1.21
Triangles: 130336
Vertices: 71794
UV Channels: 1
Approx Size: 377x268x279
```

Pierre_Veston

| Morph Ta | Weight | Auto | Vert Cou |
|---|---|---|---|
| F01_Open | 0.0 | ✓ | 7,376 |
| F02_Left | 0.0 | ✓ | 6,489 |
| F03_Right | 0.0 | ✓ | 6,478 |
| F04_Down | 0.0 | ✓ | 7,561 |
| F05_Kiss | 0.0 | ✓ | 2,903 |
| F06_BotLi | 0.0 | ✓ | 978 |
| F07_Smile | 0.0 | ✓ | 2,153 |
| F08_Smile | 0.0 | ✓ | 2,377 |
| F09_Sad_l | 0.0 | ✓ | 1,835 |
| F10_Sad_f | 0.0 | ✓ | 1,615 |
| F11_BrDo | 0.0 | ✓ | 315 |
| F12_BrDo | 0.0 | ✓ | 334 |
| F13_BrAn | 0.0 | ✓ | 267 |
| F14_BrAn | 0.0 | ✓ | 284 |
| F15_BrUp_ | 0.0 | ✓ | 456 |
| F16_BrUp_ | 0.0 | ✓ | 457 |
| F17_BrSa | 0.0 | ✓ | 345 |
| F18_BrSa | 0.0 | ✓ | 420 |
| F19_ETWi | 0.0 | ✓ | 204 |
| F20_ETWi | 0.0 | ✓ | 200 |
| F21_ETCl | 0.0 | ✓ | 150 |
| F22_ETCl | 0.0 | ✓ | 146 |
| F23_EBWi | 0.0 | ✓ | 187 |
| F24_EBWi | 0.0 | ✓ | 182 |
| F25_EBCl | 0.0 | ✓ | 139 |
| F26_EBCl | 0.0 | ✓ | 140 |
| F27_Sneer | 0.0 | ✓ | 1,090 |
| F28_Sneer | 0.0 | ✓ | 1,676 |
| F29_Cheel | 0.0 | ✓ | 2,340 |
| F30_Cheel | 0.0 | ✓ | 1,962 |
| F31_F_V_ | 0.0 | ✓ | 3,167 |
| F32_M_B | 0.0 | ✓ | 3,375 |
| F33_SH | 0.0 | ✓ | 7,170 |
| F34_L | 0.0 | ✓ | 424 |
| F35_O | 0.0 | ✓ | 7,567 |

Figure 20: Error in bone hierarchy. Head attached to an old body

## Organizational and Naming Issues

In the original MotionBuilder pipeline, each character had its own Namespace but might have two or more sub-references to different hierarchies. For example, a character named Foule might have a Namespace that looks like the image below, with two different skeletal hierarchies named Reference 1 and Reference 2.

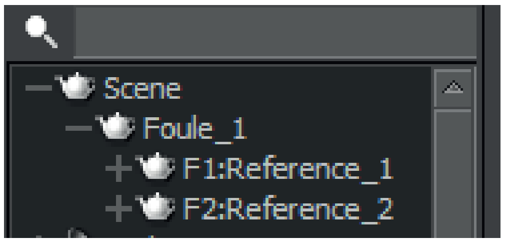


Figure 21: Hierarchy items with namespace

This method of delineating characters and skeletons didn't port well to Unreal Engine for a couple of reasons:

- Internally, MotionBuilder stores these sub-references with a colon (:) between the Namespace name and the reference name. For example, the first sub-reference in the image above would be stored as Foule:Reference 1. This naming convention doesn't work for export via FBX format to Unreal Engine.
- When the FBX file from MotionBuilder is imported, Unreal Engine interprets the entire Namespace as one skeleton. In other words, if the Namespace contains two or more skeletons, they'll be combined into one hierarchy upon FBX import.

A better practice is to set the skeletons as sub-levels just below the Scene level. Unreal Engine will read these hierarchies as separate skeletons.

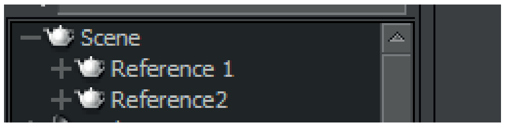


Figure 22: Two skeleton assets with hierarchy division

## Animation

The new pipeline uses MotionBuilder for all animation steps. Mocap animation is corrected and saved in FBX format for import to Unreal Engine. Once in Unreal Engine, animation data is embedded in the characters directly in Sequencer tracks.

The character bodies are animated in MotionBuilder with motion capture data, while faces and hands are animated manually. Props are animated manually or with mocap.

Each scene contains 200 to 6,000 frames at 30 fps, usually animated by a single animator per scene.

## Animation with Texture Switching

Some shots required the switching out of textures, either for an entire shot or repeatedly over the course of an animated sequence. The Laflaque team had a workflow for performing such tasks in MotionBuilder, but wished to implement these techniques in Unreal Engine instead.

## Skin Color

For changing skin color, for example the darkening of skin due to exposure to fire, the Laflaque team had a workflow in the original pipeline using texture blending with custom sliders in MotionBuilder.

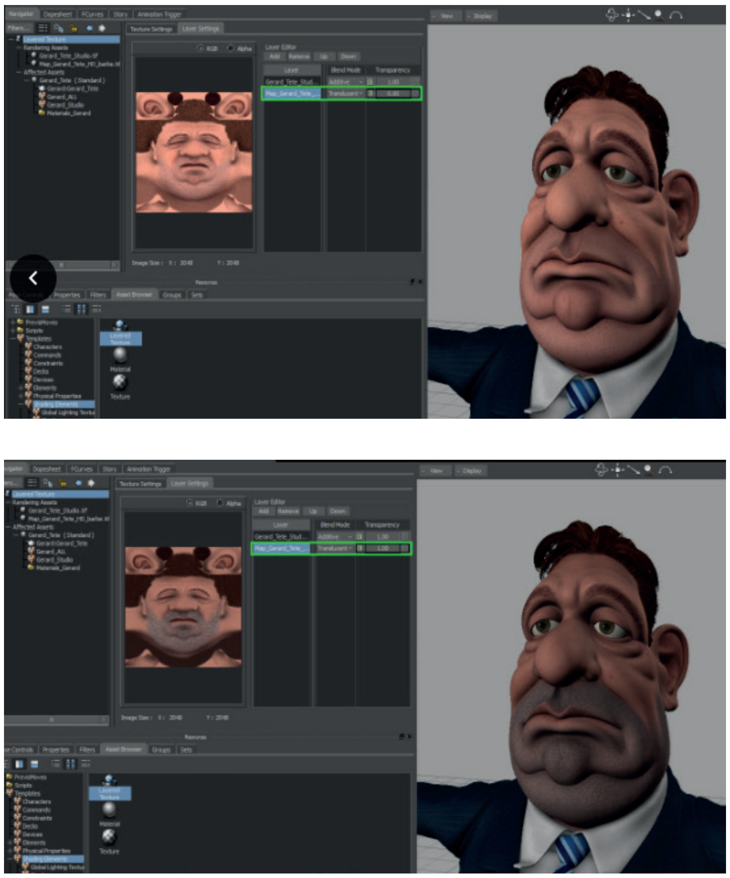


Figure 23: MotionBuilder's Texture Switcher

As part of changing their pipeline, the team needed to set up a similar workflow in Unreal Engine. They did so using the Material Parameter feature, using the following steps:

- Create a new Material Parameter asset in Material Textures/Material Parameter Collection.

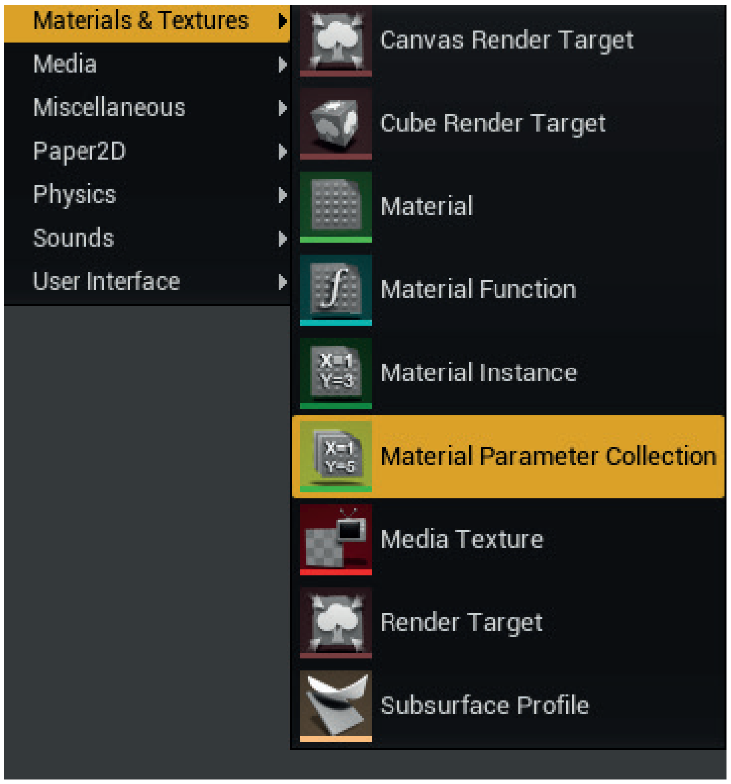
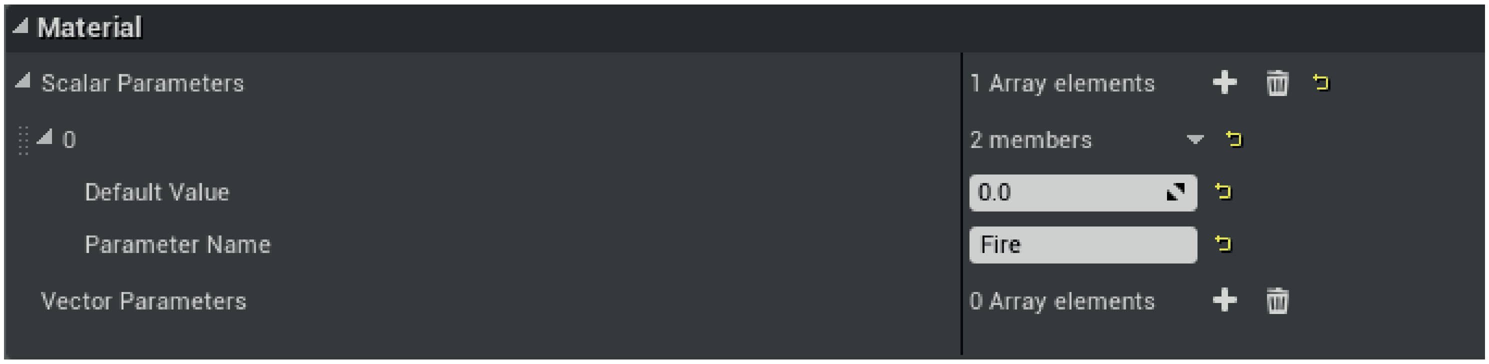


Figure 24: Select Material Parameter Collection in the Content Browser



| Material | |
|---|---|
| Scalar Parameters | 1 Array elements |
| 0 | 2 members |
| Default Value | 0.0 |
| Parameter Name | Fire |
| Vector Parameters | 0 Array elements |

Figure 25: Click the plus symbol next to Scalar Parameters or Vector Parameters

- Add Scalar Value with proper variable name.
- Go to the specific Material Editor.
- Add a collection parameter, and choose your Material Parameter Collection and variable.

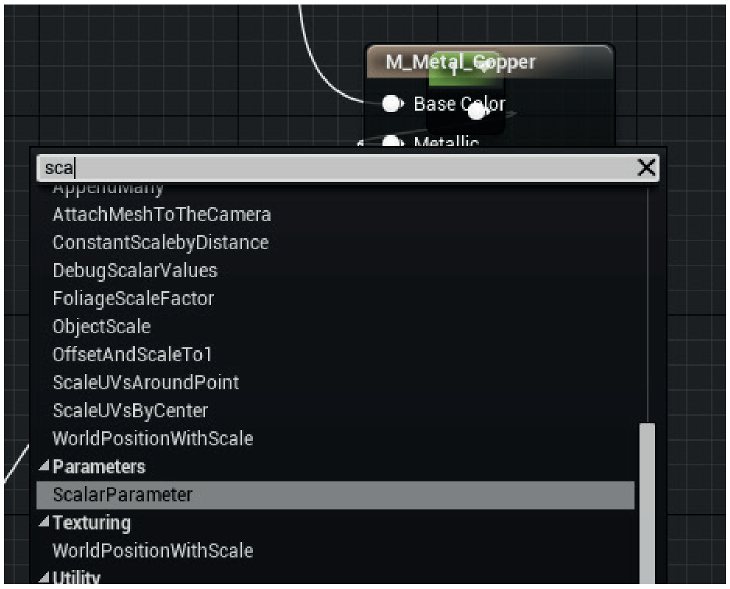
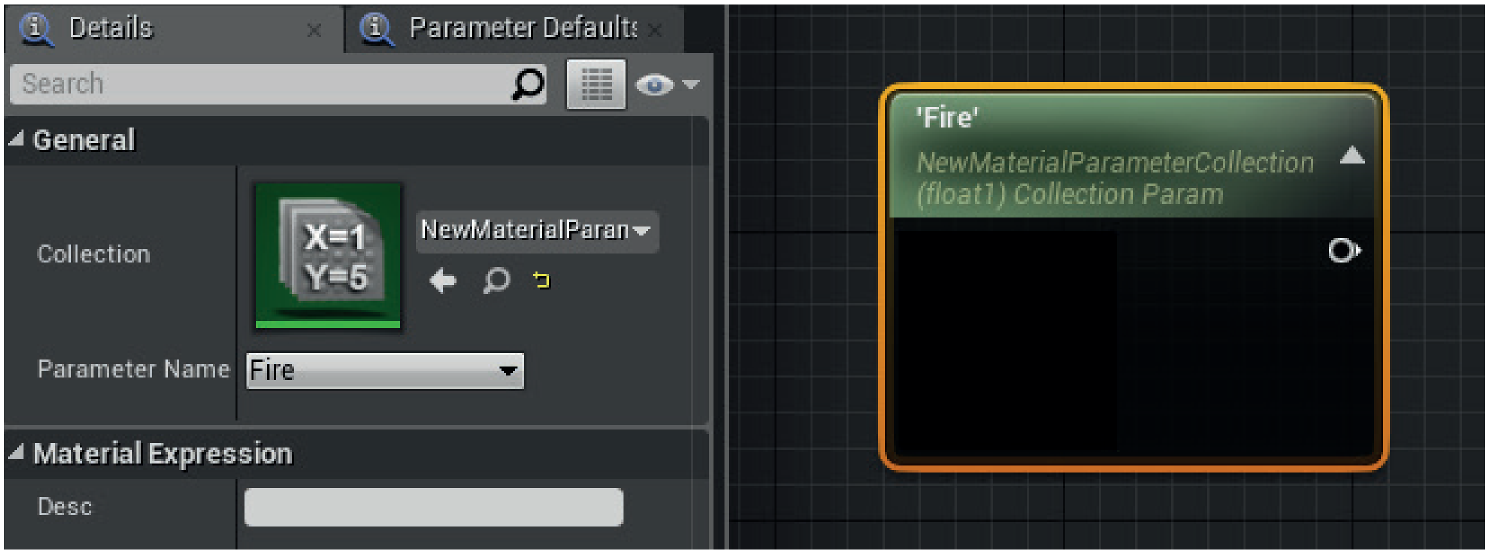


Figure 26: Scalar Parameter Material Expression node in Material Graph

- Create a Linear Interpolate (Lerp) node and add the collection parameter node to Alpha. You can  also use Multiply or other kinds of nodes, depending on the desired blend method.



Collection NewMaterialParan

Parameter Name Fire

Desc

- Add the Material Parameter and variable in Sequencer.

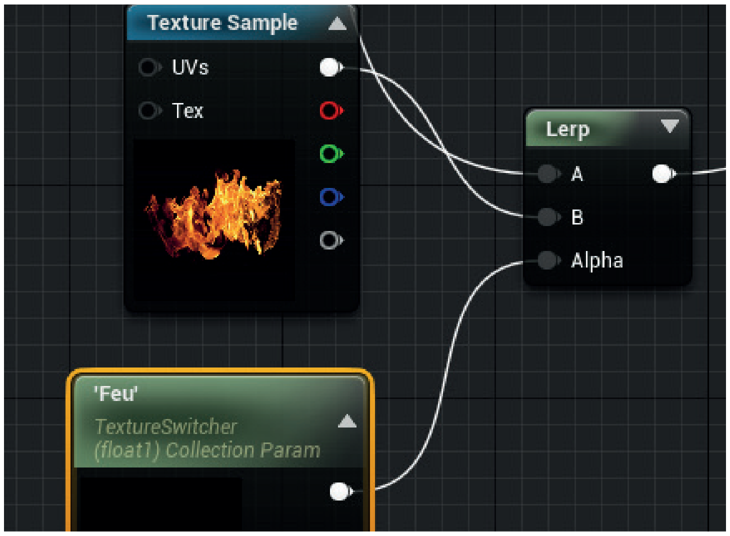
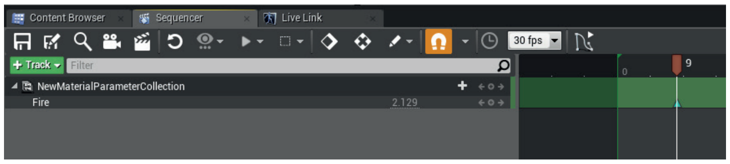


Figure 27: Material Expression Parameter node in Material Graph



30 fps

NewMaterialParameterCollection

Fire 2.129

Figure 28: Material Parameter Collection

## Facial Animation

In some sequences, the team animates facial expressions and dialogue by switching textures. To perform this type of animation with the new Unreal Engine pipeline, the team elected to set up texture switching in MotionBuilder as with the original pipeline, and then redo the setup with a Blueprint in Unreal Engine and import just the animation curves. This approach retained the familiar workflow while still making the animation available for fine-tuning in Unreal Engine.

To set up this type of texture switching In MotionBuilder, a layered texture with more than 80 textures is applied to the head. Each texture is for a different area of the face: eyes, eyebrows, mouth, wrinkles, glasses, and so on. The base texture is the background color, and the topmost texture is the glasses, if any. These two textures are static (not animated) and the remaining textures are animated using visibility and TRS keys[2]. Up to this point, the workflow is the same as the original pipeline.

Once the animation keys have been set, TRS and visibility keys must be set for all the texture channels. This part of the process is newly added for the new pipeline.

Next, the animation is exported to FBX, then imported to Unreal Engine with the Import Custom Attribute and Material Curve Type options checked. This brings in the animation curves.

Figure 29: Animation curves in Editor

In the Material Editor, a material expression was set with the exact name of the animation curves. For the Transition Rule, Blend Logic was set to Linear Interpolation to interpolate between two textures based on a height map and a transition phase value. With such a setup, the parameters change in connection with the animation curves.

The team considered using the Heightlerp function, but that setup would have required some work to separate the eyes from the mouth and other facial features that would have shared the same height value. For this reason, they settled on using Linear Interpolation.

The final iteration of the facial rig uses the actual alpha values of each texture in MotionBuilder, which were transferred into custom curves. Finally, those were used as the Alpha input of each Lerp in the graph. The 80 facial textures all share the same sampler to avoid the technical limitation of 16 samplers.

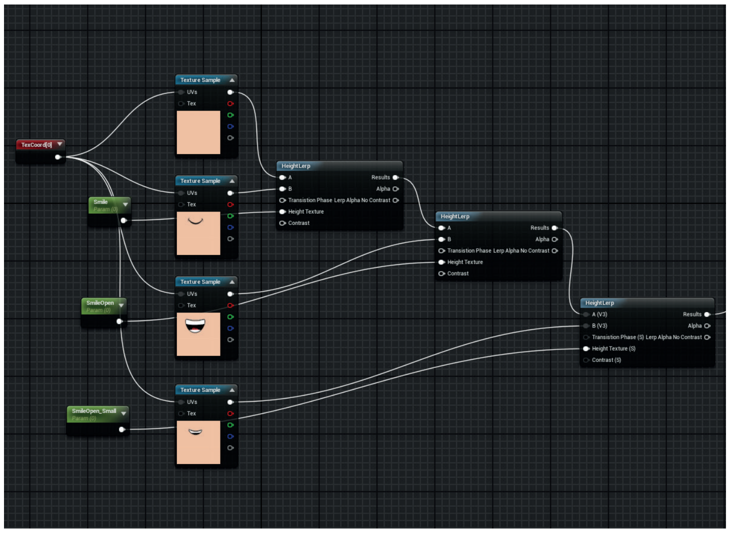


Figure 30: Animation Logic in Material Editor

2 InMotionbuilder, a TRS key sets translation (position), rotation, and scale.

## Use of Sequencer

The team relied on Unreal Engine's Sequencer for many of their tasks that were previously done in MotionBuilder—production, previs, viewing, and rendering.

In the Sequencer, the team defined a Persistent Level consisting of sublevels in order to facilitate the work (see Table 3). The Level Visibility track in Sequencer is used to manage the visibility of a group of objects. The team found the ease of hiding and unhiding objects to be very useful during production.

The following figure shows an example of tree nomenclature of an episode in sequence:

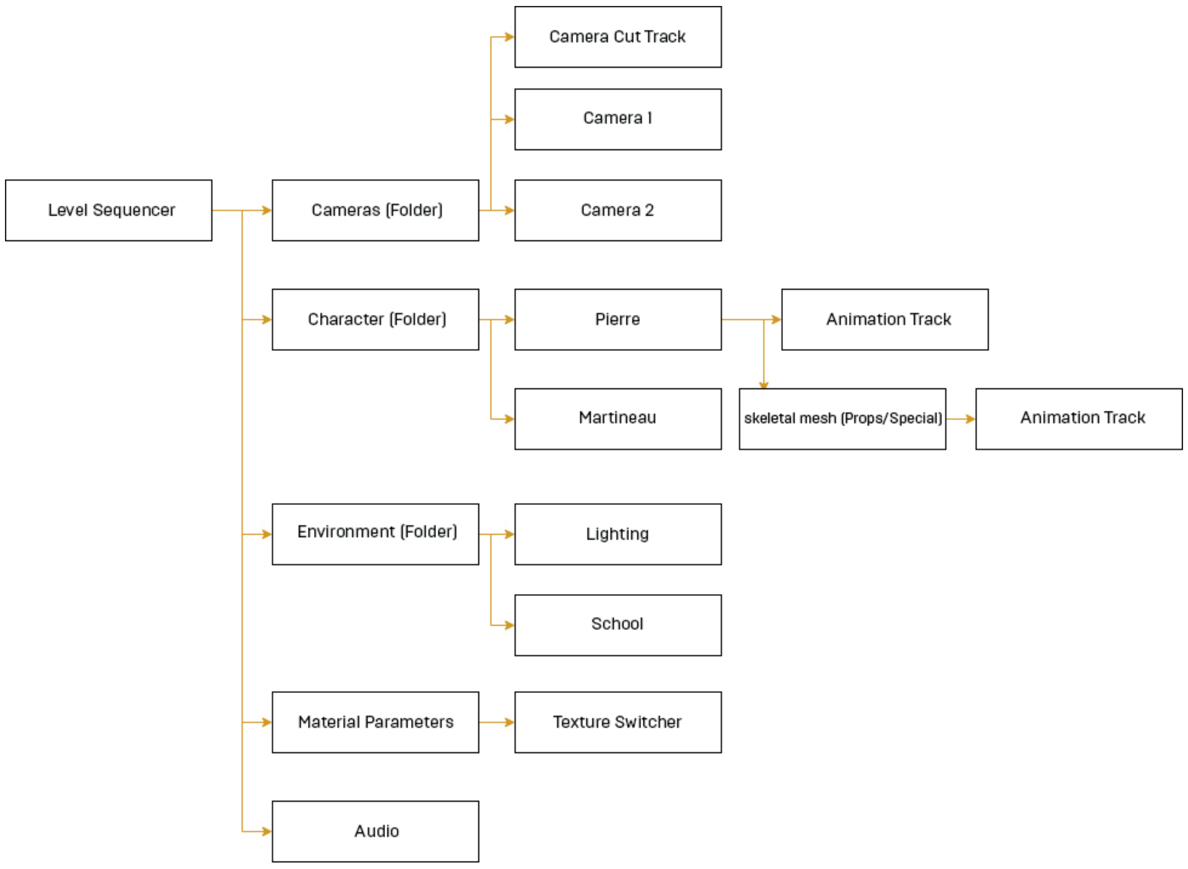


Figure 31: Laflaque Sequencer nomenclature

The team made use of Unreal Engine Sequencer to trigger events during play. This automated some of their production processes, such as changing material parameters.

To set up the Blueprint variable to be changed directly in Sequencer:

- Create an Actor Blueprint.
- Create a Custom Event, and enable Call in Editor.
- Create a variable, and enable Expose to Cinematics.
- Add the Actor Blueprint and its variable in Sequencer.
- In Sequencer, create an Event Track.

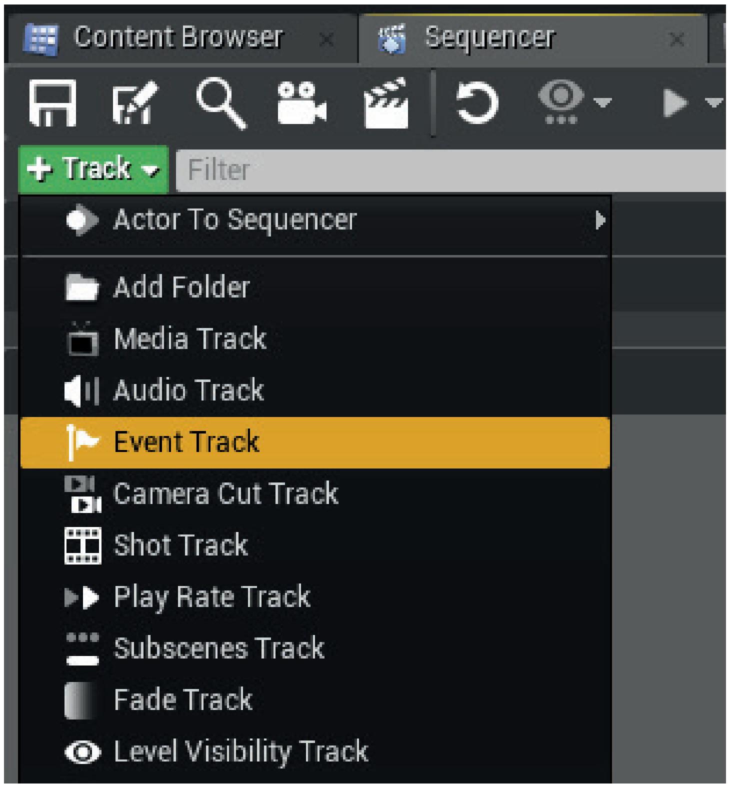


Figure 32: In Sequencer, Create Event Track

- In Sequencer, create a  key in the Event Track at the appropriate time. Right-click on the key, select Properties, and under Event Name, enter the appropriate event name.

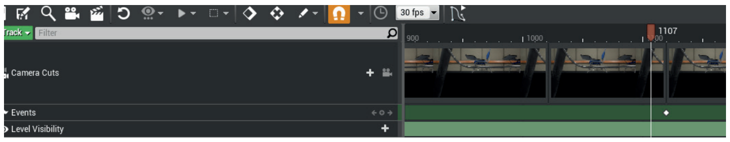


Figure 33: Create event key in the event track

- In Sequencer,  right-click on the Event Track and select Properties. Under Event Receiver, choose the appropriate Blueprint. Otherwise, the event keys will trigger only the Level Blueprint.

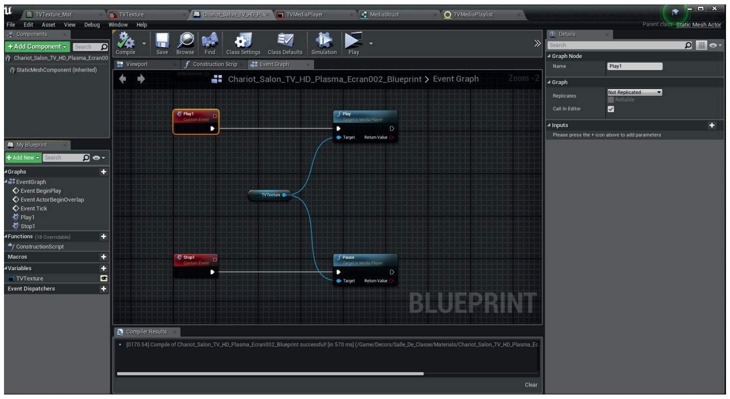


Figure 34: Create custom event

## Lighting

Lighting in MotionBuilder is limited, which meant the team had to use a number of workarounds to retain a real-time workflow in the original pipeline. Environments were lit in 3ds Max, and lighting was baked directly into textures. Only characters and animated props were lit in MotionBuilder. The baked environment textures limited the team's ability to change or adjust lighting during the animation and visual effects phases.

The team sought to improve the show's lighting workflow by using Unreal Engine lights, which are optimized for real-time rendering. In the new Unreal Engine pipeline, both environments and animated elements are lit with the same lights, and the team now has the ability to adjust lights at any time during production.

For realistic lighting with high-fidelity rendering, static lighting is the best approach in Unreal Engine. This provides access to features traditionally done with ray tracing, such as GI, Final Gathering, and area lights. It is much easier to achieve realistic lighting without needing to mimic complex behaviors manually using extra lights.

If realism is not a requirement (as with a stylized scene), dynamic lighting can be used to take advantage of interactive feedback when lighting the scene.

For different rendering features, UE4 has several techniques for approximating lighting and shadows. For static lighting, there are no limitations in terms of supported light types or rendering features—basically everything available in the Editor will work once it's baked.

Dynamic lighting, on the other hand, presents some challenges. The following are few examples of techniques for dynamic shadows:

- For sharp shadows, use shadow maps. This works with all object types.
- For soft ray-traced shadows, use Distance Field Shadows. This does not work well with skeletal meshes.
- For soft shadows on characters, use Capsule Shadows. This helps to add contact shadows around characters' feet.

## Rendering

And finally, we get to the main point of Vox Populi's switch to Unreal Engine: rendering. In the original pipeline, the team used MotionBuilder with OpenGL for real-time previews, and to render final output in near-real-time. Each of the scenes were rendered in a single pass in MotionBuilder using the QuickTime MOV DNxHD codec with audio.

## Antialiasing

There are three antialiasing techniques used in game engines: MSAA, FXAA, and TemporalAA. Usually TemporalAA is recommended, but it can generate some artifacts depending on its performance curve, and can result in blurriness. The FXAA method gives excellent performance in terms of quality but does not allow the use of depth-of-field and motion blur, which limits post processes.

The Laflaque team chose to use the TemporalAA method in Unreal Engine. However, in their first tests, TemporalAA resulted in noise on straight lines, as shown in the following image.

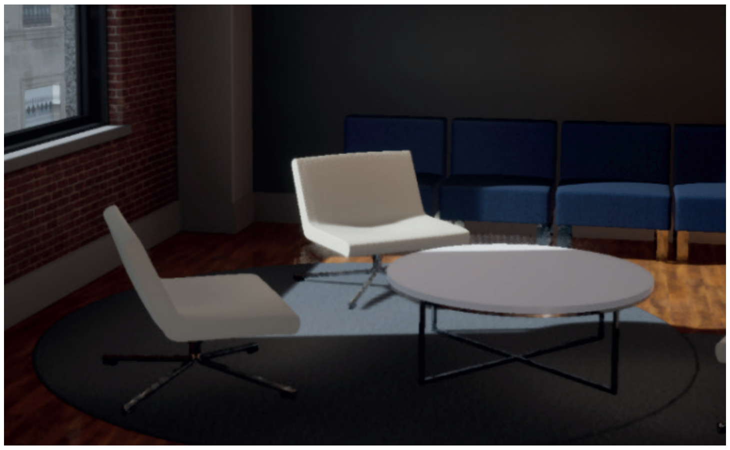


Figure 35: Without console adjustment

After some tests, the team defined custom adjustments in the Console command to optimize rendering.

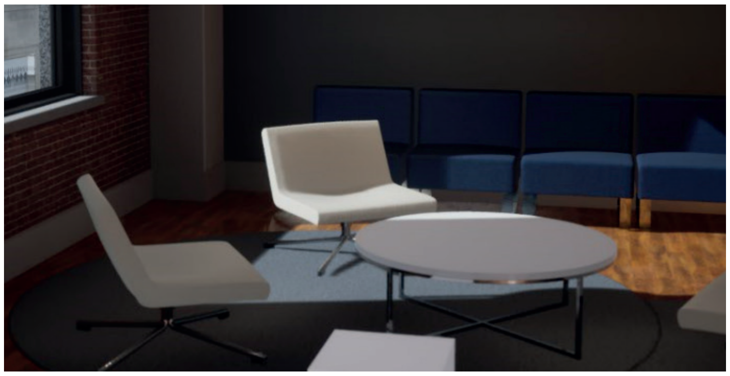
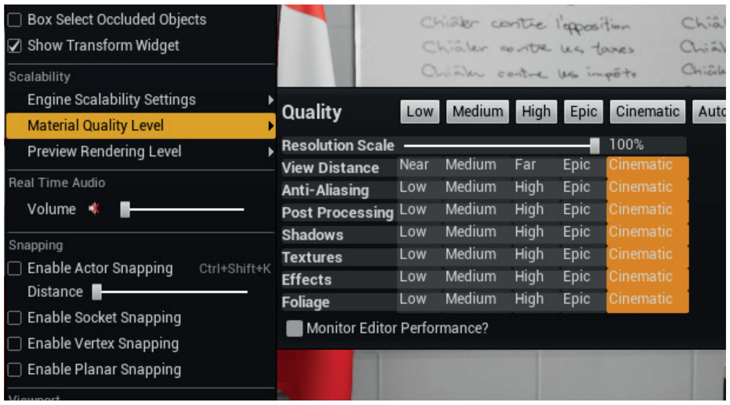


Figure 36: With Console adjustment

Below is an example of a command Console setup. The team usually tweaks the settings on a case-by-case basis, but the settings below provide a baseline.

- r.MotionBlurSeparable = 1
- r.MotionBlurQuality = 4
- r.Streaming.PoolSize=1000
- r.ScreenPercentage = 200 (Note that increasing this setting from 100 to 200 will increase render time by 4x.)
- r.PostProcessAAQuality = 6
- r.Tonemapper.Sharpen = 1
- r.TemporalAASamples = 4
- r.TemporalAACurrentFrameWeight = 0.2
- Scalability Settings on the highest setting (Cinematic) for best quality
- In Sequencer's Rendering Options, uncheck the Use Compression option for best quality



☐ Box Select Occluded Objects
✓ Show Transform Widget

Scalability
Engine Scalability Settings
Material Quality Level
Preview Rendering Level

Real Time Audio
Volume

Snapping
☐ Enable Actor Snapping Ctrl+Shift+K
Distance
☐ Enable Socket Snapping
☐ Enable Vertex Snapping
☐ Enable Planar Snapping

Quality: Low | Medium | High | Epic | [✓ Cinematic] | Auto

Resolution Scale: 100%

| | | | | | |
|---|---|---|---|---|---|
| View Distance | Near | Medium | Far | Epic | [✓ Cinematic] |
| Anti-Aliasing | Low | Medium | High | Epic | [✓ Cinematic] |
| Post Processing | Low | Medium | High | Epic | [✓ Cinematic] |
| Shadows | Low | Medium | High | Epic | [✓ Cinematic] |
| Textures | Low | Medium | High | Epic | [✓ Cinematic] |
| Effects | Low | Medium | High | Epic | [✓ Cinematic] |
| Foliage | Low | Medium | High | Epic | [✓ Cinematic] |

☐ Monitor Editor Performance?

Figure 37: Cinematic quality setup

## Rendering Quality Comparison

Here are the comparative results of renderings from MotionBuilder OpenGL vs. Unreal Engine.

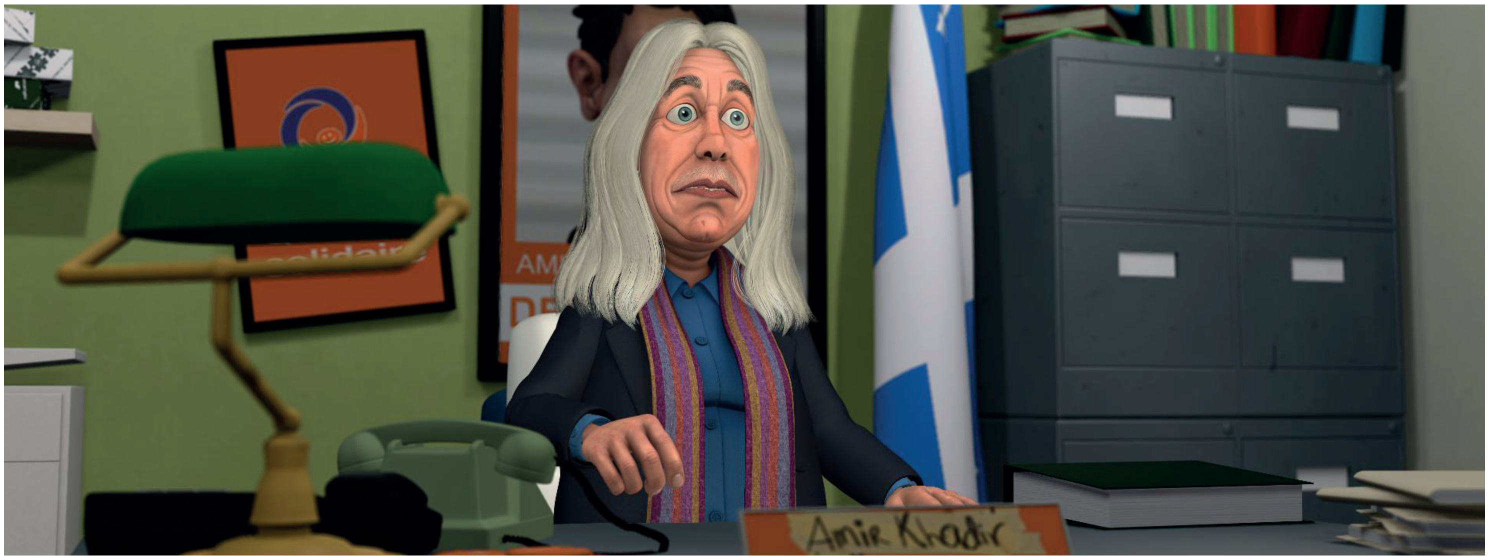


Figure 38: Scene rendered with MotionBuilder

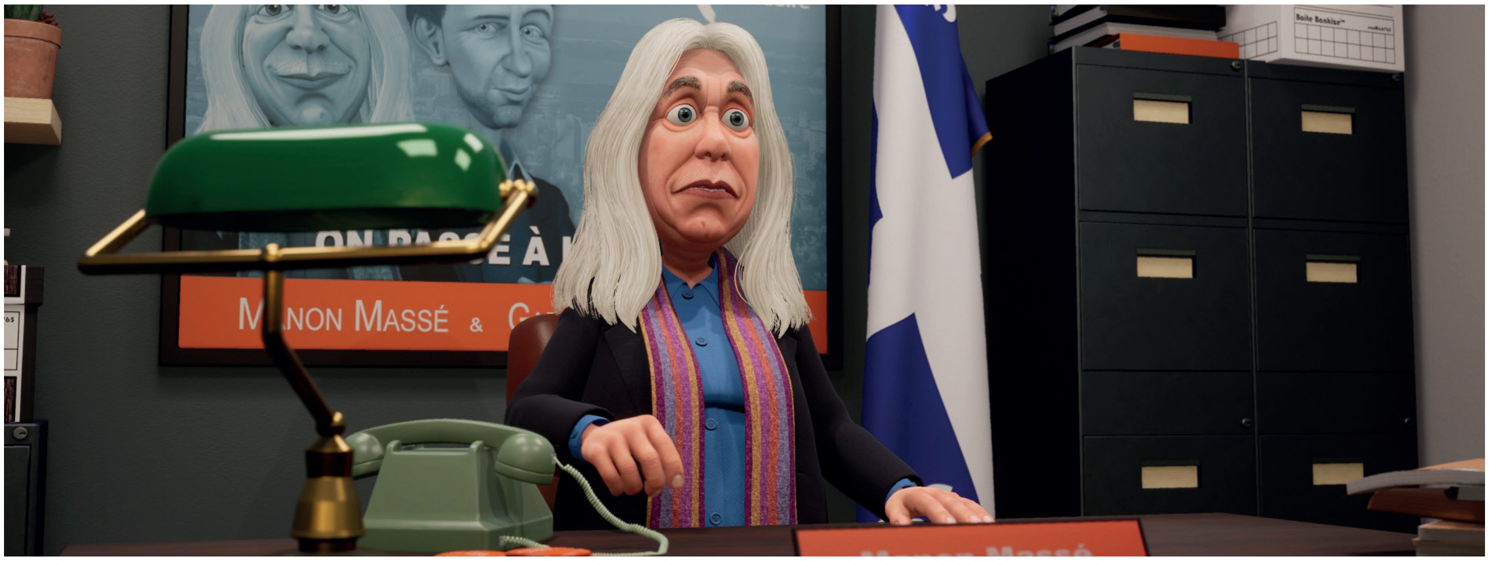


Figure 39: Similar scene rendered with Unreal Engine

The images from Unreal Engine have richer colors and a cleaner look. To Laflaque's viewing audience, these technical improvements are transparent—the show just looks better than before, making for a more enjoyable viewing experience.

## Editing and Delivery

In both the original pipeline and new pipelines, the rendered frames pass to a final editing process in Avid Media Composer, and the audio is mixed in Pro Tools. Composition of additional layers like add-on screens, textures, particles, etc. are done with After Effects.

When the episode is finished, the final version is delivered to the Radio-Canada broadcast network as a Quicktime MOV file compressed with the Avid DNXHD 110 codec, using 29.97 FPS Drop Frame in 720p.

## Future Vision

## Future Vision

Phase 1 of the switch to a UE4 pipeline with the Laflaque show has been set up to make production as familiar as possible while avoiding radical changes in the original pipeline. The future vision for Phase 2 would be to optimize the parallel use of Unreal Engine and MotionBuilder.

Access to MotionBuilder Live Link will give productivity upgrades. Instead of using two MotionBuilder scenes, mocap data would link directly to an Unreal layout scene with basic characters. This solution will enable real-time viewing of the scene at the very beginning of motion capture with PhaseSpace. MotionBuilder Live Link will also serve as a live feed for animators so they can correct the capture in MotionBuilder and view corrections in real time in Unreal Engine.

Vox Populi also looks forward to taking advantage of the Unreal Engine plugin for Shotgun integration, and new features like ray tracing solutions.

# About this Document

# About this Document

## Production Schedule

Pre-Production: Feb 2018

Production: June 2018

## Team Composition

Roxane Boutet, Producer

Yves St-Gelais, Executive Producer

Richard Belec, Production Director

Tom Wilczynski, Technical Director

Cédric Dubois, Character Modeling Supervisor and Lead Artist

Francois Bissonnette,  3D artist and Technical Director for Unreal Engine

Hugo Brodeur, Artistic Director, Environments

Marco Marandola, Senior Lighting and Rendering Artist

Mathieu Langlois, 3D Director

Approximately 10 animators and layout artists

Editorial Team

## Author

David Hurtubise

## Editors

Michele Bousquet

Brian Pohl

## Contributors

Tom Wilczynski

Cédric Dubois

Francois Bissonnette

Homam Bahnassi

Original images and data supplied by Productions Vox Populi 1 Inc.