



Image courtesy of McLaren Automotive

Accelerating Data Conversion and Visualization

Automating CAD data preparation and real-time visualization using Datasmith

Updates to this paper

The original version of this paper was published in 2018 with Unreal Engine 4.20 in mind. Since then, many important updates have been made to Unreal Engine. While the paper has been edited to reflect many of these updates, additional new features are available that pertain to the workflow described:

- [Variant Manager](#) is now the recommended tool for authoring and managing product variations within Unreal Engine.
- The Datasmith API has evolved, and now includes function calls to import CAD files and the ability to re-tessellate NURBS data. See the help topic [Customizing the Datasmith Import Process](#) for more information.
- For automating 3D model preparation, the [Visual Dataprep](#) system offers an alternative to Python and Blueprints.
- The Product Viewer project template has evolved into the [Collab Viewer template](#), which offers additional multi-user capabilities.
- The Photo Studio template offers ready-to-use studio environments for automotive visualization.
- The original version of the paper made references to Unreal Studio, a suite of tools to operate within Unreal Engine. The entirety of Unreal Studio has since been integrated directly into Unreal Engine.

Contents

Introduction	3
Automating CAD Data Preparation and Real-Time Visualization with Datasmith	4
Traditional Data Preparation Challenges	4
Leveraging Datasmith's Advantages for the McLaren Design Team	6
Understanding Data Preparation	7
CAD vs. Triangles	7
Tessellation	9
Optimizing for Speed	14
Data-Prep Strategy and Best Practices	19
Automation in Practice: McLaren Design Team	22
Tailoring a Solution	22
Automation with Python	23
Sample Python Scripts	24
McLaren Design Visualization Application	28
Navigation	28
Custom Blueprints for Car Parts	30
Graphical User Interface	32
Unreal Engine as a Platform	34
About this White Paper	35
Authors	35
Contributors	35



Introduction

Unreal Engine is becoming the go-to visualization tool for manufacturers for its flexibility, visual quality, multi-platform support, and ease of use. Designers in manufacturing are finding that with real-time rendering, they can go beyond still images and video to produce interactive media of many kinds across several platforms.

To support this growing user base in manufacturing, Epic Games has recently added data preparation features as part of its Datasmith import tool. Datasmith enables a direct-from-CAD workflow, creating a uniquely powerful end-to-end solution for manufacturing visualization.

In early 2018, Epic Games collaborated with McLaren Design to embark on a real-time visualization project: a car viewer configurator for a variety of car types (sports, classic, luxury) that would be updated with new design information as the designs evolve.



Automating CAD Data Preparation and Real-Time Visualization with Datasmith

For the McLaren project, the goal was to automate the preparation, conversion, and visualization of design data using Datasmith. This automation dramatically reduces bottlenecks in converting, organizing, and preparing CAD data within the engine. As a result, the design team was able to seamlessly incorporate real-time rendering and virtual reality into their regular workflow.

Maintaining the accuracy of the data and creating new efficiencies in data preparation were both key. Once the preparation workflow was established, the developer was then able to use the Python integration within Unreal Engine to automatically read and process new designs.

This document outlines the steps taken in the McLaren workflow to produce this project, as well as core concepts and techniques that can be applied to your own company's unique workflow and needs.

Traditional Data Preparation Challenges

Traditional approaches to data preparation present several challenges to design teams.

Traditional solutions call for a dedicated operator to convert, organize, and visualize the data. These processes are time-consuming, and the operator and his/her processes are disconnected from the design team. Quick review, iteration, and presentation are not possible with such a workflow.

In addition, CAD, 3D files, and metadata are often entrenched in formats exclusive to their applications, where they can only be accessed by a dedicated operator. This means designers and other interested personnel can't easily see or update this data.

When interactive media is produced, an additional software package is typically required for viewing. This creates a cost and logistics barrier in communicating with external teams and customers, as additional seats and trained staff are needed for each individual experiencing the product in real time or VR.

Unreal Engine and Datasmith are designed specifically to solve these problems. With Datasmith, the conversion process is simplified to the point where a dedicated operator is no longer necessary.

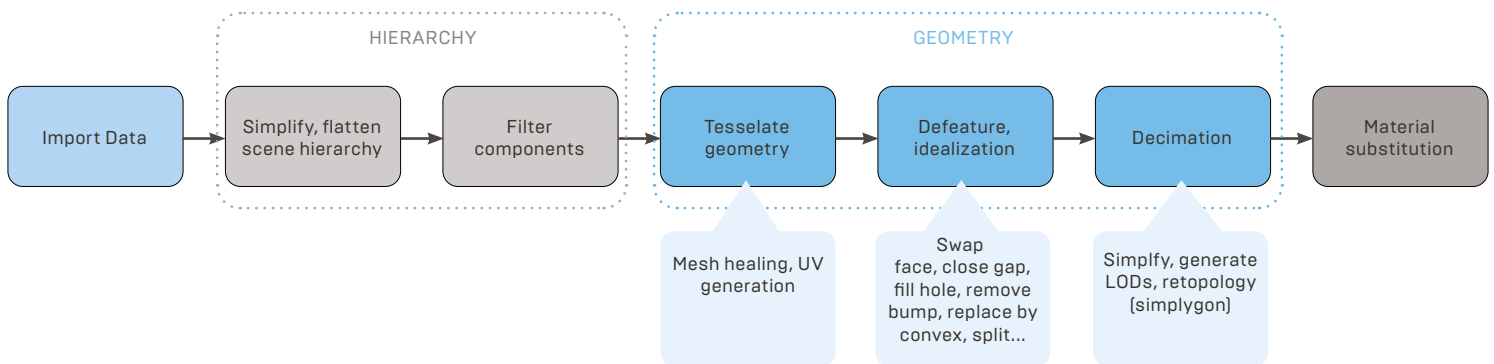


Figure 1: Import and optimization workflow from CAD to Unreal Engine



Figure 2: Unreal Engine as a central platform

In Unreal Engine, data is stored and edited in a centralized hub where many team members can access, view, or update it as needed.

Unreal Engine also produces media that requires no additional software for delivery and viewing. Users can deploy standalone applications for distribution to their organization and customers.

Leveraging Datasmith's Advantages for the McLaren Design Team

In understanding project needs with the McLaren team, we first discussed functionality and efficiency goals. In order to more effectively integrate real-time 3D vehicles into their design workflow, McLaren required the following workflow features:

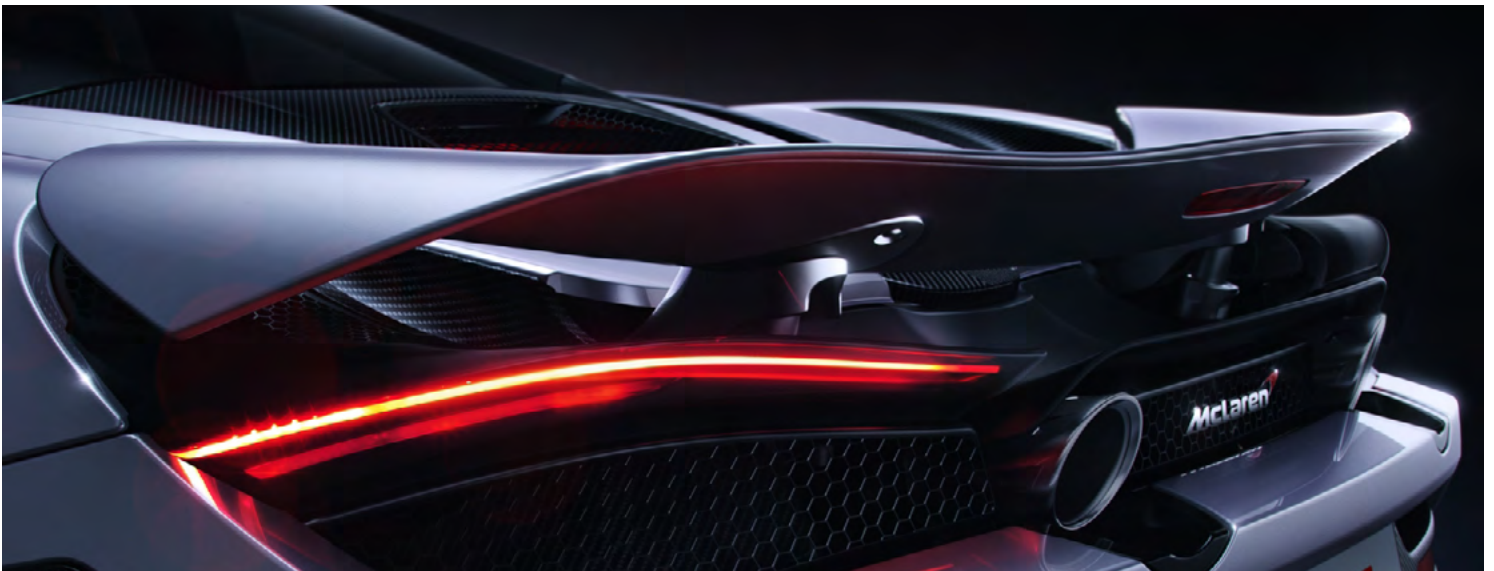
- Visualize proposed vehicle surface form and materials at a high level of product accuracy.
- Easily compare a wide range of product variations and past designs.
- Enable quick iteration between design and visualization, making it possible to view and compare new designs in Unreal Engine once they are made or modified in the design program.

A traditional barrier for McLaren in adopting visualization data-prep solutions was the manual staff overhead required to implement them. The following efficiencies were also needed to ensure that the solution worked well in practice under resource constraints:

- Ability to automate CAD translation and import process so McLaren's real-time experts can focus on creative decisions.
- Any file-saving/versioning process must fit with existing processes used within the design department.
- Ability to automatically process and integrate new designs as changes are made.
- Regarding ease of use, designers must be able to save, view, and compare their designs in real-time review or virtual reality without needing technical knowledge of Unreal Engine or other software.

In tailoring a solution to meet McLaren's goals, we established the needs, best practices, and application settings for preparing their CAD data for the purpose of design collaboration in Unreal Engine.

The following sections of this document will outline some techniques and concepts used for data preparation. We will then go over some details on how the automation and functionality was developed in practice.





Understanding Data Preparation

This section of the document outlines key concepts in preparing design data for use in a real-time visualization tool such as Unreal Engine.

Datasmith, an import tool that comes with Unreal Engine, handles the conversion of CAD data into Unreal Engine entities.

Datasmith offers support for importing files from a wide range of CAD, CAID, and DCC applications, including DELTAGEN, VRED, Alias, Revit, AutoCAD, CATIA, 3ds Max, and SketchUp Pro. A full list of supported formats can be found in the [Datasmith Supported Software and File Types](#) help topic.

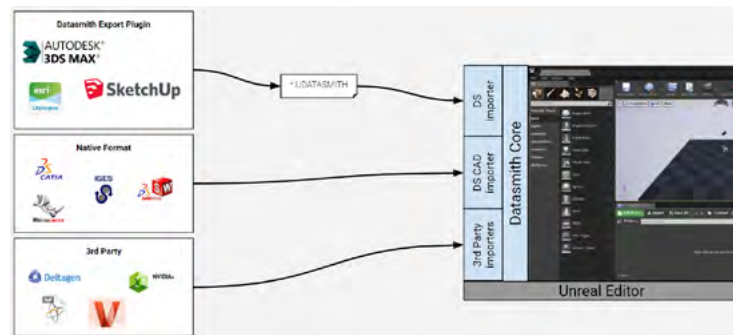


Figure 3: CAD and DCC packages and their import workflows to Unreal Engine

CAD vs. Triangles

In the industry, there are two main categories of format: precise and tessellated. While renderings of these formats can appear very similar, the formats differ in a number of ways:

Precise

- Solids and their surfaces are described as parametric functions and thus are geometrically exact.
- The model contains construction information that can be used to do filtering.
- This is the type of model typically produced by a CAD system. Such models are not suitable for real-time visualization.

Tessellated

- Surfaces are represented by a mesh, a structured representation using triangles (also known as polygons or geometry). The triangles are “stitched” together in a mesh that represents the surface of an object. By using triangles of small dimensions, it is possible to approximate a curved surface so that it appears smooth from a given distance.
- All construction information is lost.
- Tessellated models are suitable for real-time visualization.

CAD systems are tools for creation and organization of the Digital Mock-Up (DMU) of a product. The end use of the DMU is a precise model optimized for production, but not well suited for interactive visualization.

CAD Conversion

A fundamental step in preparing the data is converting engineering optimal data (CAD) into data that works efficiently for visualization (DCC). 3D engines only work on triangle models; a tessellation step is therefore necessary if the format is precise.

CAD models also often contain hierarchies, assemblies, and other organizational structures that need to be retained in any conversion to a game engine format. Conversion with Datasmith preserves the hierarchy of the part’s structure, and performs tessellation.

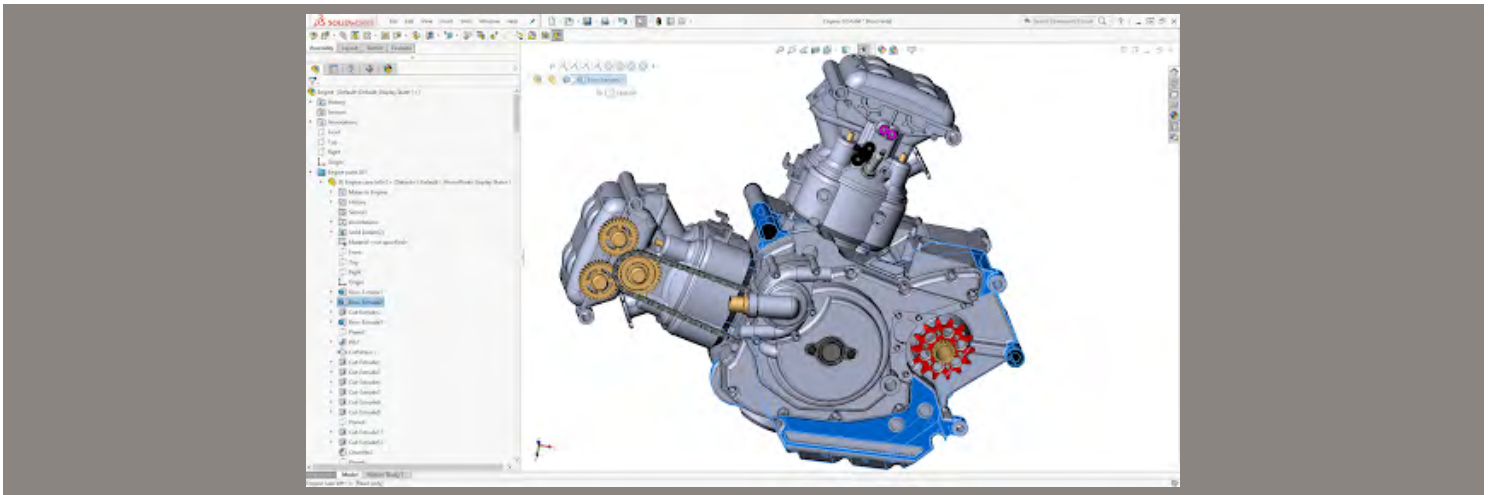


Figure 4: Screenshot of the CAD design environment. The final model results from a succession of mathematical operations represented as a graph (history of construction).

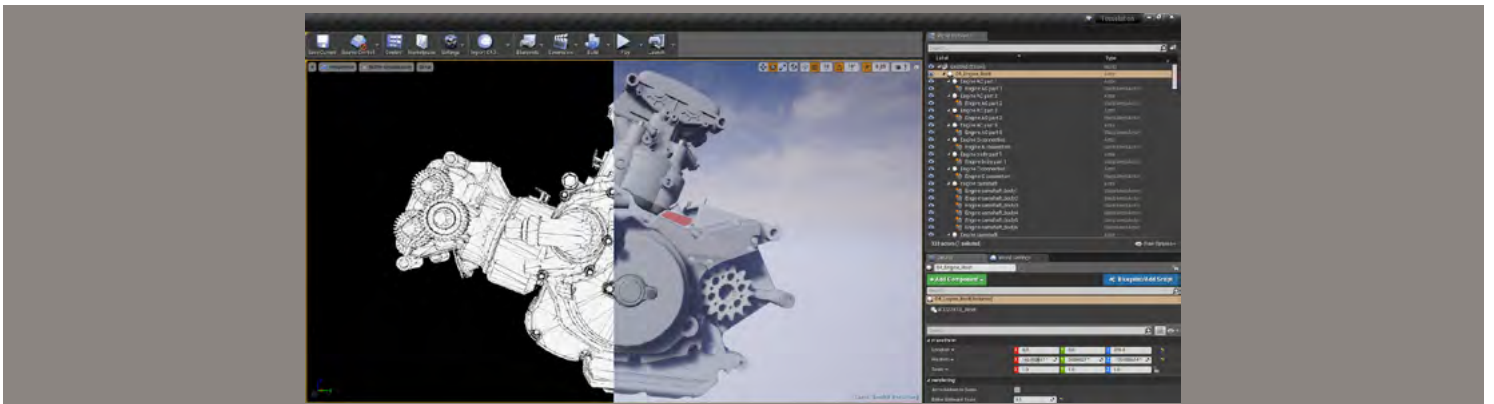


Figure 5: Screenshot of the same engine in Unreal Editor. The model is an assembly of triangle meshes, with each part of the original model as an independent triangle mesh.

DCC Conversion

Most DCC (Digital Content Creation) software packages (Maya, 3ds Max, etc.) use triangle-based models, so files coming from a DCC package generally do not need to be tessellated before being imported to a real-time engine.

However, DCC formats sometimes do not export information on history of construction, or on features local to the DCC package. While software exists to perform mesh analysis and compensate for these shortcomings in preparing a model for a game engine, such analysis is time-consuming.

The conversion process from a DCC format to a game engine must take into consideration this requirement to preserve local features in the resulting mesh.

Types of Unreal Engine Meshes

When an object is imported into Unreal Engine, it is called a Static Mesh. The Static Mesh is also part of a larger group of object types called Actors. An Actor is any type of object (mesh, camera, clickable icon, etc.) that can be placed in an Unreal Engine scene.

These terms are introduced here for clarity and assistance in understanding some of the concepts in this document.

Historically, the triangle has been the primitive geometric form of choice for rendering algorithms. It is the simplest geometric primitive that can be used to approximate a surface, and is straightforward to use in 3D mathematics and rendering to calculate position in space, orientation, and projection to screen. This is why geometric forms used in real-time applications are represented as a set of triangles.

Tessellation

The tessellation step has two purposes: to convert precise surfaces to a triangle model (sampling), and to reduce the model's level of detail. The advantage of precise models is that they can be tessellated to the level of detail needed by the user.

By sampling the precise surface, the conversion process generates an approximated version of the original surface. The user can parameterize the sampling frequency and thus impact the quality/size of the approximated output. In the context of surface tessellation, the parameters control the density of triangles used to represent the surface. Several parameters are available in the CAD importer, each with a different impact on triangle generation.

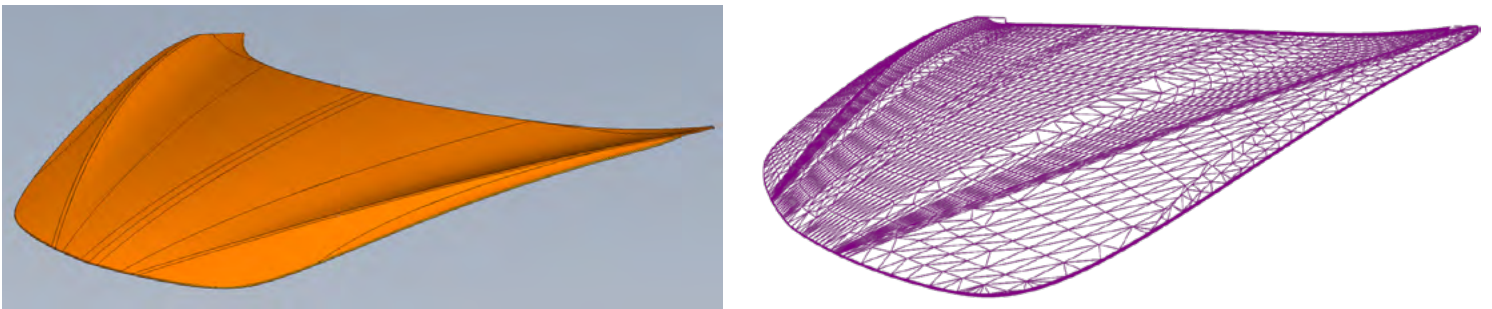


Figure 6: On the left, a native NURBS-based file representing a vehicle hood rendered in a CAD viewer. On the right, the same surface sampled to a triangle mesh.

Effective tessellation is characterized by:

- Sufficient sampling to generate a triangle density that will capture and represent surface details.
- Overall uniformity in triangle size. Elongated or improperly formed triangles should be avoided in the mesh. Uniformity of triangle size is essential to guarantee smooth shading and reflection on the mesh.

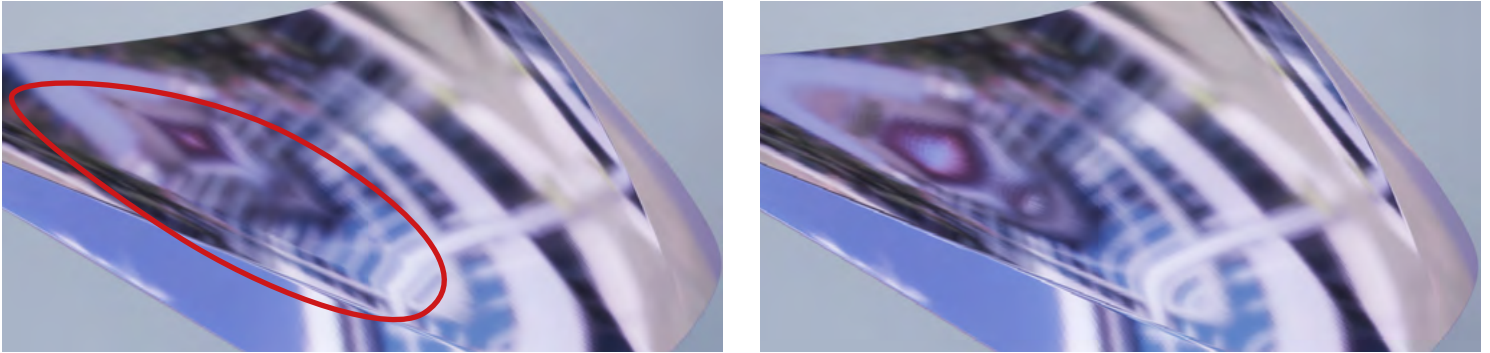


Figure 7: Low triangle density in mesh on the left leads to discontinuities in the reflection. The mesh on the right has a more appropriate mesh density and thus a more accurate reflection.

The level of detail on the converted model will depend on the parameters used for tessellation.

Three parameters control the accuracy of the tessellation: Chord Tolerance, Max Edge Length, and Normal Tolerance.



Figure 8: Tessellation parameters in the Datasmith Import Options dialog

Chord Tolerance

Chord Tolerance, also called chord error or sag, is the parameter of choice to simplify the model. It defines the tolerance between the initial surface and the tessellated surface, the maximum acceptable distance between the two surfaces. The lower this value, the higher the number of samples taken from the original surface, and thus the closest fit to the original surface.

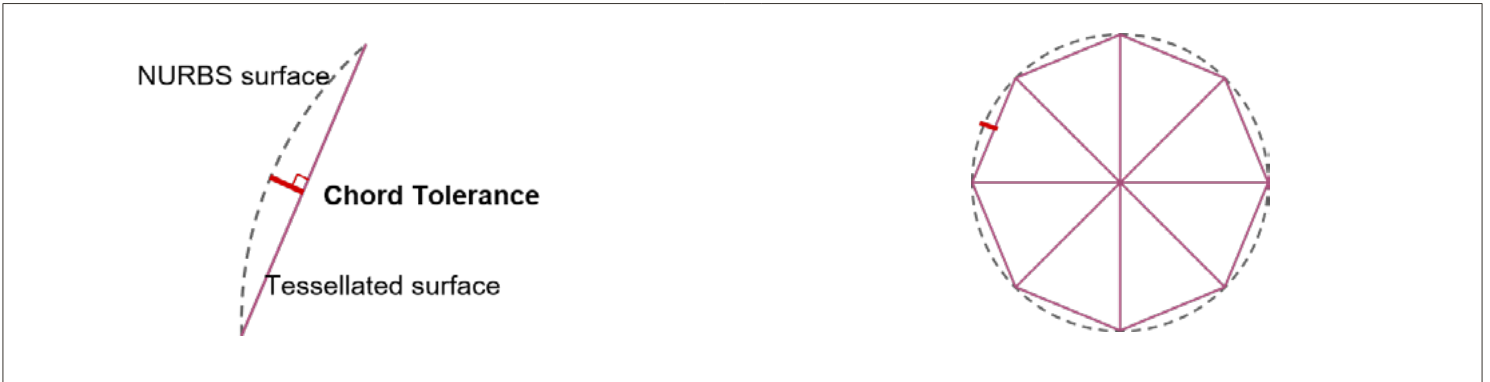


Figure 9: Chord Tolerance and relationship to geometry representation

The effect of increased chord tolerance can be visible on the vertex sampling along curved surfaces. The surface smoothness is reduced and the triangles constituting the surface become visible.

The following figure illustrates the effect of Chord Tolerance, with all other tessellation parameters held constant.

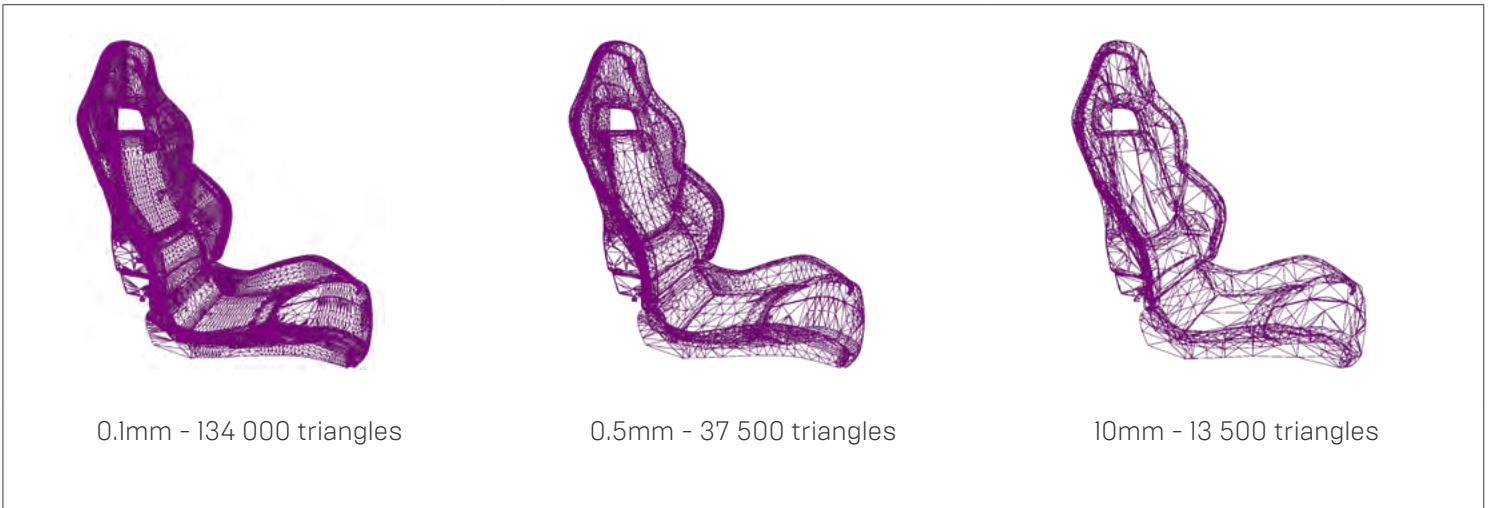


Figure 10: Surface of a car seat, shown with the resulting tessellation from various Chord Tolerance settings.

Max Edge Length

The Max Edge Length parameter controls the maximum size of any edge of the triangles and thus is useful to limit the presence of big or elongated triangles in the model. In Datasmith, setting this value to 0 means this parameter is ignored.



Figure 11: Max Edge Length and relationship to geometry representation

Example with Chord Tolerance and Normal Tolerance both constant to emphasize the effect of Max Edge Length:

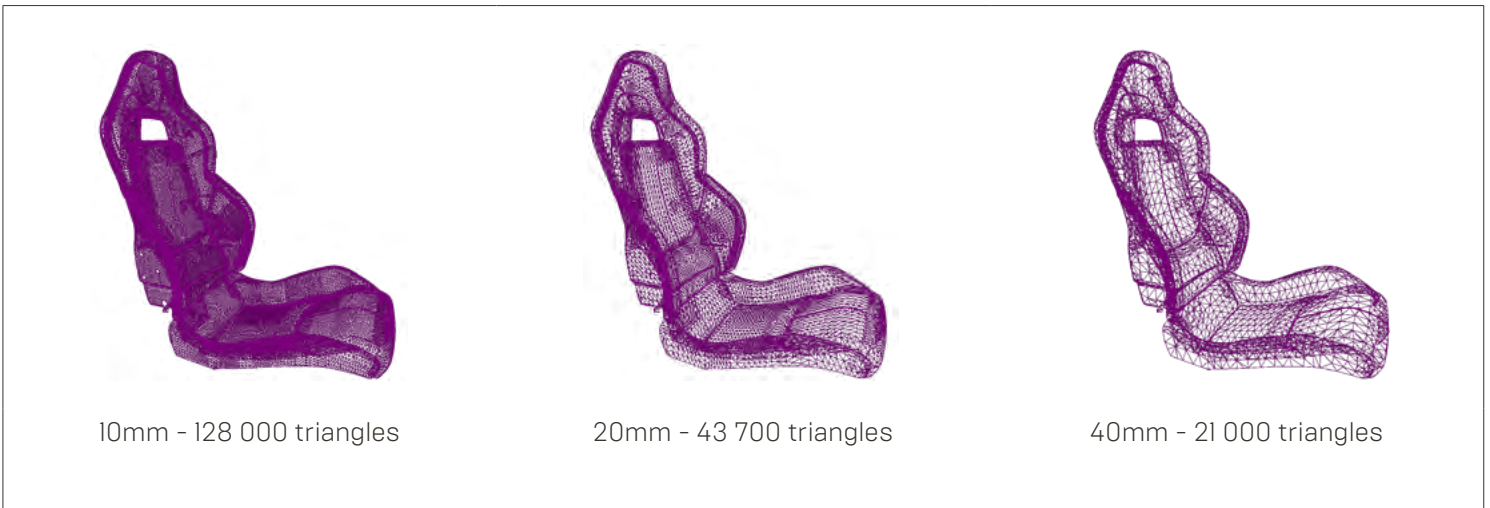


Figure 12: Resulting tessellation from various Max Edge Length settings.

Normal Tolerance

The Normal Tolerance parameters limits the angle between two adjacent triangles on the surface. The tessellation algorithm will cap these values to prevent undersampling and guarantee that the general shape of the surface is preserved.

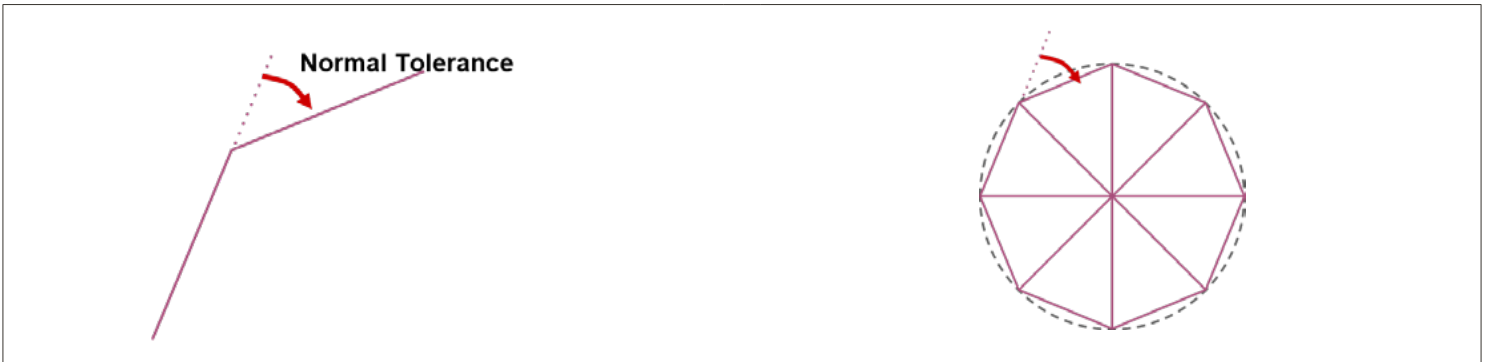


Figure 13: Normal Tolerance and relationship to geometry representation

In a similar way to Chord Tolerance, Normal Tolerance has an impact on the accepted deviation between the NURBS surface and the sampled surface. It differs from Chord Tolerance in that it guarantees a minimum curvature, with preservation of detail in areas of high curvature.

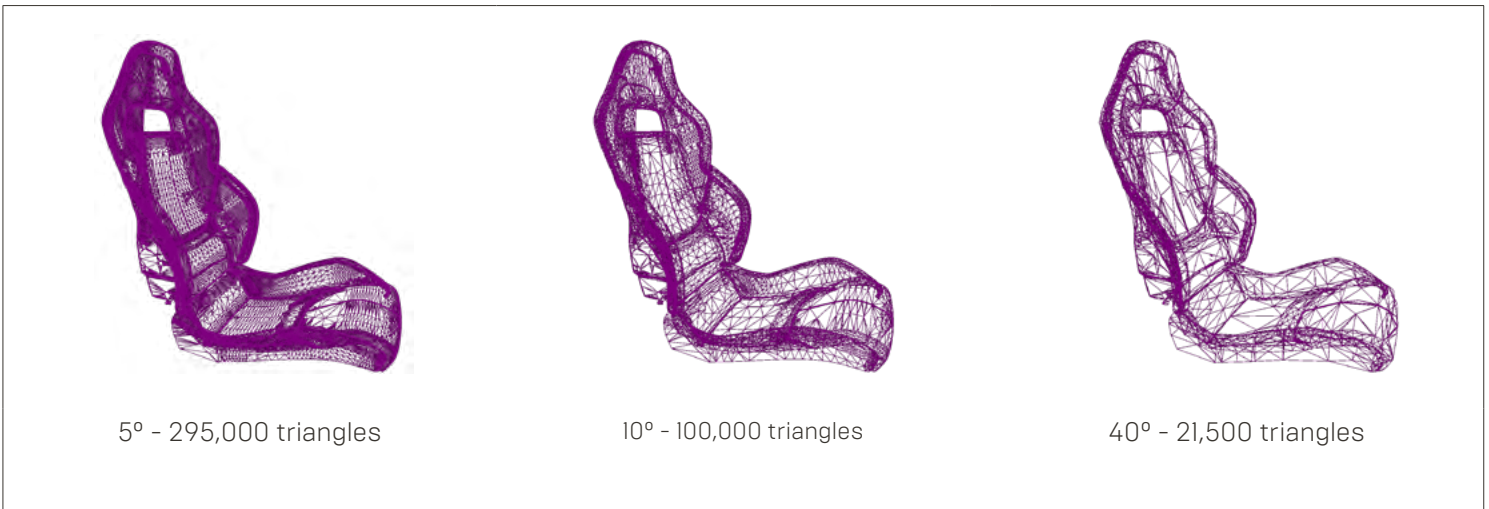


Figure 14: Resulting tessellation from various Normal Tolerance settings.

Different Needs for Different Parts

Chord Length, Max Edge Length, and Normal Tolerance apply uniformly to any geometry being processed, based on the complexity and form of the part. The ideal settings for each may change based the parts you are processing.

In some situations, it might be optimal to split an imported model into several CAD files to which different tessellation levels can be applied, and then re-merge the imported mesh inside the game engine editor.

The goal of the initial import is to have meshes that hold enough fine detail when observed at close distance. An extra simplification step is described later to optimize when the part is not close to the camera.

Degenerate Triangles

In polygon modeling, each triangle is defined internally by three points, each at a different location in three-dimensional space. If two or more of the triangle points are in the same location, the result is a degenerate triangle. If two of the points are at the same location, the triangle has only one edge. If all three of the points are in the same location, the triangle consists of a single point.

These degenerate triangle forms can cause problems in the rendering process, both computationally and in the visual result. When tessellating, it is important to use settings that do not result in degenerate forms.

Optimizing for Speed

The rendering process of a 3D scene consists of several steps where geometry and lights are converted to a rendered image. It is a pipeline where each step has a cost measured in time. The goal is to reach an equilibrium between render quality and interactivity or playback speed. This speed is measured in rendered frames per second [fps].

The ideal preparation would reduce the number of triangles to the targeted budget for the platform while preserving visual quality. This can be done on the initial import of the model by tuning the tessellation parameters.

Once the model is imported into the editor, there are tools that should be used to limit the number of triangles used by a model when observed from a distance.

LODs

Using the Level Of Detail (LOD) feature instructs Unreal Engine to use a version of a model with fewer polygons when the model is a certain distance from the camera, far enough away that the missing detail is not detectable. When LODs are available, the engine will replace geometry by their low-poly counterpart based on the model's relative size on the screen. LODs are primarily used to reduce triangle count on objects for faster rendering, but they can also reduce draw calls and shader complexity by using meshes with fewer materials or simpler shaders for the lowest LODs. Documentation on LODs can be found in the help topic [Creating and Using LODs](#).

To benefit from this feature, an LOD configuration should be created for the static meshes used in the scene. Two options are available:

- Create your own LODs by generating low-poly versions of your model, then importing and linking them to your static meshes in Unreal Engine. Another approach is to define for each of the LODs the percentage of triangles from LOD 0 and the size on screen that the engine will use as threshold to swap between LODs.
- Use one of the defined preset LOD Groups in the editor. After you select a preset, the editor will generate the corresponding number of LODs and size on screen.

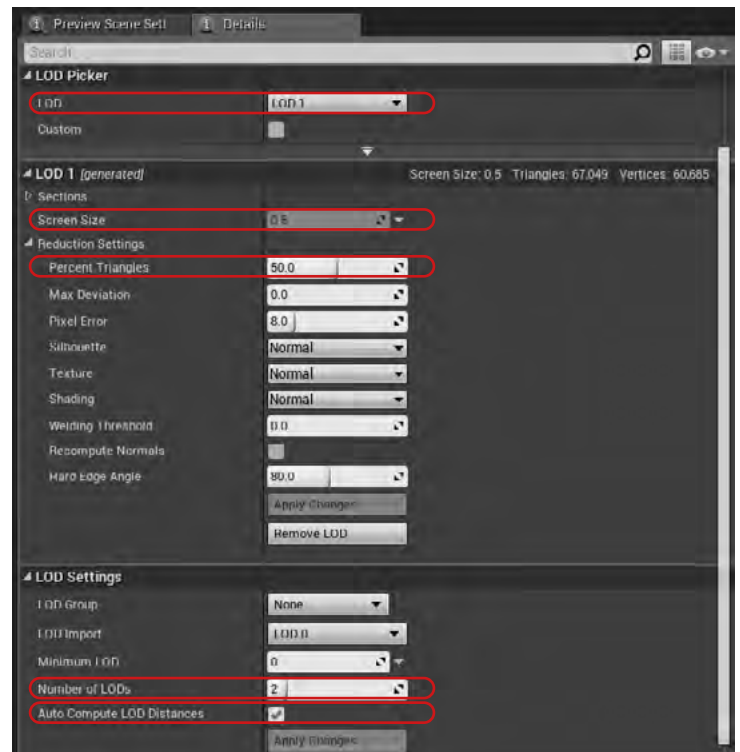
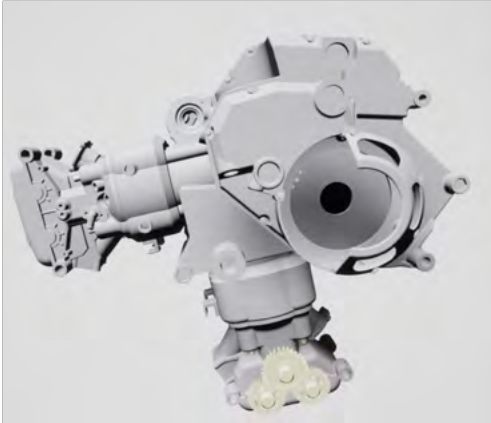
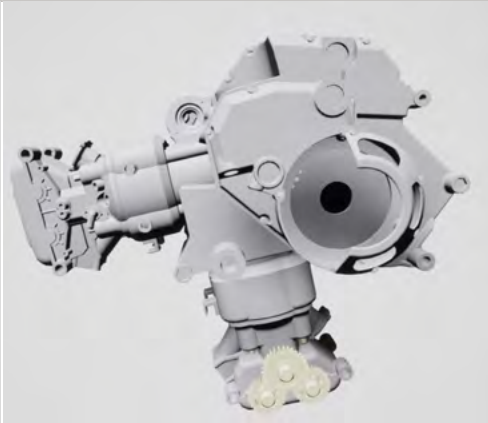


Figure 15: LOD dialog

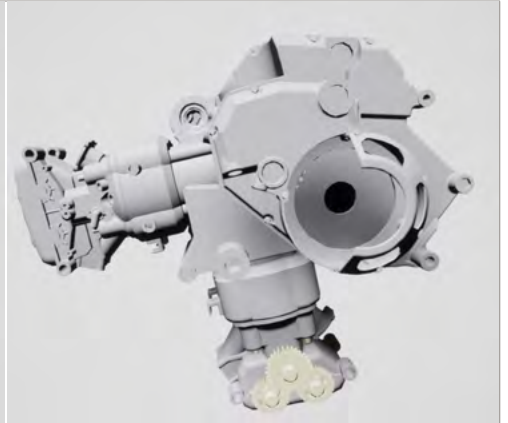
The following table illustrates the High Details configuration on the engine model. The full detail part, LOD 0, will be displayed as soon as the part takes up more than 23% of the screen space.

**LOD 0**

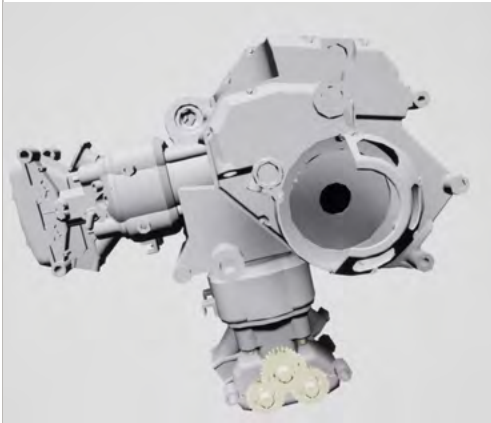
Screen Size: below 1.0
Triangles: 389 000 (N)

**LOD 1**

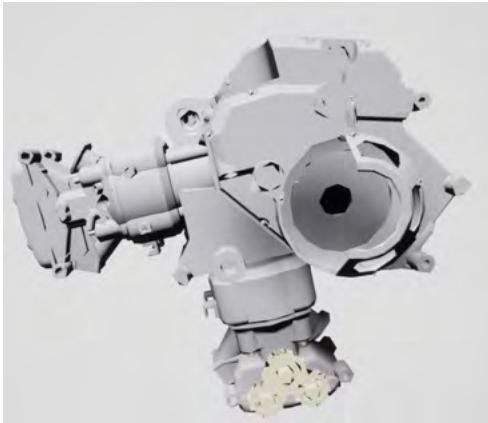
Screen Size: below 0.23
Triangles: 195 000 (N/2)

**LOD 2**

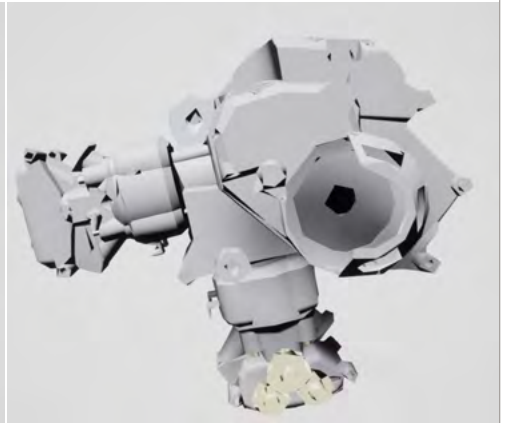
Screen Size: below 0.10
Triangles: 97 000 (N/4)

**LOD 3**

Screen Size: below 0.05
Triangles: 48 000 (N/8)

**LOD 4**

Screen Size: below 0.033
Triangles: 24 000 (N/16)

**LOD 5**

Screen Size: below 0.01
Triangles: 12 000 (N/32)

Table 1: LOD configuration

The following image illustrates the result of LOD usage in a scene, with LOD 0 in white and LOD 5 in pink.

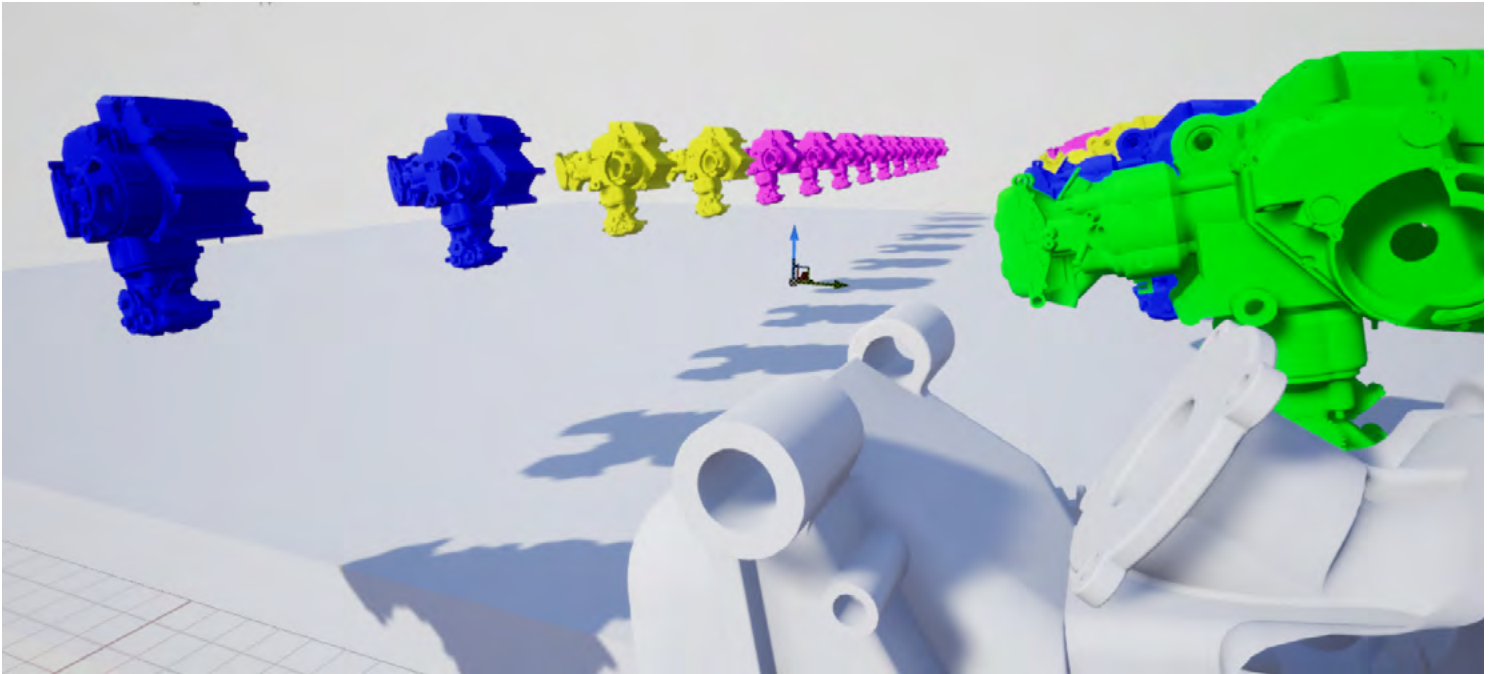


Figure 16: LOD representation as different colors

Defeaturing

The defeaturing command removes protrusions, through-holes and blind holes. Defeaturing can drastically reduce the polygon count in mechanical models.

The function specifies which kind of feature should be removed as well as the dimension threshold for them to be processed.



Figure 17: Defeaturing options

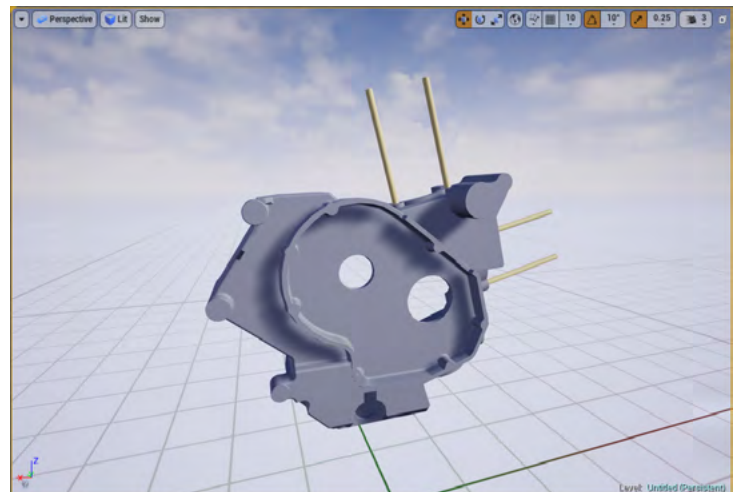
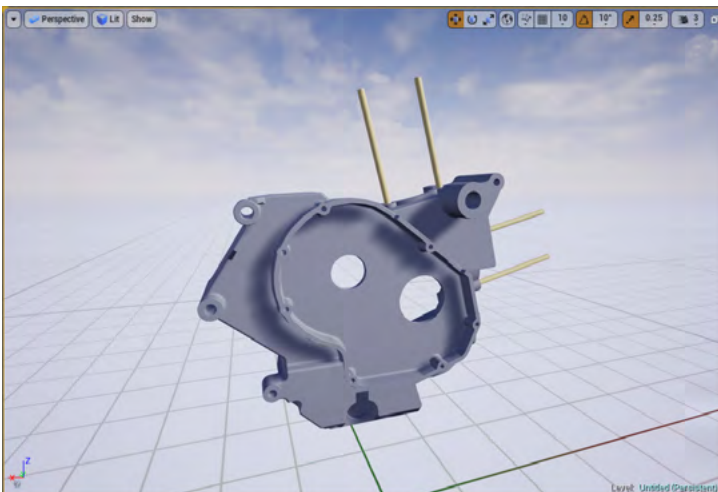


Figure 18: Smallest through-holes have been filled while the ones with larger diameters have been kept, based on the threshold.

Reducing Draw Calls

As the engine updates the scene in real time, it performs draw calls for each frame. A draw call is an internal request for all the information needed to draw the scene for the current view.

For each draw call, there is a cost in time. By reducing draw calls, we can increase the potential real-time playback speed.

Merging and Instancing

For each draw command there is a cost for the command itself—a real-time engine will take less time to draw one object of 100,000 triangles than 100 objects of 1,000 triangles, even though the total number of triangles is the same. This means that merging objects will reduce draw calls and thus improve performance.

When possible, the operator should merge objects to reduce draw calls. If the scene relies on replicated geometry, instancing can also be used to reduce the draw cost. With instancing, the actual geometry for a single object is stored, while the remaining versions of the object are stored as references to the original object. This method of storing multiple versions of an object saves time in redrawing the scene.

The following table gives an example of merging of elements on a partial car exterior. An exploded visualization is used to highlight the independent parts.




<p>No merging 90 objects</p> <p>Benefit: Only the triangles of the visible surfaces are drawn.</p> <p>Drawback: If all the exterior is visible, 90 draw calls.</p> <p>Worst triangle count for a single object: 75,000 (wheel arch)</p>	
<p>Partial merging 8 objects</p> <p>Balanced solution between draw calls and geometry, with 8 draw calls at worst.</p> <p>Hood, doors, fuel trap were kept separated for animation purposes.</p> <p>Worst triangle count for a single object: 313,000 (back exterior)</p>	
<p>All merging 1 object</p> <p>Benefit: Only one draw call</p> <p>Drawback: Even if only a part of the exterior is visible, all will be drawn.</p> <p>Worst triangle count for a single object: 1,000,000 (full exterior)</p>	

Table 2: Comparison of benefits and drawbacks of various levels of merging

After meshes are merged, new LODs can be generated for the merged entity. For more information, see the Actor Merging help topic in the Unreal Engine documentation.

Proxy LOD

Proxy LOD is an experimental option in Unreal Engine. This system will generate a single mesh with a single texture to replace a group of selected meshes. This reduces the number of triangles and reduces draw calls by merging two or more materials into one. For more information, see the [Proxy Geometry Tool](#) help topic in the Unreal Engine documentation.

LOD Control

In preparing data, there may be cases where you want to keep a high-resolution LOD available in the scene, but do not need that degree of detail for the current application. In LOD configuration, an option is available to control the minimum LOD (highest level of detail) that can be used by the application.

LOD configuration also provides a way to define different screen-size values depending on the platform running the application. By doing so, all the assets are prepared and stored in one project but only the relevant one will be used in the final application. For more information, see the [Per-Platform LOD Screen Size](#) help topic.

Jacketing

Another option to limit the number of draw calls is to use the jacketing function, which detects objects that are potentially occluded so it can hide or remove them. Jacketing can be used in complex assemblies where many internal parts have a low probability of being visible.



Figure 20: Initial model had 447 objects, and 223 occluded objects were removed.

The method specifies a threshold below which gaps in the geometry are not considered for visibility checks. This allows for assemblies not fully closed to be processed, and occluded objects removed.



Figure 19: On the left is the original chassis (3,875,000 triangles, 52 meshes, and 19 materials). On the right is its proxy version (130,000 triangles, 1 mesh, and 1 material).

Data-Prep Strategy and Best Practices

The ability to identify the specific requirements of one's application is as essential as knowledge of data preparation tools. The two key elements to identify are the targeted platform and the targeted application. The targeted platform informs on the known limitations and provides the technical boundaries of the project. The targeted application informs on how to prioritize the limited resources for the desired functionality.

Platform

The type of platform targeted will impact the optimization as well as the architecture of the application. This step is essential to identify the budget in term of quality and number of assets in the scene.

Computer Desktop or Server

Among all possible platforms, a desktop computer or server is the real-time configuration that has the most computational power and storage space, and thus the fewest number of restrictions. On a desktop, the application will be experienced through a standard display screen and inputs (keyboard, mouse, console pad). A desktop setup being the configuration that is least constrained by hardware, the graphical quality can be pushed to the highest level. It is also the configuration that will be able to support the highest level of detail in the geometry.

VR Headset

Virtual-reality headsets are typically powered by computer desktops, so require similar considerations. However, the double render per viewpoint (one per eye) as well as high frame rate (to preserve user comfort) put an extra strain on rendering. The main benefit of VR over desktop and AR is full immersion of the user in the 3D environment, giving a better perception of volumes and spaces. VR is also a more "natural" way to work in the application, as users rely on direct body movement and hand manipulation to navigate and interact in the virtual scene. This is why VR is used when the experience is key, such as with training, exploration of buildings, and reviews.

HMD Device	Target Frame Rate
DK1	60 FPS
DK2	75 FPS
Rift Retail	90 FPS
Vive	90 FPS
Gear VR	60 FPS
PSVR	Variable up to 120 FPS

Table 3: Frame rates for various VR HMD (head-mounted display) devices

VR CAVE/Powerwall

VR on a [powerwall](#) or [CAVE](#) (cave automatic virtual environment) is very similar to a VR experience on a headset, as it uses dual rendering with high refresh rates on powerful computers. The main difference is the need for a multi-display output, as the application renders several different points of view and pushes them to multiple screens or projectors. Support for multi-display output has been added to Unreal Engine as of v4.20.

AR

AR (augmented reality) is usually performed on lightweight and small devices with limited computational power and high autonomy such as mobile, tablet, or headset. AR requires position tracking to align virtual objects on the real environment, which requires resources for image processing that take away from computational resources. This is why an AR configuration requires the most care in preparation, and the strongest optimization.

Application

The needs for different audiences vary; the real-time experience desired for customers differs from the one for engineers. Determining the application and audience helps in prioritizing functionality vs. optimization, and thus in the approach to preparing data.

Application	Requirements
Design review / Inspection Audience: Designers/Engineers	Geometry detail: High level and visibility of hidden geometry Number of parts: High. Separated parts for individual inspection or merged at low level. Material complexity: Medium to high (design) Data: Engineering and variants Scene size: Medium to high (product) Interactivity: Medium; manipulation, animation, material switching
Processes simulation / Factory Audience: Engineers/Production managers/Workers	Geometry detail: Medium level Number of parts: Very high Material complexity: Simple to Medium Data: Engineering and sequencing Scene size: Very high (factory, city) Interactivity: High; animation and simulation (physics)
Marketing Audience: Retailers/Clients	Geometry detail: High level Number of parts: Medium Material complexity: High Data: Marketing, sales Scene size: Medium to high (product) Interactivity: Medium; manipulation, animation, material switching

Table 4: Applications and their respective requirements



Overall Methodology

1. Based on your application needs, identify the expected minimum level of detail for each geometry item.
2. Import a small set of representative parts and identify tessellation parameters for your product so that each static mesh matches its expected level of detail. Store those tessellation parameters in presets. Avoid tessellation parameters that create degenerate and elongated triangles.
3. Create a recipe: associate product parts with one of your tessellation presets.
4. Import your products as static meshes. For each part, you should have a level of detail that satisfies the requirements of your application.
5. Create LODs for your static meshes. This step is highly recommended for most applications, especially if you are targeting several platforms with different performance capabilities.
6. Merge objects: find a balance between meaningful merges with respect to the application purpose and spatial proximity of objects. It is not advisable to merge objects that are not spatially close.
7. Scene content:
 - a) Repetitive mesh: use static mesh instances
 - b) Large static environment: use hierarchical LODs
 - c) Large objects: split geometry before import



Automation in Practice: McLaren Design Team

When supporting McLaren's design visualization team with Unreal Engine solutions, part of our goal was to integrate with existing design workflows and minimize the resources, software seats, and time required to get design information into different types of real-time visualization scenarios, while maintaining image quality. Our test project involved automating various parts of the ingestion and conversion process to focus design manpower on added-value work.

Tailoring a Solution

The following describes the considerations when working with the McLaren project in Unreal Engine.

Available 3D Data

In this case, we are working with early design data, which (as is typical for most manufacturers at this stage) is incomplete in some areas. This data was not going out to marketing; its purpose was to empower visual communication across the team with a minimum of overhead. In this case, IGES or CATPART files will be posted to a centralized repository or database when versions are saved, along with their metadata.

Available Metadata

Similar to the 3D data, the metadata at this stage is early in its level of sophistication and usefulness. Data such as creation date, configuration, version, and part group does provide some useful information for basic organization.

Quality Needs

The data needs to be of a relatively high resolution to preserve the design vision when moving from NURBS to polygons. It should be at a quality level suitable for communicating the design vision without distractions, but it is not going to be used for marketing without additional cleanup.

Quick Iteration Needs

The design team is modifying their designs on a daily basis, so they need a solution that allows them to quickly view these changes without requiring data preparation manpower.

End Use

The team wishes to use this data for image creation, virtual reality, and augmented reality, so multiple tessellation LODs are advisable. However, speed is more important here than meticulous preparation.

Automation with Python

Data preparation and conversion can be quite time-consuming for individuals, with the individual steps being repetitive and processor-intensive. One goal in creating a data preparation system is to automate as many tasks as possible, and keep the processing from causing downtime on an individual's workstation.

While our stock software tries to automate as many common steps as possible, there are always items unique to each business that must be defined to solve particular problems. In general, if a process is logical enough that it can be outlined step-by-step in a department manual, we (and the designers and managers) would rather have code take care of this automatically.

This is where the integration of Python in Unreal Engine becomes so valuable. Python is the standard in 3D/VFX pipelines. Its ease of use and long history in 3D pipelines (as well as machine learning) makes it an ideal choice for finding or creating quick low-scope customizations for repetitive tasks. In Unreal Engine's editor, almost all the Unreal Engine tools and functions are accessible through Python, making quick automations possible. Simple scripts can be linked together to powerful effect.

For McLaren, we automated the following tasks to fit their processes and metadata, and to simplify the workflow:

- Parse design repository and a) create a new project if it is the first import, or b) import new versions inside the existing Unreal Engine project.
- Tessellate the geometry and generate LODs.
- Group generated assets in Unreal Engine in a way that respects the folder hierarchy of McLaren.
- Duplicate and mirror geometry: many parts are designed on half the car and thus need to be mirrored for the visualization inside the real-time application.
- Populate the Unreal Engine scene with actors of the imported geometry.
- Assign specific materials to the scene actors.

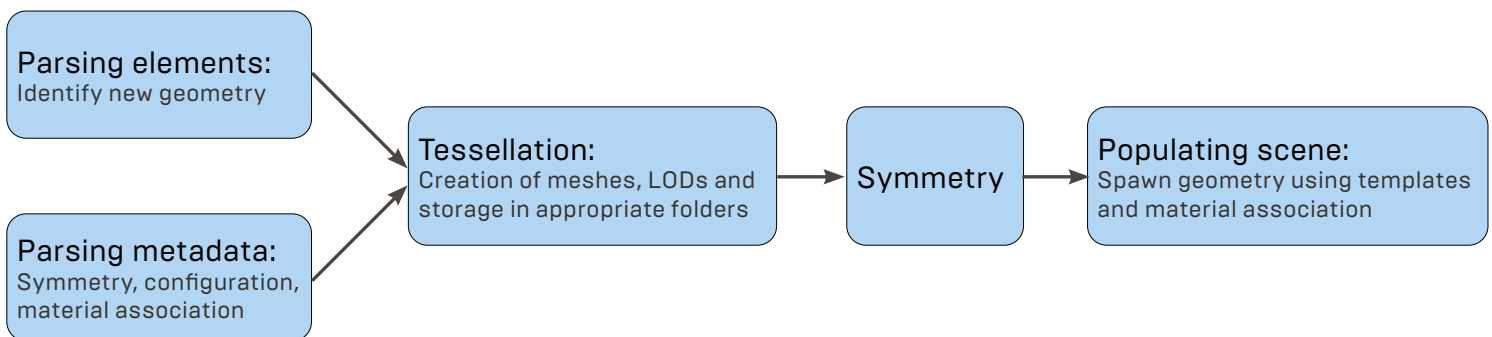


Figure 21: View of the key steps in the automated import of geometry and metadata using Python scripting.

The Python scripting capabilities are accessible through a [Python plugin in Unreal Editor](#). Once activated for a project, Python scripts can be executed to generate and modify assets for the project.

UE4 provides a set of high-level libraries specially designed to facilitate scripting. The EditorAssetLibrary, EditorLevelLibrary, and EditorStaticMeshLibrary perform standard operations on assets, levels, and static meshes in a convenient way by wrapping low-level UE4 editor functionality. The Datsmith import functions are also accessible; for example, CAD import is available through the DatsmithCADSceneElement class.

Sample Python Scripts

The purpose of this section is to show and describe codes snippets that have been used for the McLaren project, and which can be easily adapted for other design visualization projects.

Setting Up a Level and Importing CAD

The code snippet below creates a new level from the default level template (the standard scene with ground, sky, and lighting set-up):

```
from unreal import EditorLevelLibrary

# create new level from template
new_level = EditorLevelLibrary.new_level_from_template("/Game/NewLevel", "/Engine/Maps/Templates/Template_Default")

# save new level
EditorLevelLibrary.save_all_dirty_levels()
```

A CAD file can be imported with Datasmith using Python commands. In addition to the settings already exposed in the Editor's interface, the Python commands allow more flexibility with additional advanced settings.

For example, the following code imports a CAD file into the current level, setting the tessellation parameters and saving all imported assets:

```
from unreal import EditorAssetLibrary, DatasmithCADSceneElement, DatasmithImportScene

# content folder where the imported assets will be saved
content_folder = "/Game/Import"

# CAD file to import, must be a format supported by datasmith
cad_file_path = "D:/Data/Engine.SLDASM"

# init datasmith CAD scene import from a CAD file and a target content directory
datasmith_scene = DatasmithCADSceneElement.construct_datasmith_scene_from_file(cad_file_path, content_folder)
import_opt = datasmith_scene.get_import_options()
import_opt.base_options.scene_handling = DatasmithImportScene.CURRENT_LEVEL
import_opt.tessellation_options.chord_tolerance = 0.3
import_opt.tessellation_options.max_edge_length = 0
import_opt.tessellation_options.normal_tolerance = 30

# process the scene for import
datasmith_scene.process_scene()

# import the scene into the current level
result = datasmith_scene.import_scene()

# save imported static meshes and materials assets
for static_mesh in result.imported_meshes:
    EditorAssetLibrary.save_loaded_asset(static_mesh)
    for i in range(static_mesh.get_num_sections(0)):
        material_interface = static_mesh.get_material(i)
        EditorAssetLibrary.save_loaded_asset(material_interface)
```

Asset Organization

Asset organization and naming is key for proper automation. The following sample creates a new folder in the content folder and moves all static mesh assets into the folder. Note that renaming assets will prevent the re-import of the CAD file; this feature requires specific naming of the files to execute.

```
from os import path
from unreal import EditorAssetLibrary

# create a new directory in the content folder
EditorAssetLibrary.make_directory("/Game/NewDirectory")

for static_mesh in result.imported_meshes:
    # move an existing asset to the newly created directory
    new_path_name = "/Game/NewDirectory/" + path.basename(static_mesh.get_outermost().get_name())
    EditorAssetLibrary.rename_loaded_asset(static_mesh, new_path_name)
```

After importing multiple files using the same data, some assets might be duplicates in several places. Unreal Engine has a function called 'consolidation', which resolves this issue by keeping only one referenced version of an asset.

The following code sample looks for material assets in all the content directory subfolders and consolidates all materials having the same name. Note that the consolidation criteria operates on asset names, so assets having different material properties but the same name end up being consolidated.

```
from unreal import EditorAssetLibrary, MaterialInstanceConstant

# retrieves all material instances assets
all_asset_names = EditorAssetLibrary.list_assets("/Game/", True, False)

# loads all assets of the specified class from the given directories
material_assets = []
for asset_name in all_asset_names:
    loaded_asset = EditorAssetLibrary.load_asset(asset_name)
    if loaded_asset.__class__ == MaterialInstanceConstant:
        material_assets.append(loaded_asset)

# regroup identicals assets
asset_consolidation = {}
for i in range(0, len(material_assets)):
    name = material_assets[i].get_name()
    if not name in asset_consolidation:
        asset_consolidation[name] = []
    asset_consolidation[name].append(i)

# consolidate references of identical assets
for asset_name, assets_ids in asset_consolidation.items():
    if len(assets_ids) < 2:
        continue
EditorAssetLibrary.consolidate_assets(material_assets[assets_ids[0]], [material_assets[i] for i in assets_ids[1:]])
```

Scene Organization

Populating or editing scene content is also a good candidate for automation. The next example is a simple script that spawns a new static mesh actor from an existing static mesh asset at the origin of the world. It also sets the mobility of the newly created actor to 'Movable', which is required to be able to manipulate the object at runtime.

```
from unreal import EditorLevelLibrary, StaticMeshActor, ComponentMobility, Rotator, Vector

# spawn a new static mesh actor at origin
new_actor = EditorLevelLibrary.spawn_actor_from_class(StaticMeshActor, Vector(0, 0, 0), Rotator(0, 0, 0))

# load a static mesh asset
static_mesh = EditorAssetLibrary.load_asset("/Game/SM_Test")

# set the static mesh component of the new actor
new_actor.static_mesh_component.set_static_mesh(static_mesh)

# set the new actor label (text displayed for the actor in the world outliner)
new_actor.set_actor_label("Actor spawned from Python")
# set actor's mobility to Movable
new_actor.root_component.set_editor_property("mobility", ComponentMobility.MOVABLE)
```

Scene Optimization

As we have seen in the previous chapter, the ability to merge geometry is a great technique to reduce the number of draw calls and improve performance. In the following example, we automatically merge all static mesh actors in the current level with names that start with "Bolt".

```
from unreal import EditorLevelLibrary, EditorScriptingMergeStaticMeshActorsOptions
from unreal import MeshLODSelectionType, StaticMeshActor

# retrieves all static mesh actors in the level which names start with "Bolt"
actors = EditorLevelLibrary.get_all_level_actors()
actors_to_merge = [a for a in actors if a.__class__ == StaticMeshActor and a.get_actor_label().startswith("Bolt")]

# set the merge options
merge_options = EditorScriptingMergeStaticMeshActorsOptions()
merge_options.base_package_name = "/Game/SM_MergedActor"
merge_options.destroy_source_actors = False
merge_options.new_actor_label = "Merged Actor"
merge_options.spawn_merged_actor = True
merge_options.mesh_merging_settings.bake_vertex_data_to_mesh = False
merge_options.mesh_merging_settings.computed_light_map_resolution = False
merge_options.mesh_merging_settings.generate_light_map_uv = False
merge_options.mesh_merging_settings.lod_selection_type = MeshLODSelectionType.ALL_LO_DS
merge_options.mesh_merging_settings.merge_physics_data = True
merge_options.mesh_merging_settings.pivot_point_at_zero = True

# merge meshes actors and retrieve spawned actor
merged_actor = EditorLevelLibrary.merge_static_mesh_actors(actors_to_merge, merge_options)
```

It is also possible to generate levels of detail for the static meshes using Python. The next script generates two levels of detail for each static mesh used in the current level. The first LOD uses 100% of the triangle count, while the second uses 25% and is switched on when the screen size of the object falls below 60% of the total screen size.

```
from unreal import EditorLevelLibrary, EditorStaticMeshLibrary
from unreal import StaticMeshActor, EditorScriptingMeshReductionOptions, EditorScriptingMeshReductionSettings

# retrieves static meshes actors in the levels
actors = EditorLevelLibrary.get_all_level_actors()
static_meshes = { a.static_mesh_component.static_mesh for a in actors if a.__class__ == StaticMeshActor }

# generates LODs for each static mesh
for static_mesh in static_meshes:
    # reduction settings for each level of detail
    settings = [EditorScriptingMeshReductionSettings(1.0, 1.0), EditorScriptingMeshReductionSettings(0.25, 0.6)]
    # option for the mesh: don't auto compute the screen sizes and use previous reduction settings
    options = EditorScriptingMeshReductionOptions(False, settings)
    EditorStaticMeshLibrary.set_lods(static_mesh, options)
```

Set Collision Parameter

Collision shapes are required by UE4 to perform collision detection, physics simulation, or ray casting. Collision data is often required for interactive applications. The following example demonstrates how to create a simple convex envelope collision shape for all static meshes in the current level.

```
from unreal import EditorLevelLibrary, EditorStaticMeshLibrary, StaticMeshActor, ScriptingCollisionShapeType

# retrieves static meshes actors in the levels
actors = EditorLevelLibrary.get_all_level_actors()
static_meshes = { a.static_mesh_component.static_mesh for a in actors if a.__class__ == StaticMeshActor }

# add collision shape to all static meshes
for static_mesh in static_meshes:
    # use a k-DOP convex with 26 sides
    EditorStaticMeshLibrary.add_simple_collisions(static_mesh, ScriptingCollisionShapeType.NDOP26)
```

McLaren Design Visualization Application

The McLaren project is divided into two main components: an import script that is used to populate the Unreal Engine project with CAD content, and an Unreal Engine project containing logic and functions necessary for the design visualization application. This separation is useful to isolate the specific tasks and needs of preparation from the functionality of the visualization application.

We have already seen data preparation and import in action. The following sections cover the functions and processes for creating the application itself in Unreal Engine.

Navigation

To ease the deployment and usage of product review applications with Unreal Engine, a template project called the Product Viewer is provided. This project is designed to provide a quick start to experimenting with VR, mobile, and desktop product visualization with a minimal number of clicks. All the necessary logic is already developed in Unreal Engine's Blueprint visual programming language.

The usage of the template is straightforward. Just import data, connect the imported data to be used by the application's logic, then experiment on the platform of choice.



Figure 22: Easy setup of the product viewer

The project includes standard navigation techniques such as Fly and Orbit for the desktop mode, as well as Teleportation for the VR mode. Then the user can use review features, for example by manipulating parts, with the mouse on the desktop or with the controllers in VR. The X-Ray function allows intuitive inspection of an assembly by rendering a part or a set of parts as transparent.

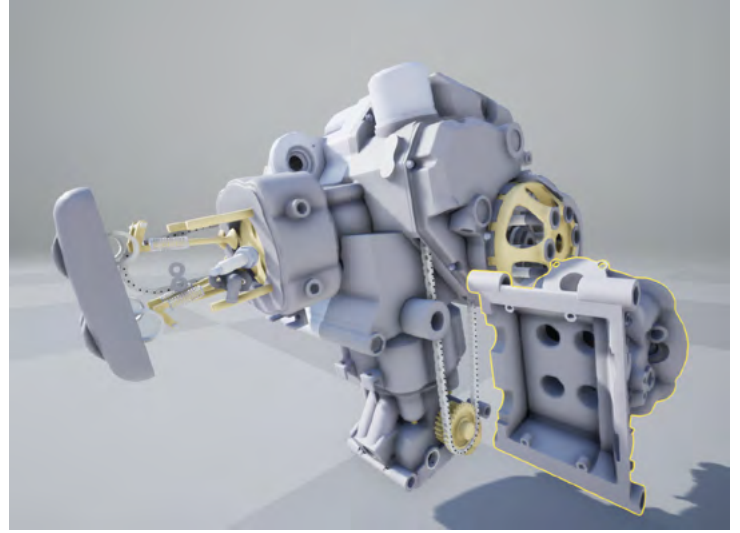
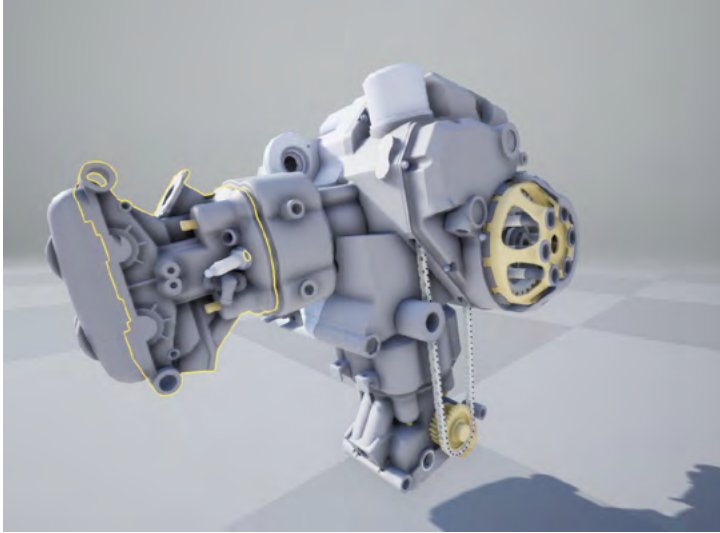


Figure 23: Manipulating a part with the mouse

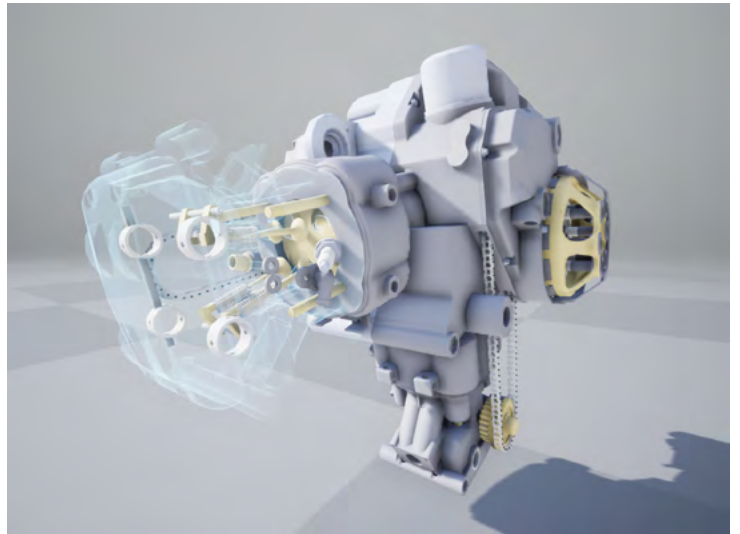
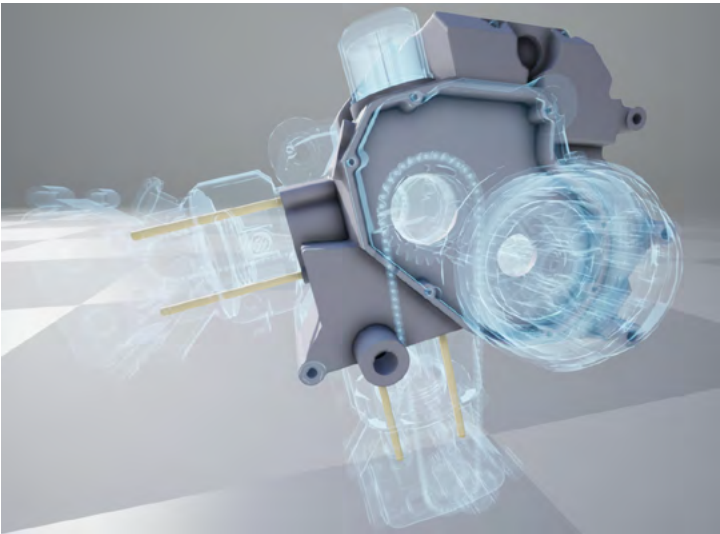


Figure 24: Inspecting assembly with the X-Ray function

The whole template has been designed with extensibility in mind so users can add job-specific functions to the project with ease.

The versatility of Unreal Engine's Product Viewer template makes it the perfect foundation for a custom design-visualization application. It is only necessary to add switching logic to create a configurator.

Using the Product Viewer template with one's own models is straightforward: any added actors containing geometry should be attached under a specific root in the default template scene. This is performed in the import script where:

- the active level is the default Product Viewer's level, and
- the spawned actors are stored in a tree that is attached under the Interactive_root node.



Custom Blueprints for Car Parts

To provide functionality for switching geometry, it is necessary to represent the data organization inside the application. First, the key properties that will be relevant for filtering the objects in the application have to be identified: version, configuration, creation date, etc. Second, this metadata has to be imported and stored. This can be achieved through the use of Unreal Engine DataTables or stored in custom Blueprints as shown in this example.

A car element can exist in different configurations such as sports, classic, or luxury, and for each configuration, several versions of the element are made as the design evolves.

The two main types of objects needed to handle the switching behavior are:

- “Part node” Blueprint, a static mesh actor representing a part and the properties associated with it.
- “Configuration node” Blueprint, a standard actor to hold the list of configurations and parts under it. When the user chooses that configuration, the parts under it will be used.

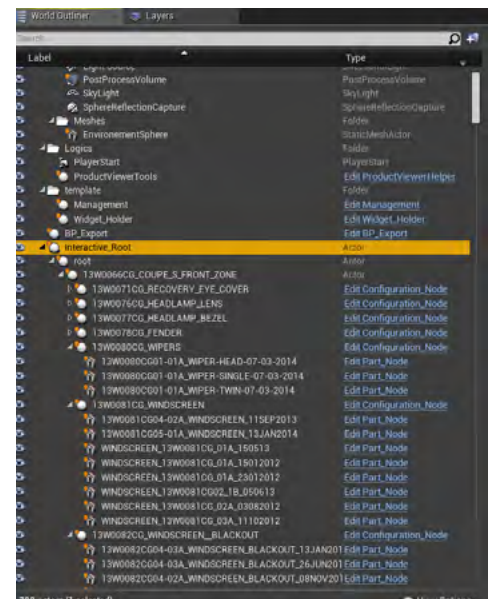


Figure 25: Interactive_root node in Outliner

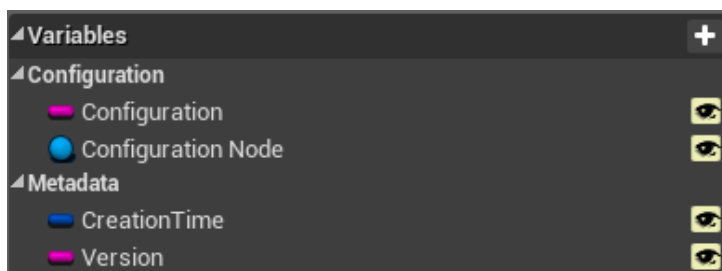


Figure 26: Part node

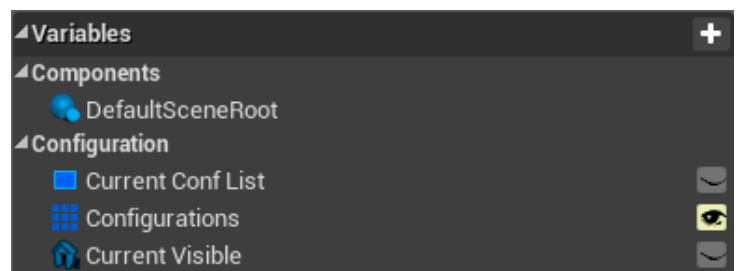


Figure 27: Configuration node

The import script is also used to spawn the specific types of actors depending on the McLaren data, and to fill in the metadata such as configuration name, creation time, and version.

The triggering of the part selection is performed in the Blueprint of a dedicated actor. On a key trigger, a hit test is performed to identify the part that is under the cursor. Once the part is identified, an event is sent to the associated configuration node, as it is the object holding the complete information regarding the different versions and configurations of the part.

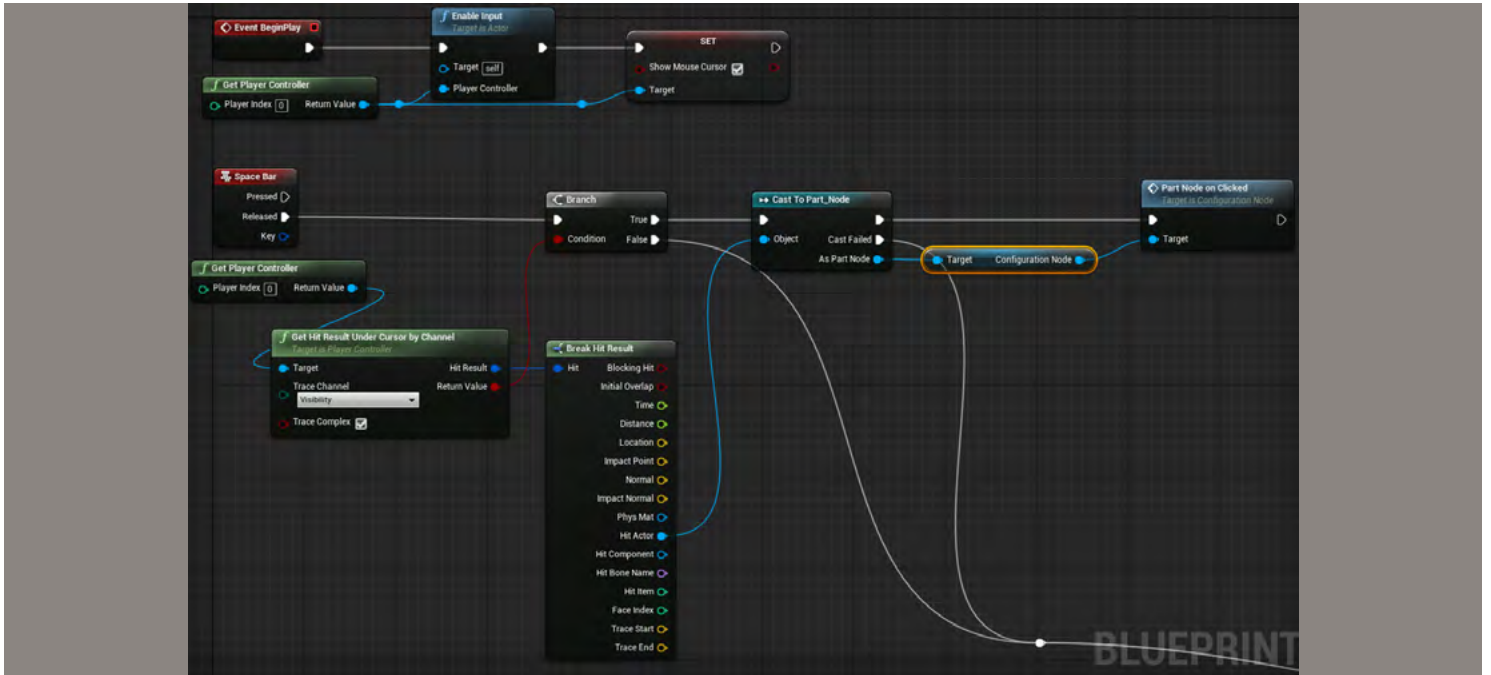


Figure 28: Blueprint graph for part configuration

In the event graph of the configuration node there are two main events:

- On begin: fill the array of configuration itself containing an array of parts. This is performed by scanning “part nodes” under the configuration node,
- On part being selected: make the UI visible and fill it with information relative to the current configuration node.

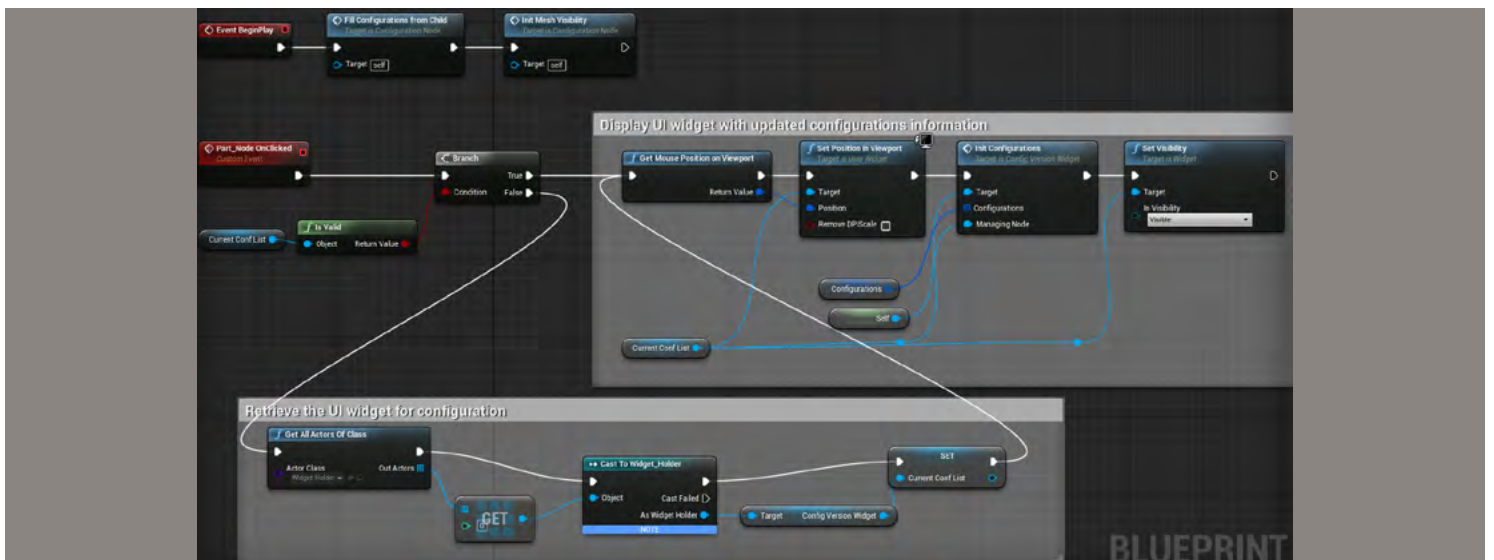


Figure 29: Blueprint with sections labeled



Graphical User Interface

The custom UI is separated into two specific assets: a widget to hold the configurations/versions lists, and a widget to represent a selectable cell in the list. When the user selects a part on the model, the UI widget is populated with the corresponding configurations and versions, made visible and displayed under the user's cursor.

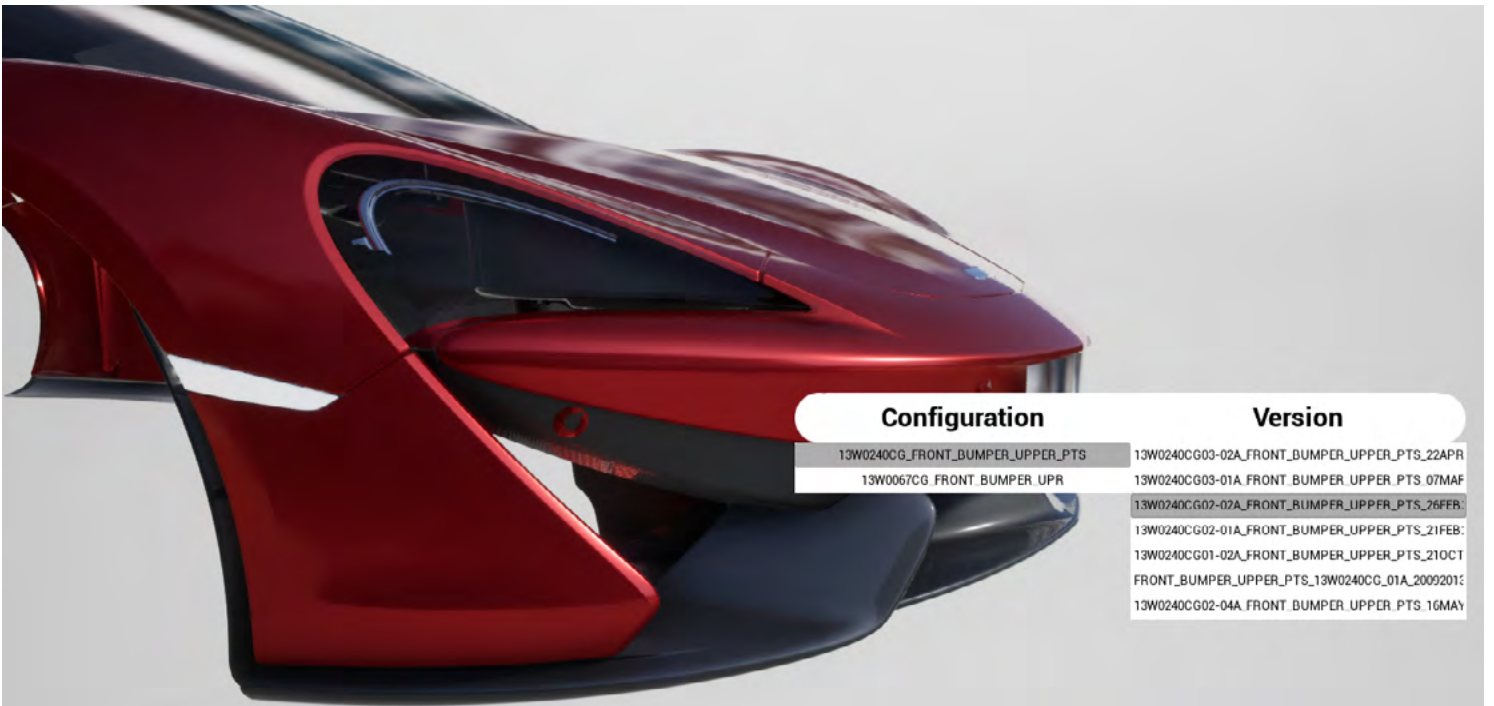


Figure 30: Custom UI

The logic is separated between the two widgets. The cell widget is used to trigger switching of configuration or versions, and the configuration widget is used to centralize the switch events triggered by browsing the list.

In the cell widget's event graph, a specific behavior is defined to be run when the cursor is hovering over the cell. Based on the type of cell, hovering changes the item's background color, or triggers a configuration change or part change.

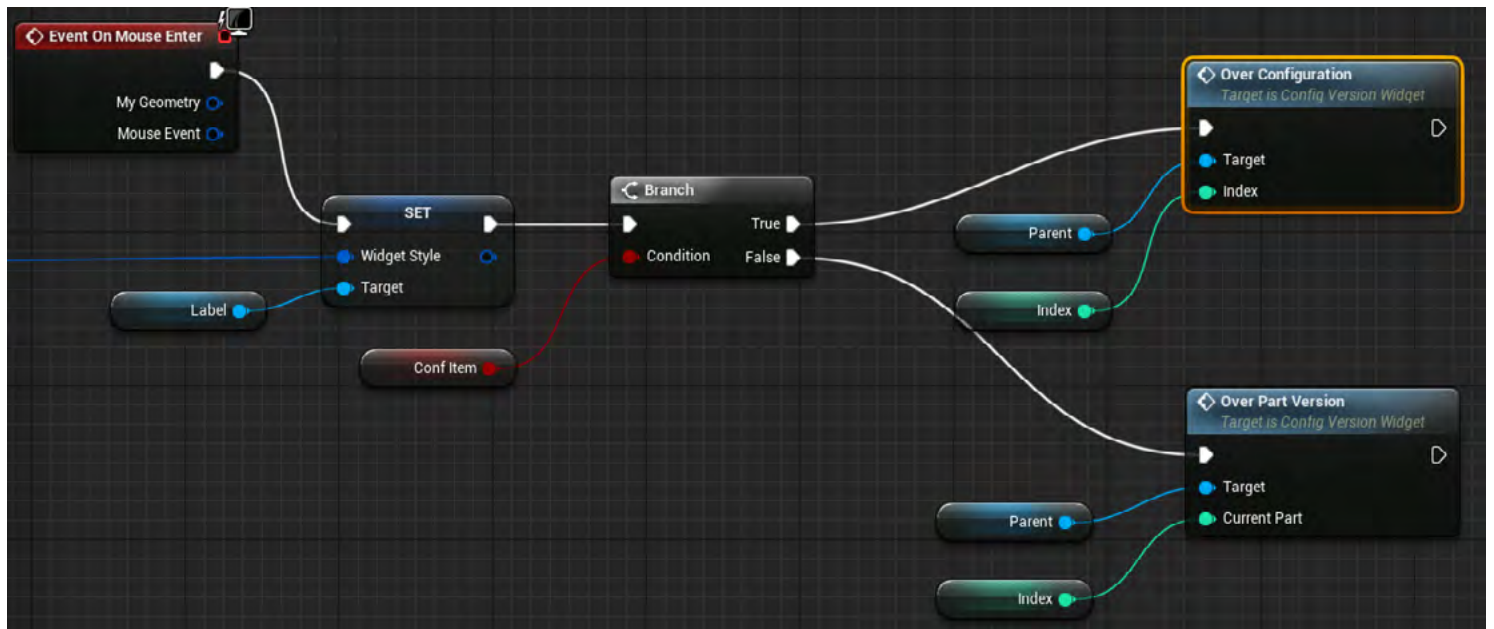


Figure 31: Cell widget event graph

In the configuration widget's event graph, the current configuration and version index are updated and the corresponding actor is made visible.

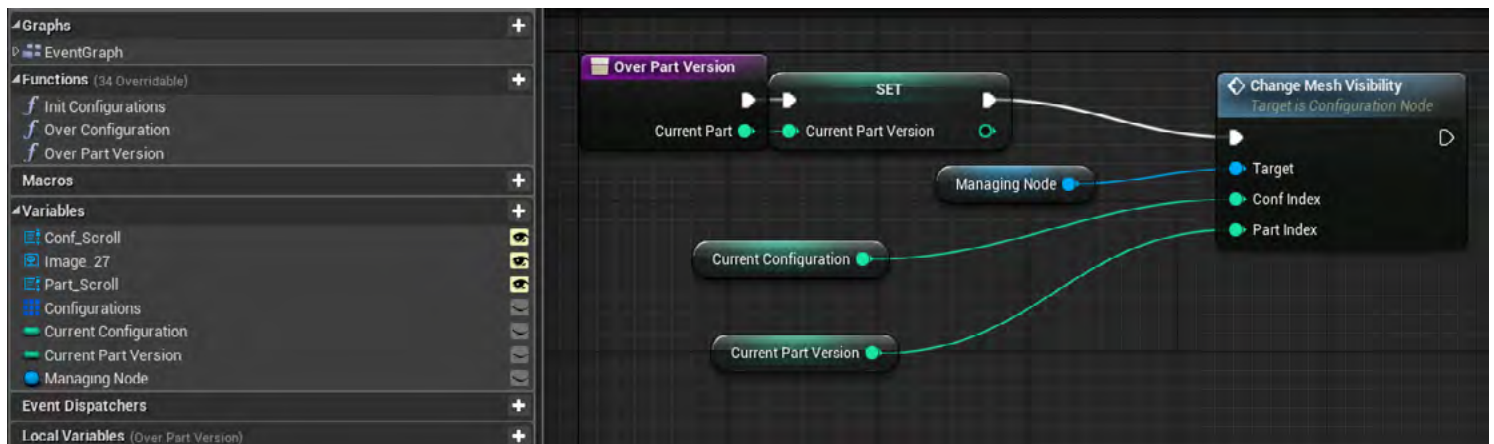


Figure 32: Configuration widget event graph

Unreal Engine as a Platform

Once your data is prepared and organized for use within Unreal Engine, you have many opportunities to create content and interactive experiences with it. Note that as Unreal Engine makes it relatively simple to migrate content and functionality from one project to another, many of these use cases can be combined into a suite of solutions to create an efficiency of scale across productions.

To see examples of how prepared data was used within Unreal Engine and inspire your own efforts, see the [Unreal Engine blog](#).

About this White Paper

Authors

Thomas Convard

Flavien Picon

Michael Wilken

Contributors

Sebastien Miglio

Original vehicle images and data supplied by McLaren Automotive