BOURNEMOUTH UNIVERSITY

MASTER THESIS

Virtual Garment Creation and Cloth Simulation: A Tool for Unreal Engine 5

Author:
Marisa LIEBNER

Supervisor:
Jon MACEY
Prof. Jian CHANG

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

Computer Animation and Visual Effects

August 22, 2025

BOURNEMOUTH UNIVERSITY

Abstract

Faculty of Media and Communication
National Centre for Computer Animation

Master of Science

Virtual Garment Creation and Cloth Simulation: A Tool for Unreal Engine 5

by Marisa LIEBNER

This paper presents a lightweight Unreal Engine 5 plugin that unifies 2D pattern drafting, automatic mesh generation and real-time cloth simulation via Chaos Cloth into a single, in-editor workflow. By leveraging UE's native geometry and UI modules, the system enables users to sketch flat patterns, generate triangulated meshes and preview drapes and collisions without external tools. While pattern authoring and mesh generation proved robust and performant, integrating seam constraints and cloth simulation entirely in C++ revealed challenges around module access. Nonetheless, the prototype delivered plausible drapes suitable for both real-time previews and cinematic pipelines, reducing iteration time and demonstrating the untapped potential of UE's native systems for digital garment production.

Contents

List of Figures

List of Abbreviations

CAD	**C**omputer **A**ided **D**esign
CDT	**C**onstrained **D**elaunay **T**riangulation
CGAL	**C**omputational **G**eometry **A**lgorithms **L**ibrary
DCC	**D**igital **C**ontent **C**reation
HCI	**H**uman **C**omputer **I**nteraction
MD	**M**arvelous **D**esigner
PBD	**P**osition **B**ased **D**ynamics
UBT	**U**nreal **B**uild **T**ool
UE	**U**nreal **E**ngine
UI	**U**ser **I**nterface
UMG	**U**nreal **M**otion **G**raphics
VID	**V**ertex **ID**
2D	**T**wo-**D**imensional (used here for 2D pattern space)
3D	**T**hree-**D**imensional (used here for 3D simulation/mesh space)

1 Introduction

Virtual garment creation plays an important role in 3D content production, used in industries ranging from game development to cinematic visual effects and digital fashion. In games, real-time cloth simulation is often prioritised for performance, whereas in cinematic pipelines and digital doubles, higher visual fidelity and physical accuracy are demanded. As virtual humans become more prominent across media, so does the need for flexible digital clothing workflows.

Industry tools such as Marvelous Designer and CLO 3D offer powerful workflows for designing and simulating garments using 2D patterns, producing results well suited to high-end visual outputs (Figure 1.1). However, these tools are external, commercial applications and require users to manage a multi-step pipeline involving export, import, reconfiguration and simulation tuning. While Unreal Engine does support cloth simulation through its Chaos Cloth system and most recently its Cloth Asset workflow, it does not support garment creation and its tools are primarily designed for efficiency in game development, prioritising performance over creative flexibility. As UE continues to expand into cinematic and virtual production workflows, this project aims to explore its potential for supporting more advanced, design-focused garment workflows directly within the engine.

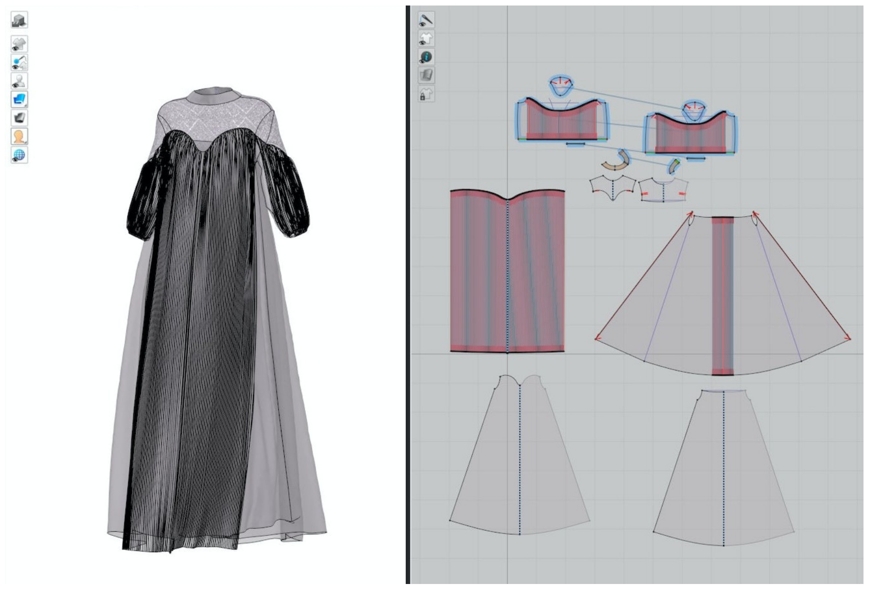


FIGURE 1.1: Example DCC tool overview including the 2D editor and 3D simulation view.

The result of this investigation into a streamlined, integrated alternative directly within Unreal Engine 5 is a custom plugin that introduces a new Editor Mode supporting 2D pattern drafting, automatic 3D mesh generation, digital sewing and real-time cloth simulation using the Chaos Cloth system. The goal is to automate and simplify as much of the workflow as possible while keeping the tool extensible and responsive to creative needs.

By bringing garment creation directly into the game engine environment, this work aims to reduce reliance on external tools, shorten iteration cycles and open up the possibility of more responsive garment design for real-time and cinematic applications. The project also serves as a technical investigation into how far Unreal Engine's native systems, such as Chaos Cloth, can be pushed to support cloth workflows typically handled by specialised software.

2 Literature Review

2.1 Cloth and Seam Simulation

Cloth simulation in interactive and offline pipelines typically builds on either force-based spring dynamics or its successor, position-based dynamics. In the classical mass-spring model, each vertex of a discretised fabric mesh is treated as a point mass connected to its neighbours by springs that resist stretch, shear and bending (Figure 2.1). Hooke's law governs the elastic force in each spring and a damping term stabilises high-frequency oscillations. Time integration, often semi-implicit Euler, updates vertex positions under the influence of gravity, wind and collision forces. While straightforward to implement, this approach can become unstable under stiff materials or large time steps, necessitating very small sub-steps or global solves to maintain realism [18].

Position-based dynamics reframes these same constraints as positional corrections. After predicting vertex displacements under external forces, PBD solvers iterate over each spring constraint, projecting pairs of vertices to restore their rest distance. This Gauss-Seidel-style projection loop naturally enforces stretch limits without solving large linear systems, yielding stability even at coarser time resolutions. Collision and self-collision are integrated as inequality constraints within the same projection framework, providing a robust, high-performance foundation for real-time cloth in games and interactive tools [15].

Finite Element Methods (FEM) model cloth as a continuous shell by discretising it into triangular or quadrilateral elements governed by a strain-energy functional and anisotropic constitutive laws. Each element contributes to a global stiffness matrix and implicit integration enforces equilibrium under external forces such as gravity and collisions, allowing much larger time steps and high-fidelity wrinkle and bending behaviour [2]. While FEM delivers unmatched physical accuracy, capturing complex material anisotropy and fine detail, it produces substantial computational and implementation overhead, which is why real-time engines like UE's

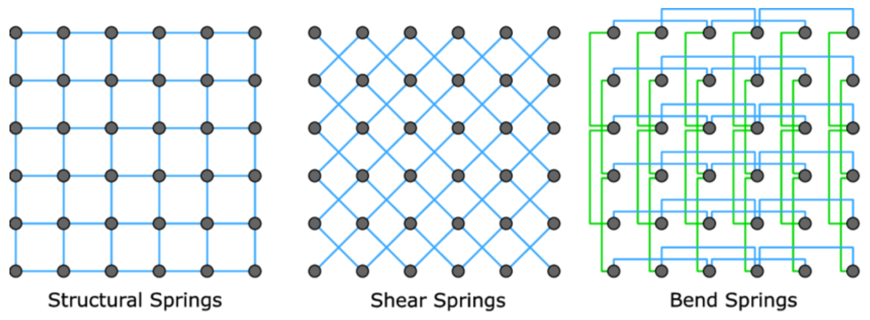


FIGURE 2.1: Spring forces.

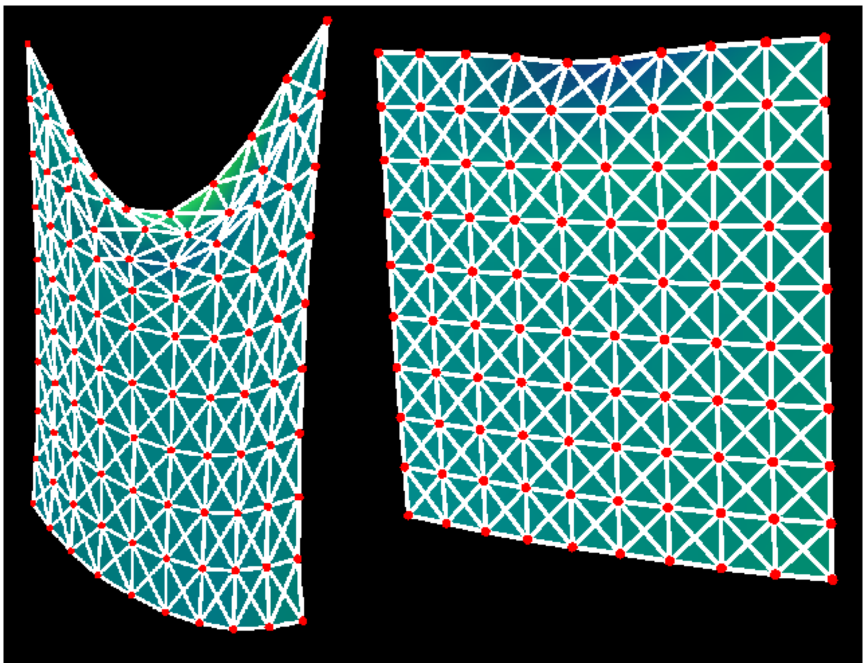


FIGURE 2.2: Elastic cloth simulation with springs.

Chaos Cloth often favour particle-based or PBD cores augmented with selective FEM-inspired constraints.

Seam modelling introduces additional complexity. In its simplest form, a seam is represented by extra distance constraints (“seam springs”) tying corresponding edge vertices of adjacent panels (Figure 2.4). Although this preserves continuity, it often yields overly stiff joins or puckering, since all springs share uniform stiffness parameters. Pabst et al. (2008) demonstrate that seams substantially alter local bending stiffness, and propose augmenting the spring network with dedicated bending elements whose stiffness decays with distance from the stitch line, calibrated against physical measurements. They further show that adaptive remeshing, subdividing triangles near seams according to a stiffness-based heuristic, can produce smooth, realistic drapes without prohibitive computational cost [17].

Together, these methods, augmented with seam-aware constraints, form the technical backbone of modern cloth systems, balancing computational efficiency with the fidelity required for both real-time and cinematic applications.

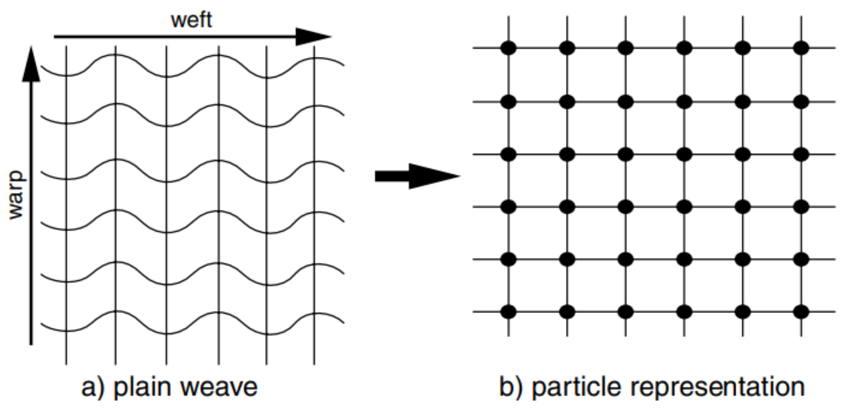


FIGURE 2.3: Particle representation.

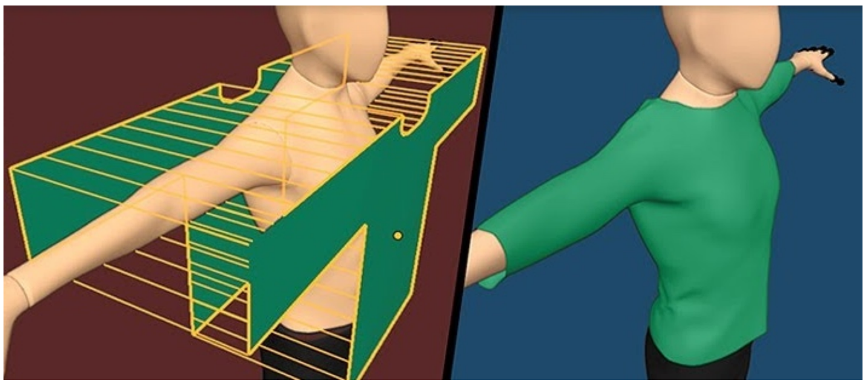


FIGURE 2.4: Seam springs visualisation.

2.2 Recent Advances in Physics Modelling

Cloth simulation is a dynamic and rapidly evolving field, progressing from early mass-spring models and position-based dynamics to sophisticated collision handling and, more recently, integration within cloth design applications. These advancements have broadened the scope of cloth simulation across animation, modelling, gaming, visual effects and digital fashion design.

In recent years, several key developments have emerged, including GPU-accelerated PBD, hybrid and implicit finite element methods and differentiable, data-driven surrogate models. This section begins by reviewing these foundational concepts and recent innovations to provide a comprehensive overview of the current state of cloth simulation. Recent advances in cloth simulation have seen significant progress in GPU-accelerated PBD frameworks aimed at real-time applications. Kim et al. (2025) demonstrate a fully GPU-offloaded PBD solver capable of running at 60 frames per second on standalone XR hardware by leveraging massively parallel constraint projection methods that effectively eliminate CPU bottlenecks [9]. In parallel, a WebGPU prototype achieves interactive rates simulating mass-spring cloth systems with up to 640,000 particles, illustrating a broader industry shift toward browser-based, GPU-first simulation engines that scale efficiently for large, complex scenes [20].

Baraff and Witkin (1998) introduced a fundamental framework for stable cloth simulation using implicit integration and global Newton methods. Their key contribution was enabling large time-step stability by solving the non-linear system of cloth constraints globally rather than relying on explicit time integration prone to instability and jitter [1]. This work essentially laid the foundation for many modern implicit and hybrid FEM solvers. Recent FEM Advances build upon this foundational idea by improving mesh adaptivity, stiffness handling and accuracy under extreme deformation without costly remeshing. They extend and refine the original implicit solver paradigm by incorporating modern numerical methods and hardware acceleration [34].

Separately, data-driven surrogate models incorporating differentiable physics have emerged as a powerful approach to capture intricate cloth behaviours while maintaining computational efficiency. Zhao et al. (2024) introduce physics-embedded deep learning architectures that encode energy-based cloth constraints, enabling real-time inference of fine wrinkles and collision phenomena without sacrificing physical plausibility [35]. Furthermore, a July 2025 study trains a learned Mass-Spring net directly from real-world fabric motion captured on video, optimising per-spring stiffness and damping parameters via force-and-impulse loss functions.

These machine learning-driven methods complement traditional solvers by providing enhanced detail and responsiveness in cloth simulation [4].

2.3 Survey of Existing Tools

2.3.1 Marvelous Designer

Several tools are available for digital pattern drafting and cloth simulation, with Marvelous Designer and CLO 3D being the most prominent. Both developed by the same company, these applications share core features but target different markets: CLO 3D is primarily designed for the digital fashion and toiling market, whereas Marvelous Designer targets the animation, modelling, gaming and visual effects industries, and is notably used by studios such as Weta Digital [6]. While alternative options exist, such as Cloth Weaver, a simplified Blender plugin, and the open-source 2D pattern drafting software Seamly2D, Marvelous Designer currently remains the industry standard and leading tool (Figure 2.5) [5].

FIGURE 2.5: Example image of finished design in Marvelous Designer.

Marvelous Designer (and CLO 3D) employ a pattern-based authoring approach directly inspired by traditional fashion industry practices. Users draft flat 2D sewing patterns in a specialised CAD interface, incorporating essential pattern elements such as curves, fold lines, seam allowances and notches (Figure 2.6). Seam correlations are then digitally defined to "sew" edges together. Each pattern piece is

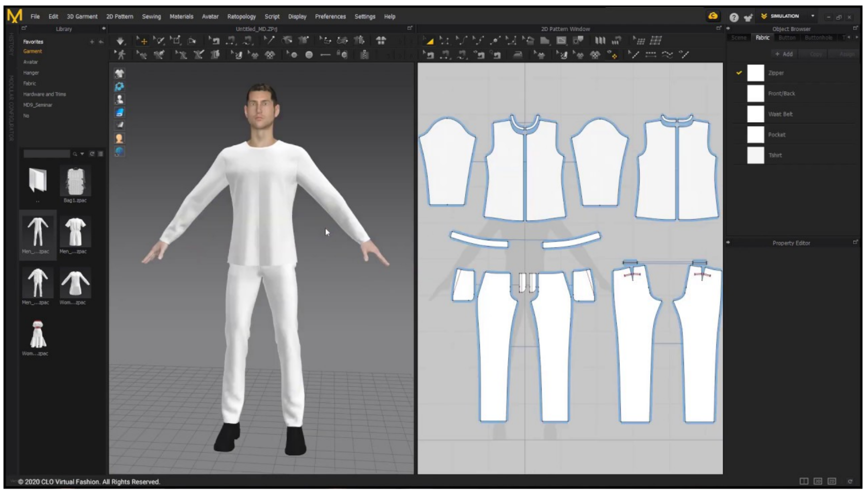


FIGURE 2.6: Marvelous Designer full UI overview.

instantly converted into a 3D mesh, which the cloth solver simulates using physics-based models, thereby draping the mesh over a customisable avatar. The 3D simulation window provides real-time feedback on silhouette, fit, wrinkles and drape as users adjust curve points or material properties [8].

In more detail it was observed that the workflow begins with avatar setup, where users import or create a base avatar with custom measurements. Pattern drafting occurs simultaneously in the 2D editor, where drawn shapes are immediately reflected as 3D meshes in the simulation view. Edge selections in the 2D editor establish spring constraints that act as seams during simulation, enabling realistic cloth draping and collision with the avatar. Marvelous Designer (and CLO 3D) implements cloth simulation with a discrete particle-based PBD solver rather than a continuous finite element system. Collision handling (with both the avatar and cloth self-collision) is naturally integrated into this projection loop, using per-vertex backstop offsets to maintain separation. Users can modify material properties, add details such as top-stitching, zippers and buttons, and manipulate the garment by direct interaction, all updated live by the solver. Fabric behaviour is further refined by anisotropic material parameters exposed in the UI. Weft, Warp and Shear intensities adjust resistance along the corresponding yarn directions, while a thickness parameter defines collision depth and internal pressure for layered garments. To accelerate this pipeline, Marvelous Designer offers an optional CUDA-based GPU solver (introduced in MD 9 and substantially improved in 2024), offloading both constraint projection and collision detection to parallel GPU kernels for speedups without sacrificing stability [12]. Completed patterns and meshes are exportable for use in digital content creation tools, game engines or physical pattern printing [13].

2.3.2 Unreal Engine

Chaos Cloth is Unreal Engine's dedicated cloth simulation module within the broader Chaos Physics suite, alongside Chaos Ragdoll, which handles realistic character physics and joint constraints, and Chaos Destruction, which drives procedural fracturing and collapse of rigid bodies, and many more. UE's native cloth-authoring ecosystem centers around two complementary paradigms: the in-Editor Cloth Paint

Tool for section-based cloth and the more recent Chaos Cloth Asset/Dataflow Editor for panel-based, node-driven setups. Both use an extended PBD solver and have optional GPU acceleration [21, 27].

For simple and more cinematic workflows, artists import assets from a DCC application as a UE Skeletal Mesh and then select material IDs on it, "paint" cloth behaviour regions in the Cloth Paint panel (Figure 2.8), adjust collision thickness and other simulation parameters, then click Simulate to see real-time drape on their asset (Figure 2.7). This lightweight pipeline requires no external tools, but limits the user to painting whole contiguous sections while still offering free control over those sections and producing results closer to a true physics simulation. Because this mode drives cloth purely through collision geometry, it maintains natural fabric contact, sliding and drape, eschewing the "pinned" look of bone-anchored regions that can make other approaches appear more game-like.

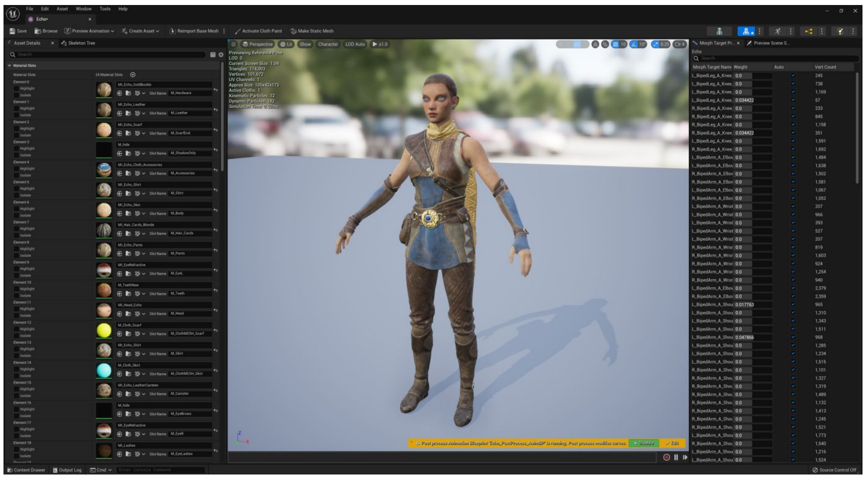


FIGURE 2.7: Unreal Engine cloth tool.

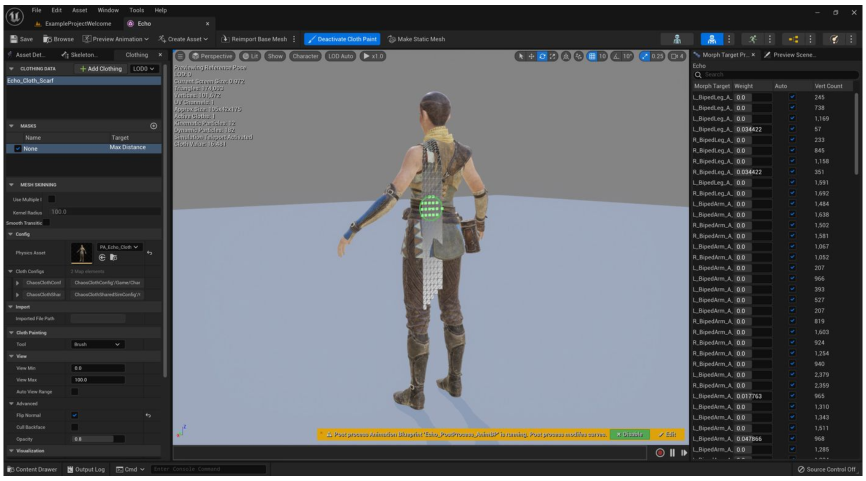


FIGURE 2.8: Unreal Engine cloth tool paint functionality.

The panel-based, Cloth Asset workflow, introduced in UE 5.4 and further developed in UE 5.5, lets the user import a full garment mesh (usually exported from Marvelous Designer via USD), extract it as a separate ChaosClothComponent and then drive its construction through an elaborate Dataflow graph of nodes. However, this pipeline depends heavily on a well rigged and skinned character. Tying cloth behaviour to the skin weights of the underlying character leads to a number of issues, most notably a loss of free-flowing physicality and an overly “painted-on” deformation style, highlighting the need for a more robust, collision-driven anchoring strategy rather than reusing skin-weight logic [14, 26].

UE’s built-in cloth tools offer a uniquely integrated, real-time solution for both quick turnarounds and more elaborate, game-ready garment setups. The Cloth Paint workflow provides an intuitive, collision-driven approach for artists who need fast, believable drapes on complex characters, while the Chaos Cloth Asset/Dataflow system brings a modular, node-based paradigm that can ingest detailed garments from external design tools and layer on procedural adjustments. As UE continues to evolve its solver performance and dataflow capabilities, these native systems lay a strong foundation for increasingly sophisticated in-engine simulation and possible garment authoring.

2.4 Identified Gaps in Existing Tools

While both Marvelous Designer and UE’s Chaos Cloth fill critical roles in digital garment workflows, each exhibits notable feature gaps, especially when pushed toward high-end animation, VFX and game pipelines.

Chaos Cloth delivers real-time performance via their solver, but sacrifices the nuanced physical accuracy required for subtle wrinkle definition and dynamic drape under complex motions. UE’s experimental Machine Learning Cloth plugin explicitly targets quasi-static garments, meaning it cannot faithfully reproduce swinging capes or flowing dresses in high-energy scenes [11]. When artists bake simulations through the Movie Render Queue, cache corruption or premature termination is commonplace unless subsampling is disabled, yet turning off subsampling reintroduces temporal aliasing and motion-blur artifacts that undermine cinematic polish [19]. Moreover, non-deterministic solver seeds lead to frame-for-frame inconsistencies, complicating plate matching and compositing in VFX workflows.

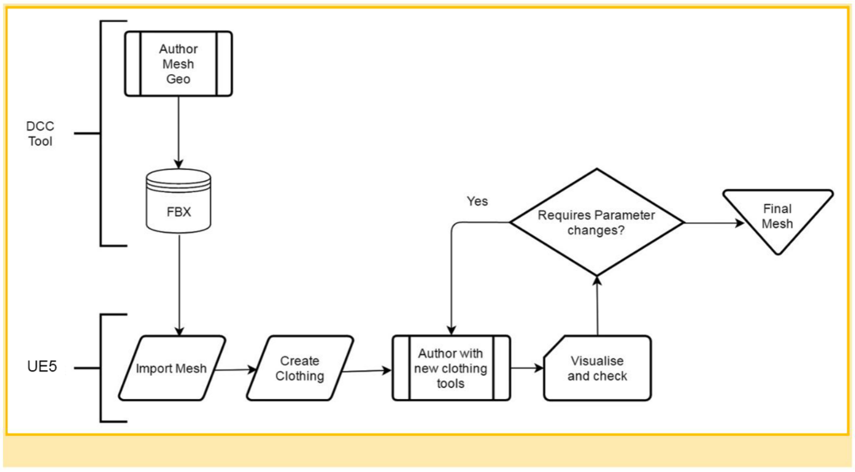


FIGURE 2.9: Unreal Engine cloth workflow.

Marvelous Designer’s LiveSync and USD export streamline the transfer of garments into UE, but material assignments, normal maps and UV setups frequently mismatch on import, resulting in flipped normals, missing tangents or transparent cloth that must be manually reconfigured [23]. Critically, there is no two-way pipeline: changes made in-engine cannot be pushed back to MD, breaking the iterative design loop that artists rely on for rapid garment refinement. Additionally, as mentioned before the cloth asset data flow causes issues due to the skinning of the assigned character [13].

UE cloth tools work well at collision-driven draping, but offer no built-in mechanism for pattern drafting. Unlike MD or CLO 3D, UE provides no 2D CAD interface for drawing pattern pieces, defining seam correlations or automatically generating meshes from flat patterns. Artists must pre-model garments in external applications, then import static meshes into UE, defeating the goal of a unified, in-engine pipeline (Figure 2.9). This gap forces studios and individuals to maintain parallel toolsets for pattern authoring and simulation, adding overhead and potential data-sync errors when moving between DCCs and the engine.

Taken together, these limitations underscore the absence of a fully integrated, art-directable pattern authoring workflow within Unreal Engine, an essential feature for cloth simulation that would unify design, simulation and rendering under a single, engine-native umbrella.

3 Technical Background

3.1 Human-Computer Interaction (HCI) in Digital Pattern-Authoring Tools

Users of digital pattern-authoring tools benefit immensely from a tightly coupled feedback loop between their actions and the visual outcome. As soon as a user drags, sketches or edits a 2D curve, the corresponding 3D drape or marker layout updates in real time, creating a "what-you-see-is-what-you-get" experience that narrows the gap between intention and result, reduces cognitive load, and accelerates learning curves [33]. This effect is enhanced by synchronised views: modifications made in the 2D pattern window immediately reflect in the 3D preview (and vice versa), helping users form accurate mental models of how flat geometry translates into draped cloth (Figure 3.1) [3].

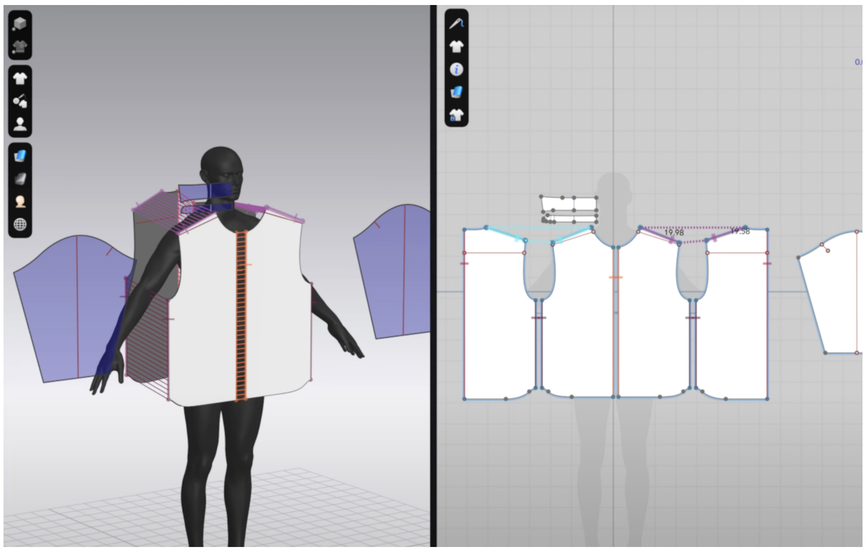


FIGURE 3.1: HCI example of CLO 3D UI design.

To further guide artists and prevent invalid edits, pattern tools show domain-specific constraints, such as seam joins, grainlines and darts, as interactive visual handles or "springs." These affordances highlight where edges can snap together and how panels may deform, ensuring users understand both the possibilities and limitations of their manipulations [10].

Beginners are supported through progressive disclosure: a streamlined core palette presents only the essential tools at first, while advanced features like grading, block libraries and other tools remain hidden until needed. Professionals can then customise their workspaces to expose the controls they rely on most [7]. Finally, robust history management, including comprehensive undo/redo stacks and semi-automating repetitive tasks (for example, mirror-sewing a sleeve) provide the confidence and efficiency necessary for exploratory design [16]. Together, these HCI elements create an intuitive yet powerful environment in digital pattern creation for both novice and expert pattern makers.

3.2 Unreal Engine's C++ Framework for Plugin Development

Unreal Engine's extensible C++ architecture is built around a clear separation of functionality into three module categories, Runtime, Editor and Developer, which together define a plugin's scope and dependencies (Figure 3.2). Runtime modules encapsulate features available in both the editor and a packaged build, exposing core systems such as input, rendering interfaces and basic physics. Editor modules, by contrast, are only loaded within the Unreal Editor and cannot be packaged for standalone runtime [28]. The most restricted tier, Developer modules, contains advanced engine internals, such as advanced skeletal-mesh processing utilities and certain Chaos Cloth solver modules and functions like accessing the simulated vertex ID of a mesh. These modules are intended for Epic's own subsystems rather than general plugin authors. Accessing these Developer modules often introduce build issues and maintenance challenges [24]. This extra indirection contrasts sharply with systems like Houdini, where point and primitive data are directly queryable and is a common source of friction when integrating custom plugins and cloth solvers.

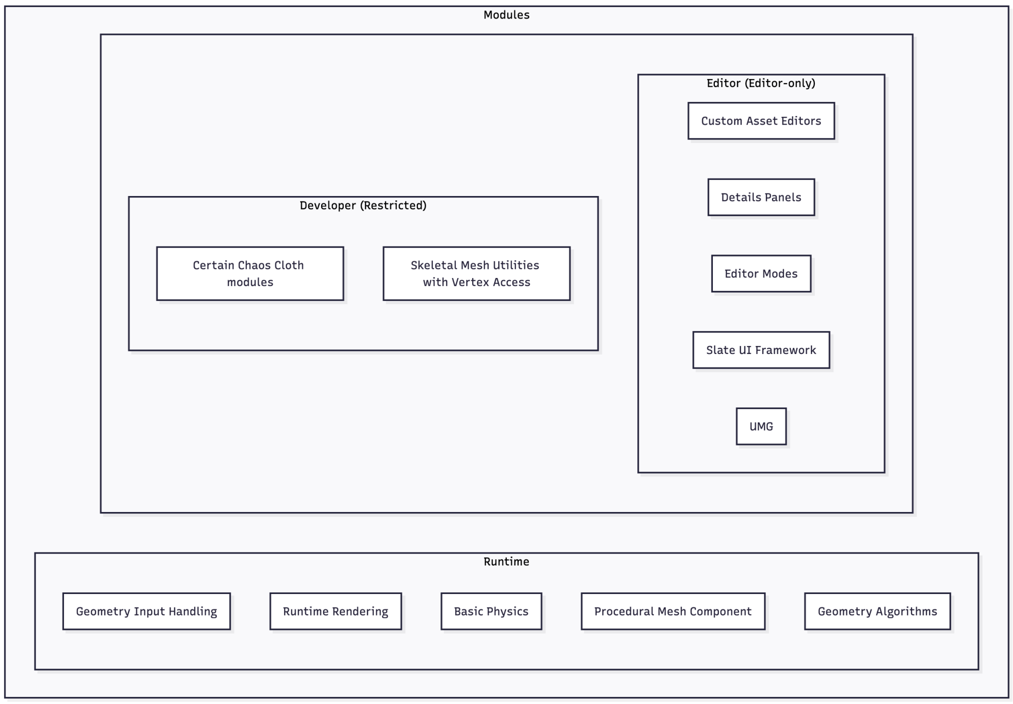


FIGURE 3.2: Unreal Engine modules overview.

UE provides a rich set of native modules for geometry processing and UI that often obviate the need for external libraries. At the core of mesh generation and editing is the Geometry Framework, which defines the FDynamicMesh3 data structures used throughout the Modeling tools and Geometry Scripting plugin [25]. Building on this, the Procedural Mesh Component supplies a simple, actor-component-based system for constructing triangle meshes at runtime, exposing vertex positions and curve tangents directly in C++. Meanwhile, the Geometry Algorithms module offers a suite of functions like triangulation, many of which underpin both the Modeling Mode tools and user-facing Blueprint nodes, and tasks like constrained Delaunay triangulation or transform actions can be performed with native function calls rather than importing and adapting third-party geometry libraries.

On the UI side, UE avoids third-party frameworks such as Qt in favor of its own Slate system, a fully custom, platform-agnostic declarative C++ framework that serves as the foundation for both editor tools and in-game interfaces. Unlike UMG, which is a designer-oriented layer built atop Slate, plugins that use Slate directly gain precise control over widget composition and styling, and can integrate seamlessly into the editor's existing details panels, toolbars and viewport overlays. Slate's deep integration with UE's system and architecture makes it largely maintainable for tool development[32].

In practice, leveraging these native modules streamlines development: the Procedural Mesh Component and Geometry Framework supply all necessary primitives for generating and modifying mesh topology, Geometry Algorithms provide robust implementations of operations like mesh triangulation and Slate allows for high-fidelity, interactive editor UIs without external dependencies. The primary challenge lies not in acquiring functionality but in discovering the appropriate module and API, and in navigating subtle versioning changes as Epic advances the engine. Nonetheless, by capitalising on UE's built-in geometry and UI subsystems, a plugin can remain lightweight, performant and fully in-step with ongoing engine improvements.

3.3 Data Structures

In Unreal Engine, nearly all in-game and editor data is built upon the UObject system, which serves as the base class for all serialisable engine types [29]. Classes derived from UObject are declared with the UCLASS macro, which generates metadata used by the engine's reflection, serialisation and garbage-collection systems. Members of these classes are exposed to the editor and Blueprint via the UPROPERTY macro, which not only marks a variable for serialisation but also specifies its visibility, editability and replication behaviour [30]. For lightweight, value-type data, such as simple structs that group parameters or small coordinate sets, UE employs the USTRUCT macro to define data containers that integrate seamlessly with the same reflection and property systems, allowing them to appear in UE's details panels and participate in undo/redo operations.

Actors (AActor subclasses) represent objects placed in a level and combine one or more UActorComponent instances to handle functionality such as transformations, rendering or physics. Components themselves are UObjects and can likewise declare UPROPERTIES that expose subobjects or asset references, such as material or mesh assets stored in .uasset files, to artists via the editor [31]. By leveraging these core type, it is possible for a plugin to maintain tight integration with UE's serialisation, reflection and rendering pipelines without introducing external data formats or bespoke file loaders. This consistent use of built-in types ensures that custom pattern-drafting assets, settings and procedural mesh constructs remain fully compatible with the engine's native toolchains and content pipelines.

Unreal Engine adheres to a disciplined, prefix-based naming convention that distinguishes types and variables at a glance. Structs and non-UObject classes carry an F prefix (e.g. FVector) [22]. Classes derived from UObject begin with U (e.g UStaticMeshComponent and UDataAsset), while those inheriting from AActor use A (such as ACharacter) to signal their place in the actor hierarchy. Template classes are prefixed with T (e.g. TArray, TMap), abstract interfaces with I (e.g. IInterface), and enums with E (e.g. EBlendMode). Slate widgets use S (e.g. SCompoundWidget), and concept-oriented helper structs sometimes carry C (e.g. CStaticClassProvider). Boolean variables are prefixed with b (e.g. bIsVisible), reinforcing their role as binary flags. By enforcing these conventions through the UnrealHeaderTool and build-time checks, UE ensures that code remains self-documenting, maintainable and understandable across its C++ system.

4 Solution

4.1 Project Scope and Initial Goals

This project explores the feasibility of building a simplified version of a virtual garment creation pipeline, similar in concept to Marvelous Designer, directly inside Unreal Engine 5.5. The core idea is to allow users to create 2D fabric patterns, define how those patterns are sewn together and view the resulting garment in 3D using UE's built-in Chaos Cloth simulation system. The implementation takes the form of a custom UE plugin, offering both a 2D pattern authoring interface and a 3D viewport for simulation and visualisation, with as much of the simulation workflow automated as possible.

The primary objective of this project was not to recreate every feature of existing garment tools, but to explore whether a full garment creation and simulation pipeline could be established entirely within Unreal Engine. While earlier tests in Houdini showed that such a pipeline could be implemented relatively easily, due to Houdini's procedural nature and built-in cloth tools, UE presents a much more challenging environment due to its fundamentally different architecture, real-time constraints and programming model. However, UE is also where many real-world production teams already work, particularly in virtual production and real-time rendering. As such, building this pipeline in UE has the potential to be significantly more useful.

In order to focus on core functionality, 2D pattern creation, mesh generation, sewing and cloth simulation, some auxiliary features such as pattern scaling and rotation in the UI were deliberately left out, which could be added in future iterations once the foundational system is proven to work. To focus on validating the core concept the project deprioritised aspects like UI polish. The goal was to establish a technically working pipeline rather than a production-ready design tool. Human-Computer Interaction was considered in the design of the user interface and concepts were implemented where feasible, but was not the main focus in this implementation. While the basic interaction model is functional, polish and usability refinements remain for future iterations.

4.2 Plugin Structure

The project began with configuring the UE development environment in the JetBrains Rider IDE and evaluating available plugin templates offered by UE. Two templates, the Editor Standalone Window and the Editor Mode, were identified as the most relevant and were combined: the Editor Mode serves as the core of the plugin, which is called "ClothDesign", while selected components from the Standalone Window template were incorporated for the separate 2D editor. The template code is preserved in the delivered tool so readers can compare the original scaffolding with the project's extensions and see where custom logic for this plugin was introduced.

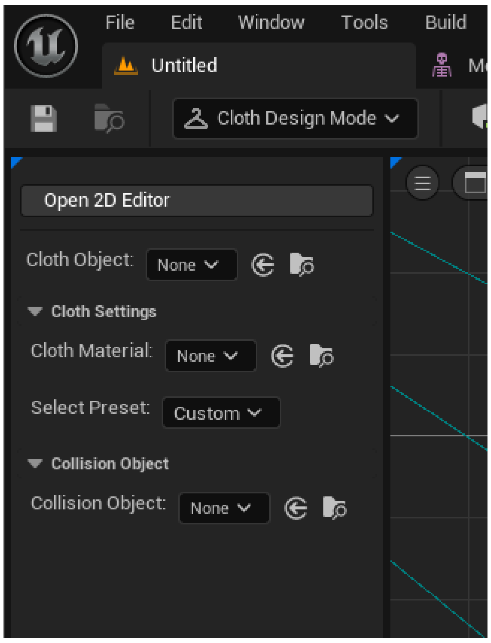
At startup, FClothDesignModule initialises FClothDesignStyle, which loads Slate icons, while FClothDesignModule registers FClothDesignCommands, which declares the Open2DWindow action, maps that command to Spawn2DWindow(), and registers a nomad tab spawner, managing the 2D tab lifecycle and CanvasWidget reference. When the editor mode is enabled, the engine constructs UClothDesignEditorMode, which creates FClothDesignToolkit. The toolkit builds the in-mode Slate UI, such as object picker, cloth settings and the "Open 2D Editor" button. Pressing it triggers FClothDesignModule::Spawn2DWindow(), which requests the global tab manager to spawn the tab. OnSpawn2DWindowTab() then creates an SDockTab and assigns an SClothDesignCanvas into CanvasWidget. SClothDesignCanvas, the main canvas class of this plugin, then handles 2D drawing, input and editing while the toolkit and module continue to provide in-editor controls.

The ClothDesign.uplugin file acts as the distribution and runtime descriptor for the plugin. It names the ClothDesign module, marks it as an editor-only target and specifies startup and loading behaviour and lists other engine plugins that must be present. The module's ClothDesign.Build.cs complements that by being the build description. It tells Unreal Build Tool which engine and plugin modules to link and include.

4.3 User Interface

FClothDesignToolkit constructs the 3D-side UI as a Slate SVerticalBox during Init(), composing controls from smaller functions, such as creating the button to open the 2D UI and creating object and preset pickers (Figure 4.1). Each control is data-bound with lightweight lambdas and delegates: SObjectPropertyEntryBox instances expose ObjectPath and OnObjectChanged bindings, with an optional scene-usage filter implemented by iterating world actors. SComboBox is used for preset selection with OnGenerateWidget and OnSelectionChanged lambdas, and buttons call OnClicked handlers that invoke module and toolkit methods. State is kept in member variables such as SelectedClothMesh and SelectedTextileMaterial, and UI-to-world effects are effected via explicit editor-side operations: ForEachComponentUsingSelectedMesh iterates the editor world and applies changes to USkeletalMeshComponent instances. Visual state in the toolkit (for example, the current preset label) is derived on-demand from these members so the UI always reflects the internal state. In short, the toolkit is a thin, Slate-based input layer that exposes asset pickers, parameter controls and actions, binds them with delegates, and translates user intent into immediate editor-side operations on scene components and into calls to the module (such as opening the 2D canvas).

The 2D editor's UI is built as a two-column Slate layout in FClothDesignModule. A fixed-width left control panel composed of nested containers, including expandable areas, object pickers, numeric entry boxes and grouped buttons, and a right-hand canvas (SClothDesignCanvas) that fills the remaining space, created with SAssignNew and stored in the module's CanvasWidget pointer. Controls forward intent rather than reimplementing canvas logic, object pickers call canvas accessor/mutator lambdas, the numeric entry box updates the canvas background scale, and action buttons invoke canvas methods, such as Generate Mesh, Sewing, Save, Clear All and more). Mode switching is implemented via small mode buttons in a top-right overlay and keyboard shortcuts. Those buttons call the canvas' mode and



Cloth Object: None

Cloth Material: None

Select Preset: Custom

Collision Object: None

(A) 3D UI

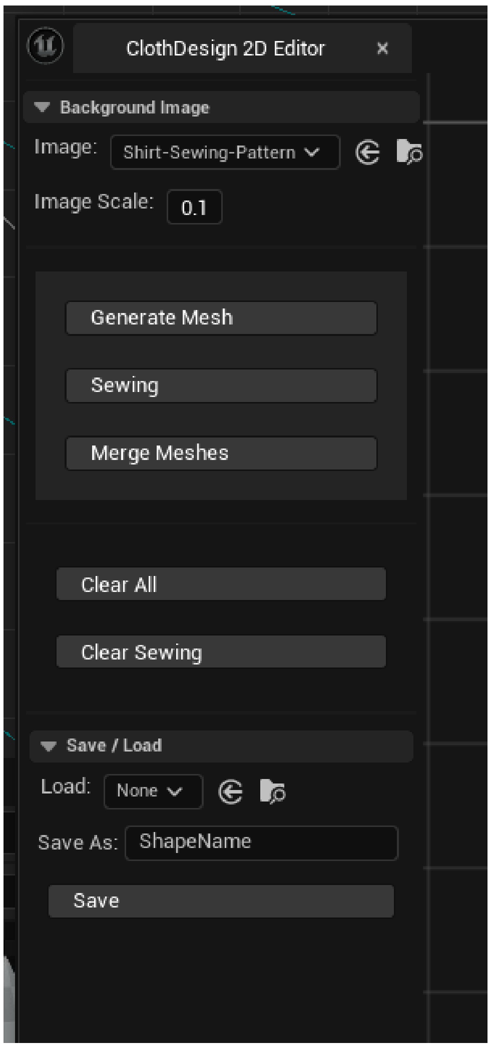


Image: Shirt-Sewing-Pattern

Image Scale: 0.1

Load: None

Save As: ShapeName

(B) 2D UI

FIGURE 4.1: User interface overview.

derive their highlight colour from the canvas' current mode so the toolbar reflects live state. Focus and lifecycle are managed explicitly: the module sets keyboard focus to the canvas on tab activation and schedules a ticker to ensure focus after construction, and all interactivity uses lightweight lambdas and direct method calls to keep the control panel as an input front while the canvas retains authoritative state and rendering responsibilities.

4.4 Canvas Widget

The SClothDesignCanvas Slate widget implements the 2D editing surface and is the bridge between the plugin's UI elements and its cloth-production logic. It renders the canvas (background, grid, in-progress and completed shapes, seam previews) inside the nomad tab and receives Slate input, translating mouse and keyboard events into editor actions. Input handling supports distinct modes using EClothEditor-Mode (Draw, Select, Sew), panning and zooming, shape point and Bézier tangent dragging, and keyboard-driven commands (mode switching, undo/redo, delete, focus). Coordinate transforms between screen and canvas space are applied consistently so UI interactions remain correct across zoom and pan.

State management, undo/redo and asset input and output are owned by the canvas: the current curvepoints, completed shapes, Bézier flags and pan/zoom are stored as a compact FCanvasState, and helper utilities encapsulate saving/loading shapes and restoring state. The canvas delegates low-level drawing to canvas helper classes and calls into the sewing and mesh utilities to generate FDynamicMesh3 results, APatternMesh actors when the user requests mesh creation and to build seam data. Interaction with the sewing layer is coordinated through a SewingManager abstraction that keeps seam definitions, preview points and runtime constraints. The canvas updates sewn-point caches and requests redraws when sewing state changes.

Therefore, SClothDesignCanvas is the single place where user gestures become editing operations and where those operations are materialised into pattern geometry and sewing data. It encapsulates rendering, input interpretation, state snapshots, asset persistence and the calls into the mesh-generation, sewing and merging subsystems, while remaining a Slate widget that the plugin's toolkit and module classes can control.

4.5 2D Pattern Drawing

FCanvasPaint implements the canvas rendering layer used by the 2D editor, which turns the canvas' geometric state into Slate draw calls while rendering is driven from the widget's OnPaint caller. The painter first handles background composition and grid drawing. If a background texture is present in the 2D UI's texture picker, it is painted via an FSlateBrush with configurable scale, and major/minor grid lines are produced by sweeping world-space grid coordinates, transforming them to screen space and emitting Slate line primitives. All drawing is clipped to the widget geometry and arranged using an explicit layer ordering so background, grid, shapes and points appear in the correct order (Figure 4.2).

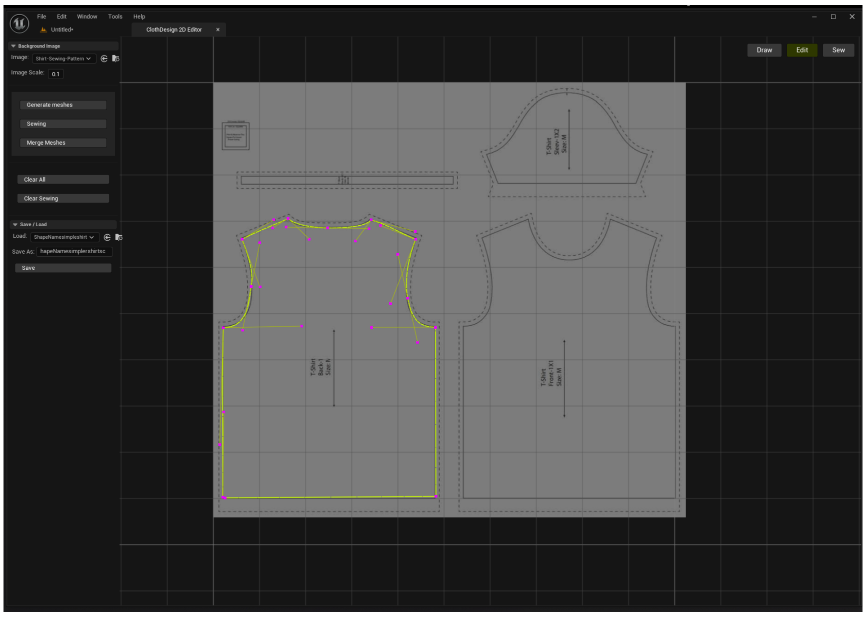


FIGURE 4.2: 2D UI with additional background image for tracing purposes.

Curve and shape rendering is performed in two similar flows: completed shapes and the current, in-progress curve. Both flows evaluate interpolated points, including Bézier tangents where applicable, and sample each segment into a fixed number of line samples to produce curved and straight lines that are rendered with line primitives. Tangent handles are drawn as connecting lines with small endpoint boxes and shape points are drawn as small filled boxes. To ensure robust downstream processing (triangulation and seam logic), shapes are finalised explicitly by the user (by pressing Enter) and are by default closed with a straight segment between the final and first point. If the user deliberately places the final point close to the start point the closing segment becomes visually negligible, effectively approximating the common DCC pattern of placing the end point on/near the start point to close a loop. Colours and thicknesses are specified centrally as constants so different elements (grid, shape lines, tangent handles, points, sewing lines) are visually distinct.

Seam-related rendering is integrated with the sewing manager: per-shape sets of sewn indices and previews are queried to determine which segments require highlighting. A BuildShortestArcSegments helper computes the minimal arc of segment indices between two endpoints on a looped shape and is used to accumulate segment indices that belong to seams. Finalised seams are drawn as paired start-start and end-end correspondence lines between pattern pieces to show the user the direction of the seams. Seams are drawn thicker and with an alternate colour when selected. The painter keeps each logical drawing step isolated and returns an updated layer index after each stage, ensuring predictable compositing when the Slate renderer blends the results into the UI.

Draw input is handled in FCanvasInputHandler, which converts the mouse click from widget screen coordinates to pattern space and immediately records the current canvas state on the undo stack. A new interpolation point is then appended to the active curve and the per-point Bézier flag is stored. Tangents are updated depending on the current Bézier mode: non-Bézier/linear points trigger a recalculation of N-tangents (for non-Bézier tangents), while Bézier mode calls an automatic tangent setter. Finally, the routine initialises the first and last tangents: first point receives a zero arrive tangent and a leave tangent of half the next segment, while the last point receives a symmetric arrive tangent and a zero leave tangent, so endpoints behave predictably for sampling and later triangulation.

4.6 Mesh Triangulation

APatternMesh and the FMeshTriangulation utility form the mesh-construction core of the pipeline: APatternMesh is a lightweight actor wrapper that holds a runtime mesh instance and the bookkeeping required for sewing and alignment, while FMeshTriangulation performs the geometric processing that transforms 2D pattern curves into triangulated, spawnable 3D mesh actors.

APatternMesh is an AActor containing a UProceduralMeshComponent and several arrays used by downstream systems: a copy of the generated dynamic mesh, a list of seam vertex IDs recorded at build time, sampled boundary points in 2D, the corresponding dynamic-mesh vertex indices and cached world-space positions for those sample vertices. The actor is spawned into the editor world when a mesh is created. The stored dynamic-mesh copy and the PolyIndexToVID mapping allow other subsystems (notably the sewing/ alignment code) to query vertex positions and map 2D boundary samples back into the 3D mesh for alignment and constraint construction.

The mesh construction subsystem in FMeshTriangulation uses a compact, dependency-free triangulation strategy that leverages UE's native constrained Delaunay triangulator combined with a lightweight Steiner-point generator. Curve sampling produces a dense boundary polyline, a uniform grid of candidate interior points is laid over the polygon bounding box, and an even-odd (ray-cast) point-in-polygon test filters seeds to produce reliable interior points. Boundary edges are supplied as constrained edges to UE's CDT, producing robust triangulations that capture concave and convex features (Figure 4.3). The number of Steiner points is hardcoded, but could be made adaptive to pattern size in future iterations.

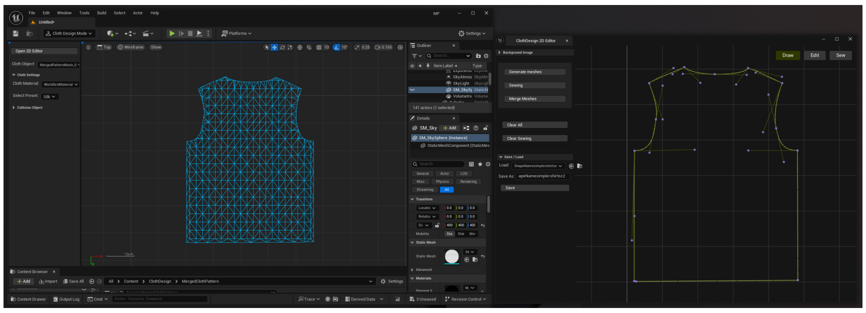


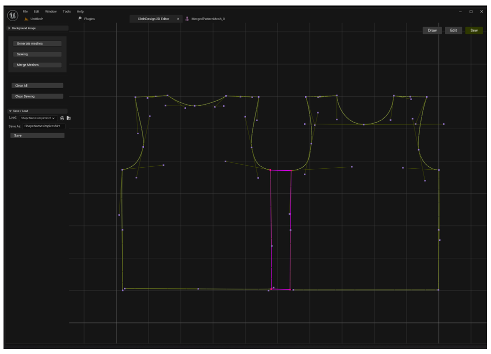
FIGURE 4.3: Pattern drawing with resulting mesh triangulation.

FMeshTriangulation implements the conversion pipeline from 2D interpolated curves to triangulated 3D geometry: it samples Bézier or linear curve segments, optionally records seam vertex indices, augments the boundary with grid interior seeds, and runs the constrained Delaunay triangulator. The triangulation is converted into a FDynamicMesh3, vertex and index buffers are extracted for a procedural mesh, and the geometry's pivot is recentered. CreateProceduralMesh spawns an APatternMesh actor and populates it with the final mesh data. Throughout the pipeline, mappings between polygon sample indices and dynamic-mesh vertex IDs are recorded and propagated into the spawned actor so that seam-building and runtime alignment can reference exact vertex IDs.

4.7 Sewing

The sewing system is implemented as a cohesive unit around FPatternSewing, a set of data structs and the main sewing class. Authoring-level types are FClickTarget (a ShapeIndex/PointIndex pair), FEdgeIndices and FSeamDefinition (which record the two edge intervals selected on pattern pieces), while FPatternSewingConstraint encapsulates the data needed for 3D alignment, such as the two UProceduralMeshComponent pointers, the mapped vertex indices and the sampled 2D screen points along each seam. User clicks drive a four-stage state (ESeamClickState) that fills AStartTarget, AEndTarget, BStartTarget and BEndTarget (Figure 4.4). When the sequence completes the code calls FinaliseSeamDefinitionByTargets, which samples a fixed number of 2D points along each selected interval, emits a UI-facing FSeamDefinition and constructs a matching FPatternSewingConstraint. During that process the implementation looks up the spawned pattern actors, consults each actor's PolyIndexToVID mapping to translate sampled polygon indices into dynamic-mesh vertex IDs, and appends the new constraint to the AllDefinedSeams list.

Seam alignment from authored 2D seams into 3D is handled by BuildAndAlignSeam and batched by BuildAndAlignAllSeams. BuildAndAlignSeam maps the stored screen samples to the nearest boundary samples on each spawned actor, converts


(A) Seams placed correctly.

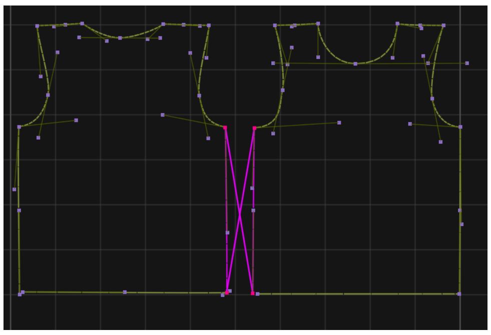

(B) Seams places incorrectly.

FIGURE 4.4: Comparison of correct and incorrect seam placement.

those to vertex IDs, filters invalid indices and produces paired vertex lists. It then computes world-space positions for the paired IDs and automatically detects ordering (reversing the B sequence when the reversed ordering produces a smaller average point-to-point distance). The paired vertex lists are cached on the corresponding APatternMesh as LastSeamVertexIDs and actors handed to AlignSeamMeshes, which computes an alignment that brings straight seam intervals into correspondence. AlignSeamMeshes derives seam direction vectors from endpoint vertices, constructs a rotation quaternion to rotate B's seam direction onto A's, rotates actor B about its seam midpoint, then computes an average translation between paired vertices and applies the offset to actor B so seam points meet in 3D. By design, only B is moved, which avoids unintended misalignment cascades when sewing three or more meshes into a single piece as long as the user maintains the sewing order. In future work, the tool should track connected sewn sets and apply transforms cumulatively so that all previously sewn pieces follow automatically.

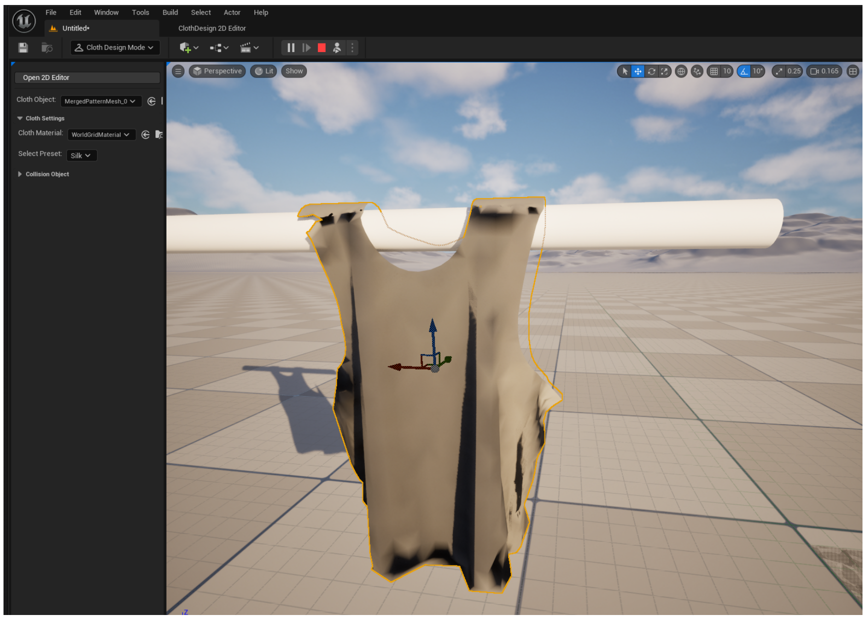
Input handling implements a four-click workflow: clicks are hit-tested against the in-progress curve and completed shapes. If the user clicks a point on the in-progress curve the handler prompts to finalise that curve before allowing seam authoring, ensuring seams are only created on completed shapes. Each valid hit advances the seam state, stores the clicked shape-point-pair into the corresponding FClickTarget, and updates preview sets so the painter highlights selected points while the user is authoring. The handler enforces shape-consistency validations and when the fourth click completes the sequence it invokes FinaliseSeamDefinitionByTargets, triggers seam preparation and updates the canvas' sewn-point caches for rendering.

Higher-level coordination is provided by MergeSewnPatternPieces, which hands sewn constraints to FPatternMerge to combine connected groups of actors into merged meshes suitable for export and later simulation.

4.8 Cloth Simulation and Chaos Cloth Integration

4.8.1 Pattern Merge (preparing sewn groups for simulation)

The pattern merge system (FPatternMerge) collects the spawned pattern actors and the recorded sewing constraints, then combines sewn groups of pattern pieces into single meshes that can be converted into assets ready for cloth simulation. The merge process is driven by FPatternSewingConstraint, a USTRUCT that stores the two procedural mesh components pointers, the corresponding vertex indices and sampled screen-space seam points. Using these constraints, an adjacency graph is constructed that captures which spawned actors are sewn to which others. Connected components of that graph identify independent sewn groups to be merged, so that (as in physical sewing too) only the pieces that are sewn together will act as one piece and not all generated pieces. Each connected component is merged only if it is self-contained, and components that have external seam edges connecting to actors outside the group are skipped. Vertices and triangles from the member APatternMesh instances are copied into a single FDynamicMesh3 while transforming each vertex into world space. The merge uses UE's FMergeCoincidentMeshEdges routine with hardcoded search and vertex weld thresholds to join the seam vertex sets generated during triangulation.



Cloth Object: MergedPatternMesh_0

Cloth Settings

Cloth Material: WorldGridMaterial

Select Preset: Silk

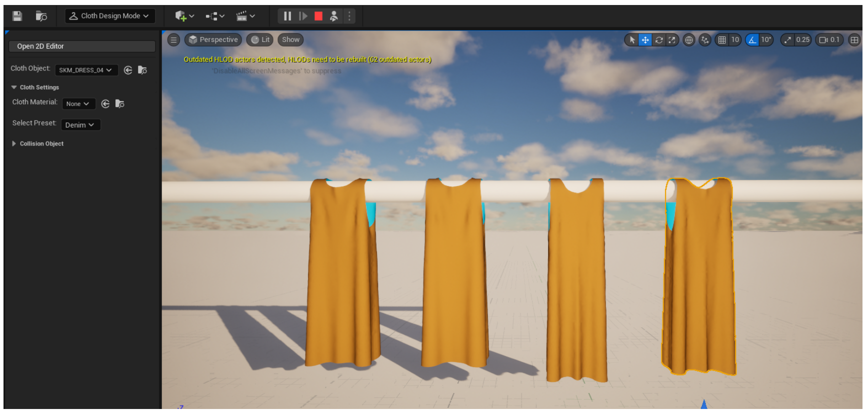
Collision Object

FIGURE 4.5: Simulation of produced cloth mesh.

The merged FDynamicMesh3 is turned into an in-editor actor by translating the mesh so its centroid becomes the actor origin, spawning a new APatternMesh, populating its procedural mesh section and storing the merged FDynamicMesh3 for later reference. Actor lists and internal seam metadata are updated so that any internal, now redundant, seam constraints are removed and the original per-piece actors are destroyed. This replacement keeps the spawn list coherent for subsequent operations, such as further merges, scene cleanup and simulation preparation. Since cloth simulation using Chaos Cloth requires skeletal assets, the merged mesh is converted to an interim UDynamicMesh and passed to UE Geometry Script helpers to generate bone-weighting and a skeletal mesh asset. A simple rigid bone weighting, with all vertices bound to a single bone, and a supplied skeleton asset, which is copied into the content folder during plugin installation, are used as the basis. The resulting skeletal mesh can then be spawned into the level as a SkeletalMeshActor. These steps produce a skeletal mesh saved in the plugin's content folder and a scene actor which can then be used with UE's cloth tooling and regular Chaos Cloth workflow (Figure 4.5).

4.8.2 Simulation Settings

To simplify the Chaos Cloth workflow, simulation settings are implemented in FCloth-SimSettings, which exposes a small preset system (EClothPreset, FPresetItem) and a mapped configuration type FClothPhysicalConfig that holds the numeric parameters used by the cloth solver (density, stiffness, tether stiffness and scale, drag, lift, friction, damping and gravity scale). Preset data are instantiated at construction and exposed to the 3D UI as selectable options. Selecting a preset (Figure 4.6) drives ApplyPresetToCloth, which iterates the clothing data assets attached to a target skeletal mesh selected in the cloth object picker and writes the preset values into the asset-level UChaosClothConfig fields. Unfortunately, UChaosClothSharedConfig appears to be inaccessible from the C++ side so the user still has to manually set more general cloth simulation settings such as iteration and substep count. After updating asset data, the code invalidates cached clothing data, marks packages dirty and triggers refresh calls so the changes take effect immediately in-editor. SetClothCollisionFlags sets the common skeletal-mesh component flags needed for cloth collisions on the mesh asset selected by the user in the cloth object picker on the UI.



Cloth Object: SKM_DRESS_04
Cloth Material: None
Select Preset: Denim

Outdated HLOD actors detected, HLODs need to be rebuilt (62 outdated actors)
'DisableAllScreenMessages' to suppress

FIGURE 4.6: Example of the different simulation presets applied to an imported mesh (left to right: denim, leather, silk, jersey).

4.9 Utilities

4.9.1 Save and Load in the 2D Editor

The editor-side shape storage is implemented with a small UDataAsset type, UClothShapeAsset, and a helper layer, FPatternAssets and FPatternAssetManager. UClothShapeAsset stores the drawing primitives as native UPROPERTY arrays: per-point data (input key, 2D position, Bézier tangents and a per-point Bézier flag) for the current in-progress curve, and an array of completed shapes with the same point records on each shape. The save routine converts the runtime curve representation into the asset USTRUCTs, either creating or reusing a dedicated plugin folder in the project's content drawer and saving to disk. The saved asset is a standard UE asset (uasset), ensuring compatibility with normal editor workflows and content browser usage.

Loading reconstructs the canvas' runtime geometry from the stored asset by reversing the conversion: each saved point entry is turned back into FInterpCurvePoint<FVector2D> entries and Bézier-flag arrays, which are then assembled into an FCanvasState struct used by the canvas widget. The loader resets interactive state (selection indices, pan/zoom) to sensible defaults after restoring geometry. FPatternAssetManager holds the currently selected UClothShapeAsset as a pointer, exposes a path accessor for UI linking and exposes OnShapeAssetSelected which invokes the loader to produce an FCanvasState from the selected asset for immediate use in the 2D editor. The current asset format captures only shape geometry and per-point tangent/flag data, while seam and sewing data are not recorded in the UClothShapeAsset and therefore must be redefined after meshes are regenerated. This is due to the sewing selection and export operations being dependent on corresponding generated mesh in the scene. This behaviour is reflected in the UI and documented as a limitation.

4.9.2 Other Supporting Utilities

In addition to the core classes, a set of utility functions was developed under FCanvasUtils to provide common editing operations and geometric helpers. These functions play an essential role in ensuring smooth interaction, maintaining geometric correctness and extending the overall usability of the system.

For state management, the utilities include methods to save the current canvas configuration and to perform both undo and redo operations. This mechanism allows edits to be applied non-destructively and gives the user the ability to step backward or forward through changes.

Editing interactions are also supported at a finer level through the input handler. The HandleSelect method determines whether a user click corresponds to a curve point, a tangent handle or a seam definition. By comparing the click position to stored points and tangents and applying distance thresholds that are adjusted for zoom level, the system can resolve the intended target. When a valid selection is made, the utility immediately records the current canvas state for undo, updates the selected indices and enables dragging for either the point or the tangent handle. Since seam editing is done through selection of the line connecting the sewn points on each shape, similar logic is applied to seam definitions, where the system calculates distances from the click location to each seam segment using helper functions such as DistPointToSegmentSq, ensuring that seam selection behaves predictably across zoom levels and transformations.

For handling curves, FCanvasUtils also provides functionality to recalculate tangent vectors for linear points, which is used in both FCanvasInputHandler and SClothDesignCanvas. This ensures that shapes remain drawn correctly when Bézier interpolation is disabled, which is important for precise shape construction while editing shapes. The utilities also include geometric operations that are useful in the context of sewing and simulation. A centroid computation for meshes is provided, which produces a centred pivot location that is assigned to meshes during triangulation and merging operations.

Together, these supporting utilities form foundational functionality of the system's editing workflows. They enable robust geometric processing, ensure that curve and mesh operations behave predictably, and provide a safety net for user interaction through undo and redo functionality.

4.10 Overall User Workflow

Following the detailed explanation of classes, templates and custom extensions, it is useful to consider the end result from the user's perspective. After the plugin loads, the user enters the custom editor mode to access the 3D-side controls: asset pickers for cloth, collision and material, as well as preset selection and a button that launches the separate 2D editor.

The 2D window is the authoring surface for pattern pattern pieces and seams (Figure 4.7). There are three modes: draw, edit and sew and further UI controls in a panel on the left. Drawing mode places points (with a toggle between Bézier (B key) and non-Bézier/linear (N key) point types), edit mode exposes point and Bézier point handle manipulation with the ability to separate the two Bézier handles of a point (S key), while deletion (backspace or delete key) and an undo (Ctrl + Z) and redo (Ctrl + Y) is implemented via the widely used keyboard shortcuts. Sewing mode lets the user define seam correspondences by clicking start and end points on two pattern pieces. The UI offers seam removal by clicking on the lines connecting the seam points while in edit mode and deleting them. It also offers clearing of all sewing or all shapes through buttons in the left column and optionally saving and loading of drawn shape data. Optionally, before starting to draw, background textures, previously imported into the content browser, can be loaded to trace existing 2D patterns.

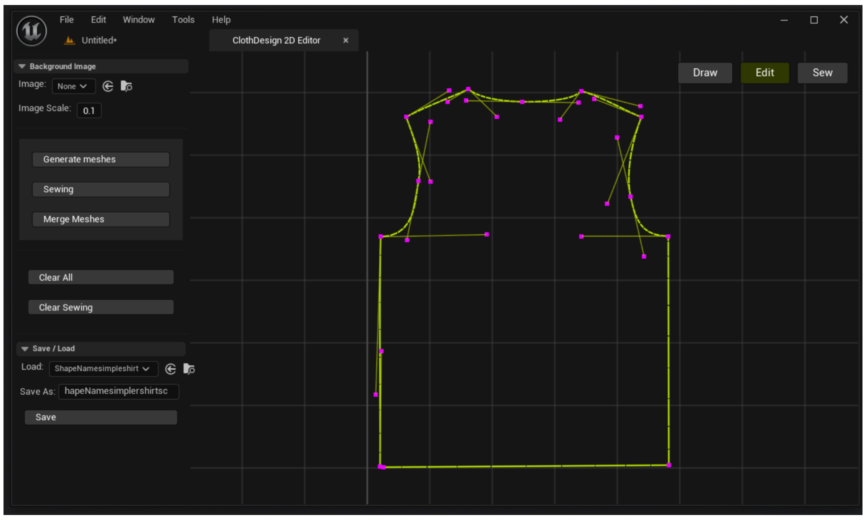


Image: None

Image Scale: 0.1

Load: ShapeNamesimpleshirt

Save As: hapeNamesimplershirtsc

FIGURE 4.7: 2D UI overview with drawn shape.

Once drawn shapes are finalised, the canvas can triangulate and build 3D meshes (using the "Generate Meshes" button) which the user then positions in the 3D viewport to match collision geometry. Then the user can select which edges on the drawn shapes to sew and finalise the sewing by clicking "Sewing", before exporting the pattern pieces into a merged mesh ready for simulation ("Merge Meshes" button). The sewing and merging is currently still separated for more user control of the placement of sewn pieces in the 3D scene before merging into a single, simulatable mesh.

While several workflow steps are automated by the tool, once a skeletal mesh has been produced, a few manual steps remain before the mesh can be simulated: on the generated, merged skeletal mesh saved in the content browser, clothing data must be created and applied, cloth-painting mode used to paint the whole object and painting mode disabled again. Optional simulation improvements, for example raising per-cloth iteration or subdivision counts to prevent collision penetration, can also be applied via the clothing settings. These manual steps are standard for UE cloth workflows and require only a short amount of time.

Once this has been set up the skeletal mesh, earlier placed in the scene during the merging of pattern pieces, can then be selected in the cloth object picker on the 3D UI for further processing and automated set up of simulation settings, and the mesh is ready to be simulated. Additionally, the user can select a material for the mesh selected in the cloth object picker and apply a preset of cloth simulation settings which are intended to reflect different materials (denim, leather, silk, jersey or custom). The cloth and material asset pickers and preset system on the 3D UI also work with externally imported cloth meshes (e.g. from Marvelous Designer), allowing rapid assignment of materials, simulation settings and cloth presets to streamline setup for any cloth asset in UE.

5 Evaluation

5.1 Technical Challenges and Mitigation

5.1.1 Triangulation

Initial triangulation issues were traced to a lack of interior (Steiner) points rather than a failure of the triangulation engine itself. Investigation showed that UE's constrained Delaunay implementation does not automatically seed interior points, which led to excessively elongated or "pointy" triangles when only boundary vertices were supplied (Figure 5.1). External libraries (for example CGAL or Triangle) were considered, but integrating such dependencies would have introduced significant bridging code and maintenance overhead. The chosen remedy was a lightweight, deterministic approach: a bounding-box grid of candidate interior seeds combined with an even-odd point-in-polygon filter, then passed as input to UE's native CDT. This produced predictable mesh density and quality while preserving a dependency-free implementation.

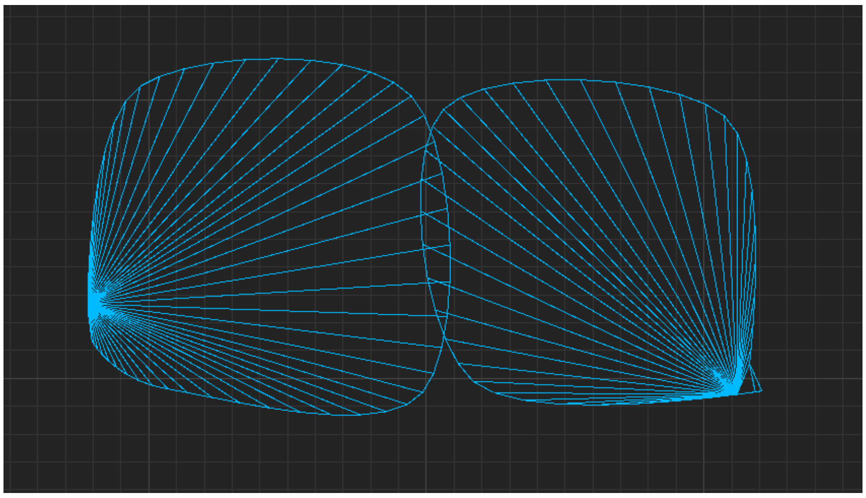


FIGURE 5.1: Simple triangulation without interior points resulting in elongated triangles.

5.1.2 Sewing and Seam Simulation

Implementing seam behaviour to match the level of integrated physical sewing found in specialised DCC tools proved infeasible within the project scope because the low-level internals, that are required for direct Chaos Cloth manipulation, are not readily exposed. Rather than attempting an invasive integration with hidden engine systems, a pragmatic approximation was developed that supports accurate authoring and deterministic alignment without relying on internal simulation hooks. The implemented workflow records seam endpoint pairs, supports undo/redo and deletion, and aligns corresponding straight-edge intervals in 3D. This approach yields a compact four-click user flow (start/end on piece A, start/end on piece B) that is robust and easy to use, even though it does not simulate seam physics interactively.

A current limitation of the system is that the handling of sewing connections and sewing alignments in the 3D scene is not fully separated from mesh generation. At present, sewing operations can only be performed after the 3D meshes have been created, which requires the user to explicitly generate the mesh before initiating any sewing. This dependency is enforced by warnings in the interface. Furthermore, because sewing depends on the existence of fully generated shapes, sewing information is not yet stored in the save and load functionality of 2D patterns, which introduces a workflow inefficiency. A future iteration of the system will decouple sewing from mesh generation, allowing for more flexible editing and persistence of sewing data alongside the 2D pattern.

5.1.3 Collision and Engine Accessibility

Early experimentation with cloth collision against animated skeletal geometry revealed unreliable behaviour. Robust collision with animated skeletons could not be achieved within the available time frame. Static geometry produced more consistent results, while native, simple primitives (boxes, spheres) proved to be the most reliable collision targets in practice.

In addition, several editor operations that are trivial from the UI (for example exporting static/skeletal meshes or manipulating clothing-data properties in the Skeletal Mesh editor) presented limited or undocumented access via C++. Locating programmatic equivalents required substantial investigation and some operations remained infeasible to automate. This engine module inaccessibility also leads to a significant usability limitation, as the current workflow requires the user to regenerate the 3D meshes and reapply sewing after editing a 2D shape. This prevents a continuous editing experience in line with HCI principles where pattern pieces and sewing connections update dynamically as the shape is modified. Achieving such a workflow would require establishing a direct link to the runtime simulation mesh data, such as the vertex positions generated by the Chaos Cloth system, allowing sewing to adapt physically in real time, which represents an area for future improvement.

5.1.4 Scope and Prioritisation

Faced with the above integration constraints and a fixed project timeline, emphasis was placed on delivering a complete, usable pipeline, from pattern drawing through triangulation, seam authoring and export, rather than attempting to fully reproduce the advanced simulation features of dedicated garment tools. The resulting implementation therefore favours a dependable, end-to-end authoring experience and clear, undoable user operations over deeper but riskier integrations with hidden engine internals. This prioritisation enabled delivery of the core functionality while leaving higher-fidelity seam simulation and advanced collision workflows as logical targets for future work.

5.2 Results

5.2.1 Performance and Quality

The plugin demonstrates responsive runtime behaviour: UI actions, such as opening the 2D editor, drawing shapes, generating meshes and invoking sewing operations, execute quickly on typical development hardware. Installation is straightforward, copying the plugin folders into an existing project yields immediate availability, and was tested on other Linux machines with UE 5.5 installed.

The plugin underwent testing by peers, whose feedback emphasised the clarity of the UI, particularly features such as 2D seam definition highlighting. The testing confirmed previously identified limitations and provided suggestions for future enhancements to improve usability.

Functionally, the implemented features are stable and perform as intended for the majority of workflows tested. Remaining issues are primarily feature gaps rather than reliability problems, except for some smaller issues. The implementation deliberately prioritised the correctness of output and user-facing functionality over deeper code-level architecture early in development. Later code refactoring improved readability and structure, but opportunities remain to streamline data flow and tighten separation of responsibilities, such as a clearer API boundary between UI and processing layers, and further refactoring to reduce coupling between canvas state and sewing/mesh subsystems, as well as more modular performance for different sized pattern sets.

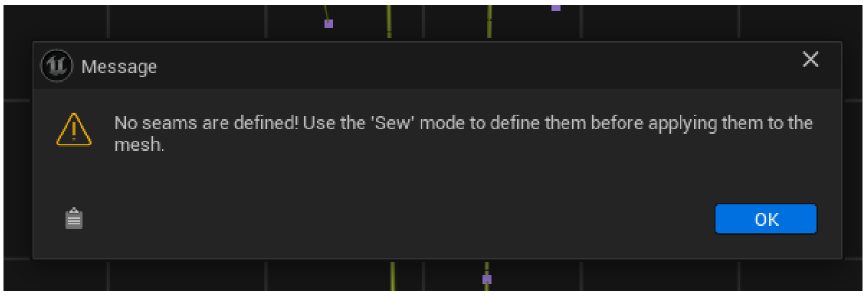
5.2.2 Comparison to Unreal Engine's Cloth Setup

The plugin extends Unreal Engine's native capabilities by providing an integrated pattern-authoring and sewing pipeline that does not exist in the engine by default. Whereas UE typically relies on externally authored garments, commonly produced in DCC tools like Marvelous Designer, and requires manual steps in the Skeletal Mesh editor to prepare cloth for simulation, this implementation enables drawing multiple 2D pattern pieces, saving/loading pattern data, defining seam correspondences and exporting merged skeletal meshes directly within the editor. These features reduce the need to perform early-stage pattern work in an external DCC tool and streamline several preparatory steps for cloth simulation.

At the same time, the plugin is intentionally a simplified, scoped solution rather than a replacement for a full-featured DCC system. Seam alignment is performed as a geometric operation and preview fidelity during layout is therefore limited. Likewise, the plugin does not fully obviate the standard UE steps that remain necessary for production-quality simulation, as creating and assigning clothing data are still required after export. The project demonstrates that a compact, end-to-end draw-to-sew-to-simulate pipeline can be realised inside UE to accelerate pattern authoring and initial setup, while recognising that it is not a substitute for specialised garment tools.

5.2.3 UI and HCI

A few minor issues remain in the current UI implementation. Occasionally, the tangents of in-progress Bézier points reset when a shape is edited and then extended, which interrupts the expected continuity of the curve. Furthermore, because there is no persistent flag marking Bézier tangents as separated, the user must manually press the separation shortcut again when returning to edit the point later. Additionally, if a shape begins with Bézier points, contains intermediate non-Bézier points, and then closes on a Bézier point, all intermediate non-Bézier points are displayed as if they were Béziers but without active tangent handles. If closing on a linear point, this issue does not appear. Practical feature extensions also include mesh transformation tools (move, copy, scale of 2D pattern pieces) and improved export options for downstream workflows.


Message

No seams are defined! Use the 'Sew' mode to define them before applying them to the mesh.

FIGURE 5.2: User warning when trying to sew without having first defined any seams.

To reduce authoring errors, the editor incorporates redundant feedback and error-prevention mechanisms around sewing. Sewing points cannot be placed in the 2D editor before meshes are generated, preventing invalid states by design. When switching into edit mode, a transient reminder appears in the corner of the interface to prompt the user to regenerate meshes if the shape has been modified. Entering sewing mode triggers a similar reminder, both serving as lightweight, non-intrusive nudges that fade after five seconds. These are complemented by a more explicit warning dialog if the user still attempts to sew without valid meshes, ensuring that errors are caught before they propagate (Figure 5.2). As a potential future improvement, this guarding could be extended by saving a canvas state at mesh generation time and restricting sewing to unmodified meshes, thereby enforcing stricter consistency while further reducing opportunities for user error.

The UI design follows a workflow-driven approach. Controls are grouped and ordered to reflect the most common sequence of user tasks: initial asset selection and configuration, pattern authoring, mesh generation, sewing operations, then save/load. Non-essential controls are stored in collapsible sections to reduce visual clutter, while frequently used mode controls (Draw, Edit, Sew) are emphasised and placed in a prominent overlay to ensure discoverability and efficient switching.

The interface was designed to be familiar to artists working in DCC tools: common interactions and shortcuts were implemented to match typical workflows, like mode hotkeys, keyboard focus on drawn shapes and middle-mouse dragging to pan the 2D canvas. The 3D toolkit acts as a compact control panel and the 2D canvas provides an authoring surface with drawing, edit and sewing modes. Controls are data-bound so the panel forwards intent while the canvas remains authoritative for geometry and state.

6 Conclusion

6.1 Conclusion

6.1.1 Summary

The project delivered a working, end-to-end prototype for cloth pattern authoring inside Unreal Engine 5, implementing a pipeline from drawing over mesh triangulation and sewing to export using a UE editor mode and a dedicated 2D canvas (Figure 6.1). The toolkit exposes asset pickers, presets and scene controls, while the canvas supports Bézier and linear drawing, editing, seam authoring, undo/redo and mesh generation. A grid-seeded constrained-Delaunay approach was used for triangulation to produce reliable mesh quality and seams are recorded, meshes aligned and merged to produce skeletal meshes ready for UE's cloth workflow. The implementation emphasises a dependable authoring experience and clear user operations leveraging UE's native geometry tools and modules, producing a simple, usable toolchain that introduces many preparatory steps for cloth simulation into Unreal Engine and enabling user iteration, while remaining compatible with UE's established simulation pipeline.

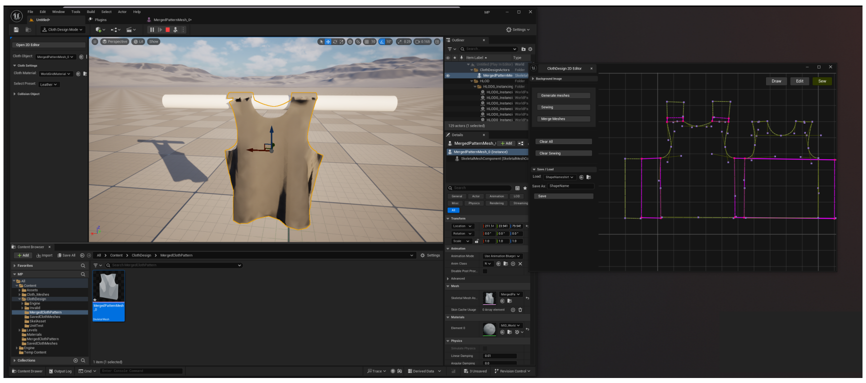


FIGURE 6.1: Full overview of drawn, triangulated, sewn and then simulated mesh in UE.

6.1.2 Future Work and Extensibility

Several clear extensions remain to raise fidelity, robustness and automation. Immediate priorities include codebase refactoring to improve modularity and data flow, serialisation of sewing state, and additional mesh-manipulation tools. Mid-term objectives are to introduce seam-level physical behaviour (interactive seam simulation or tighter integration with a cloth solver), more robust collision handling for complex static geometry and animated skeletal meshes, and further automation of clothing data setup to reduce manual editor tasks. Longer-term possibilities include support for non-uniform seam matching (unequal-edge stitching and ruffling), richer textile behaviour and multi-layered garments as well as additional garment features (top-stitching, trim, fastenings). Additionally, the UI should be extended to manage multiple garments concurrently, allowing multiple pattern sets to be authored, arranged and simulated together. Architecturally, the implementation is positioned as a foundation: the current codebase can be extended according to future project priorities, enabling a path from a pragmatic in-editor authoring prototype toward a more feature-complete production tool.

Overall, the project demonstrates that a compact, maintainable pattern-authoring and sewing pipeline can be realised inside Unreal Engine and provides a practical base for further development toward higher-fidelity simulation and production workflows.

Bibliography

[1] Baraff, D. and Witkin, A. “Large Steps in Cloth Simulation”. In: *COMPUTER GRAPHICS Proceedings, Annual Conference Series* (1998). URL: https://www.cs.cmu.edu/~baraff/papers/sig98.pdf.

[2] Bender, J. and Deul, C. “Efficient cloth simulation using an adaptive finite element method”. In: *Workshop on Virtual Reality Interaction and Physical Simulation, VRIPHYS* (2012). URL: https://diglib.eg.org/server/api/core/bitstreams/a2a49c8b-e219-4daa-856a-ea0bad48ef12/content.

[3] Chan, M. “Mental Models”. In: *NN Group* (2024). URL: https://www.nngroup.com/articles/mental-models.

[4] Chen, G. et al. “Learning Simulatable Models of Cloth with Spatially-varying Constitutive Properties”. In: *arXiv* (2025). URL: https://arxiv.org/html/2507.21288v2.

[5] Choi, K.-H. “3D dynamic fashion design development using digital technology and its potential in online platforms”. In: *International Journal of Interdisciplinary Research: Fashion and Textiles* (2022). URL: https://fashionandtextiles.springeropen.com/articles/10.1186/s40691-021-00286-1.

[6] Conferences, S. “The Magic Behind Marvelous Designer: An Interview With Jaden Oh”. In: *ACMSIGGRAPH Blog* (2024). URL: https://blog.siggraph.org/2024/03/the-magic-behind-marvelous-designer-an-interview-with-jaden-oh.html.

[7] Dearden, A. and Finlay, J. “Pattern Languages in HCI: A Critical Review”. In: *HUMAN–COMPUTER INTERACTION, 2006, Volume 21,* (2006). URL: https://research.cs.vt.edu/ns/cs5724papers/dearden-patterns-hci09.pdf.

[8] Habib, M. A. and Alam, M. S. “A Comparative Study of 3D Virtual Pattern and Traditional Pattern Making”. In: *Journal of Textile Science and Technology* (2024). URL: https://www.researchgate.net/publication/377047849_A_Comparative_Study_of_3D_Virtual_Pattern_and_Traditional_Pattern_Making.

[9] Kim, T., Ma, J., and Hong, M. “Real-Time Cloth Simulation in Extended Reality: Comparative Study Between Unity Cloth Model and Position-Based Dynamics Model with GPU”. In: *MDPI, Journal of Applied Sciences* (2025). URL: https://www.mdpi.com/2076-3417/15/12/6611.

[10] Liu, K. et al. “3D interactive garment pattern-making technology”. In: *Computer Aided Design, Volume 104* (2018). URL: https://www.sciencedirect.com/science/article/pii/S0010448518304093.

[11] UE Documentation Machine Learning. Accessed August 2025. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/machine-learning-cloth-simulation-overview.

[12] Marvelous Designer Documentation. *Marvelous Designer 2024.2*. 2024. URL: https://www.marvelousdesigner.com/learn/newfeature?v=2024.2.

[13] Marvelous Designer Sofware. Accessed June 2025. 2025. URL: https://www.marvelousdesigner.com/.

[14] Marvelous Designer Sofware Advice. Accessed July 2025. 2025. URL: https://support.marvelousdesigner.com/hc/en-us/articles/47358145573401--Tips-Tricks-Discover-Better-Workflow-with-Marvelous-Designer-and-Unreal-Engine.

[15] Müller, M. et al. "Position Based Dynamics". In: *3rd Workshop in Virtual Reality Interactions and Physical Simulation, VRIPHYS* (2006). URL: matthias-research.github.io.

[16] Murano, P. and Sethi, T. "Anthropomorphic User Interface Feedback in a Sewing Context and Affordances". In: *International Journal of Advanced Computer Science and Applications* (2011). URL: https://arxiv.org/abs/1208.3323.

[17] Pabst, S. et al. "Seams and Bending in Cloth Simulation". In: *Workshop in Virtual Reality Interactions and Physical Simulation, VRIPHYS* (2008). URL: https://diglib.eg.org/server/api/core/bitstreams/c247cf86-304e-4d75-8805-72d40765e761/content.

[18] Provot, X. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth". In: *Institut National de Recherche en Informatique et Automatique (INRIA)* (2001). URL: cs.rpi.edu.

[19] Rutherford, A. "Here's How To Fix Cloth Physics For Cinematics With Unreal Engine 5". In: *80 Level* (2024). URL: https://80.lv/articles/here-s-how-to-fix-cloth-physics-for-cinematics-with-unreal-engine-5.

[20] Sung, N.-J. et al. "Real-Time Cloth Simulation Using WebGPU: Evaluating Limits of High-Resolution". In: *arXiv* (2025). URL: https://arxiv.org/abs/2507.11794.

[21] Unreal Engine Software. Accessed June 2025. 2025. URL: https://www.unrealengine.com/en-US/unreal-engine-5.

[22] UE Documentation Coding Standard. *Coding Standard*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/epic-cplusplus-coding-standard-for-unreal-engine.

[23] Unreal Engine Community Forum. Accessed July 2025. 2020. URL: https://forums.unrealengine.com/t/marvelous-designer-material-appears-differently-in-unreal-engine-usd-import-chaos-cloth-asset/2223232.

[24] Unreal Engine Community Forum. Accessed July 2025. 2020. URL: https://forums.unrealengine.com/t/editor-and-runtime-versions-of-the-same-plugin-library/140849/10.

[25] UE Documentation Geometry Scripting. *Geometry Scripting User Guide*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/geometry-scripting-users-guide-in-unreal-engine.

[26] UE Documentation Chaos Cloth Overview. *Chaos Cloth Tool Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/clothing-tool-in-unreal-engine.

[27] UE Documentation Physics Manual. *Physics in Unreal Engine*. Accessed June 2025. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/physics-in-unreal-engine.

[28] UE Documentation Modules. *Unreal Engine Modules Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-modules.

[29] UE Documentation Objects. *Objects*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/objects-in-unreal-engine.

[30] UE Documentation Properties. *Properties*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-uproperties.

[31] UE Documentation Referencing Assets. *Referencing Assets*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/referencing-assets-in-unreal-engine.

[32] UE Documentation Slate. *Slate Overview*. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/slate-overview-for-unreal-engine.

[33] Wong, E. "User Interface Design Guidelines: 10 Rules of Thumb". In: *Interaction Design Foundation* (2025). URL: https://www.interaction-design.org/literature/article/user-interface-design-guidelines-10-rules-of-thumb?srsltid=AfmBOoo7555Z0YqJ92KXlT6zAUWqabOQ9rLc32enAHO8VaPOfTidofvX&.

[34] Zhang, D. et al. "Physics-inspired Estimation of Optimal Cloth Mesh Resolution". In: *Siggraph Conference Papers 25* (2025). URL: https://dl.acm.org/doi/10.1145/3721238.3730619.

[35] Zhao, Z. "A Physics-embedded Deep Learning Framework for Cloth Simulation". In: *arXiv* (2024). URL: https://arxiv.org/pdf/2403.12820.