# [TexDoubleRaschel 시뮬레이션 문서]

작성자 : 김태형
작성일 : 2018.11.12
최종 수정일 : 2019.01.28

# 0. 머리말

기존 C++ Builder에서 Visual Studio로 마이그레이션이 된 DoubleRaschel(구 Textricot 4.0)를 처음 인수인계 받았을 때, 시뮬레이션 파트는 DirectX를 활용해 프론트와 백은 3D로 만든 뒤 2D 이미지로 캡처해서 놓고, 파일사는 없는 형태의 미완성 프로젝트였다.

> "미치셨습니까 휴먼? 마무리는 어떻게 된 것입니까?"

DoubleRaschel은 구성이 잘 짜여진 프로젝트였으나 다음과 같은 문제가 존재했다.

> - 두 달 정도 남은 시간 동안 프로젝트가 완성이 되어야 하는데 Direct X를 사용해서 개발하기에는 시간이 부족했다.
> - 사내에 DirectX를 사용할 줄 아는 사람이 없었다.
> - 심지어 DirectX Native가 아닌 SharpDX라는 라이브러리를 사용하고 있었다.
> - C++/CLI와 C#이 섞여 있어 C++/CLI를 새로 공부해야 했다.

때문에 필자는 기존 'SharpDX' 라이브러리로 제작했던 시뮬레이션 부분을 삭제하고, Unity로 한 번 더 마이그레이션을 했다. 우선 Unity는 C#을 기반으로 한 스크립트를 지원하기 때문에 C++/CLI를 공부할 필요성을 없앴으며, 동시에 DirectX를 몰라도 Unity만 할 줄 알면 유지보수가 가능한 장점이 있다. 혹자는 사내에 Unity를 제대로 다룰 수 있는 사람이 필자밖에 없다고 하였으나, 추후에 인력 보강을 할 경우 DirectX 사용자보다 Unity 사용자를 구하는 것이 훨씬 쉽다는 것에 동의하였다.
이 문서에서는 기존 DoubleRaschel 인수인계 문서에서 기술된 기초적인 이론 그리고 기존 프로젝트에서 어떻게 마이그레이션이 됐는지, 어떤 식으로 변경되었는지 그리고 클래스 등에 대해서 최대한 자세하게 작성하려고 노력했다. 기존 인수인계 문서를 받았을 때 'Octopus'라는 변수명이 도대체 무엇인지 몰라 몇 번이고 코드를 갈아엎었던 기억이 있기에 다음에 이 문서를 보게 될 사람은 그러한 수고를 덜기 위해 최대한 자세하게 설명하려고 했다.
추가적으로 간단하게 자카드가 구현되어 (있는 것처럼 되어) 있다. 별다른 것은 없고 트리코트 위에 그림을 하나 얹었다고 생각하면 이해하기 쉽다. 실제로는 이렇게 구현하는 것이 아니지만, 시간 부족 이전에 기획이 이런

식으로 기획되어 있었기 때문에 추후 수정이 진행될 수도 있다.
참고로 이 프로젝트를 전달받았을 때, 시뮬레이션 부분에 대해서 아는 사람이 사내에 없었을 뿐더러 인수인계 받은 프로젝트 파일에 주석이 제대로 달려있지 않은 것이 많고 작성된 문서 또한 빈약하기에 필자의 추측이 어쩔 수 없이 들어가있다. 추측한 부분의 경우 추측했다고 작성할테니, 혹여 틀릴 경우 이 문서를 수정해줬으면 한다.
앞으로 이 프로그램이 얼마나 더 발전이 될지는 모르겠으나, 이 프로그램을 완성시킨 사람으로서 망하지 않길 바라며 글을 마치겠다.

2019년 1월 기준으로 후가공을 개발했다.

# 1. 트리코트와 더블라셀에 대하여

## 1-1. 트리코트란?

트리코트란 경편물의 일종으로 수영복이나 카시트 등을 생산하는데 주로 사용된다. 빠른 속도로 뽑아내는 트리코트용 기계를 사용한다

## 1-2. 더블라셀이란?

더블라셀이란 경편물의 일종으로 카시트나 신발을 생산하는데 주로 사용된다. 트리코트보다 느린속도로 뽑아내는 라셀용 기계를 사용하지만, 가이드바가 뒷면 및 파일사를 생성하는데 들어가 라셀과는 다른 설계방식을 갖는다.

## 1-3. 직물의 분류

```
섬유 원단 - 직물
          - 편물 - 위편
                 - 경편 - 트리코트
                        - 라셀
                        - 더블라셀
                        - 그 외
```

# 2. 트리코트의 설계

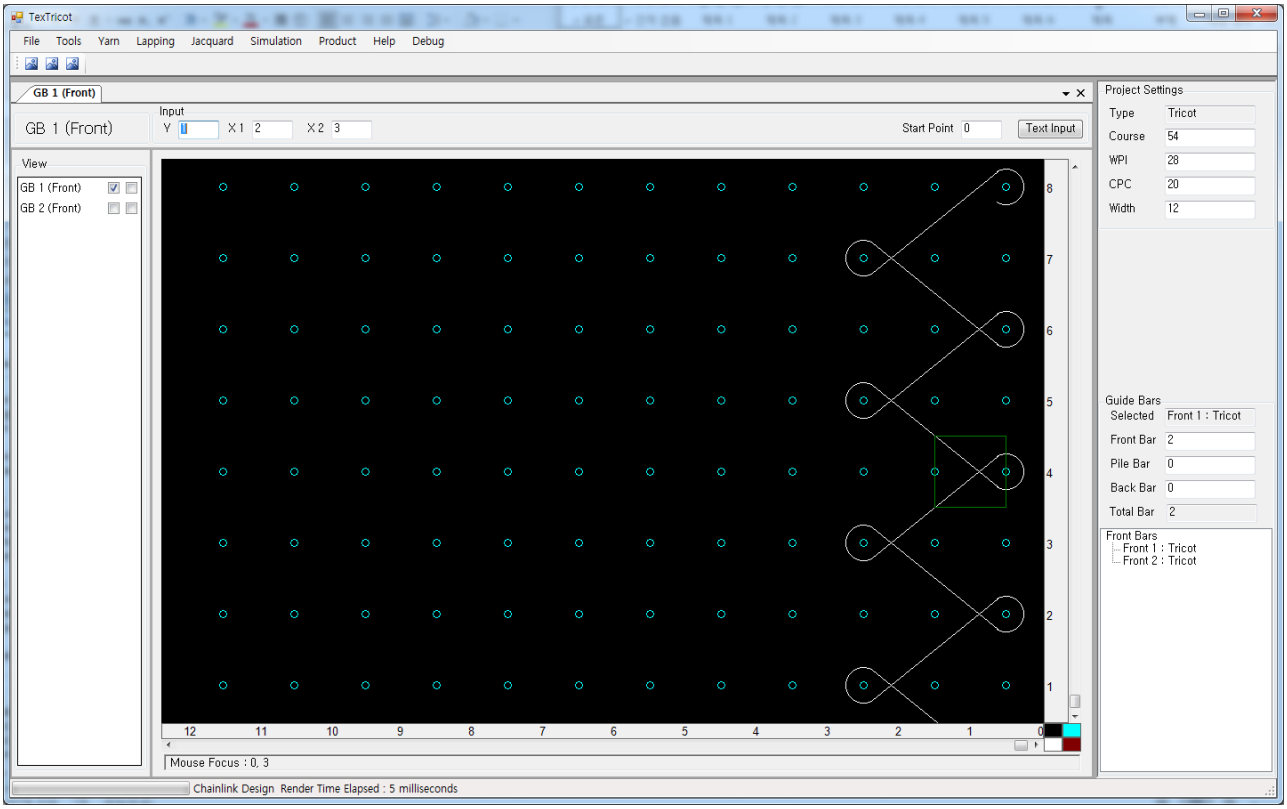
트리코트 설계를 이해하기 위해서는 기본적으로 가이드바와 체인링크에 대해 이해해야 한다. 포토샵을 생각해볼 때 가이드바란 하나의 레이어에 해당하며, 체인링크는 해당 레이어에 대응되는 그림 정보라고 생각하면 된다. 수편기를 떠올리면 된다. 자세한 건 아래에 설명해 두었다.

## 2-1. 가이드바

가이드바는 실바늘에 실을 걸어 스킬 바늘이 실을 엮어갈 수 있게끔 좌/우로 이동하는 가로로 긴 장치이다. 가로로 길게 실바늘이 무수히 늘어져 있으며, 그 실바늘마다 실을 넣고, 동시에 좌/우로 움직이게 한다. 그렇게 움직이는 도중, 스킬 바늘이 이 실을 엮어가는 방식이다.
트리코트기계의 국내 트리코트의 경우 현재 7개의 가이드바가 있는 기계가 있는 것이 확인되었으며, 이 가이드바가 많아질수록 설계가 다양하고 복잡해지게 된다. 기존 TexTricot3.2는 '가이드바 = 레이어'의 개념을 적용하였다.
트리코트 가이드바의 경우 Threading 정보를 설정할 수 있다. 이는 어느 실바늘에 실을 거는지에 대한 정보이다.

### 2-2. 체인링크

체인링크는 트리코트 가이드바가 좌/우로 이동하는 정보이다. 따라서 트리코트 가이드바의 경우, 단 하나의 체인링크 정보를 갖는다.
체인링크는 코스라는 정보로 이루어져 있다. 코스는 X1에서 실이 들어가고, X2로 실이 통과하는 정보이다.




위의 래핑정보는 (23 10) * 3 으로 아래서부터 x축으로 2번위치로 실이 올라가고 3번위치로 내려온 뒤 다음 코스에서 1번 위치로 실이 올라가고 0번위치로 실이 내려오는 것을 볼 수 있다. 따라서 23 10으로 쓰고 이것이 3번 반복되었으므로 (23 10) * 3으로 적는다. 이때 2는 첫번째 코스의 X1이고 3은 첫번째 코스의 X2, 1은 두번째 코스의 X1, 0은 두번째 코스의 X2로 읽는다. 3 Lapping 형태로 표현될 경우 해당 코스가 나가는 방향도 정해야 한다. 이는 다음 코스의 X1 정보를 이용해야 한다.

## 3. 더블라셀의 설계

더블라셀은 트리코트에 비해 느린 라셀 기계를 사용해 만들어진다. 가이드바가 7개인 7바 라셀기계에서 1번과 2번바를 이용해 앞면을, 3번과 4번바를 이용해 파일사를, 7번바를 이용해 뒷면을 짜게 시킬 수 있다. 앞면 / 파일사 / 뒷면의 순서는 바뀔 수 없으나 각 면에 사용되는 가이드바의 수는 어느정도 바뀔 수 있다. 하지만 각 기계마다 바늘이 접근 가능한 가이드바의 번호 최대치가 다르거나 파일바의 한계로 인해 설계가 불가능한 설계도면도 나올 수 있다.
트리코트와 같이 체인링크를 이용해 설계한다 (라셀 및 라셀 자카드는 이미지에 그리드를 그려 설계하는 것으로 알고있다). 이때, 뒷면 및 파일사는 단순한 체인링크가 사용되어왔으나, 이 설계 프로그램으로 더 다양한 설계를 할 수 있어야 한다.

## 4. 기타로 이걸 받는 사람에게 이야기해주고 싶은 것

위에 간략한 이론들은 필자도 받았던 내용이라서 그냥 알아만 두라고 복붙해놨다. 실질적으로 밑에 내용들은 입력도 안되어있었고 더블라셀 프로젝트에서는 구현조차 안되어있었기 때문에 필자가 구현하면서 이건 좀 알아야겠다 싶은 내용들을 적어놨다. 개발하는 데에 있어서 조금이나마 도움이 될 수 있었으면 좋겠다.

# 4-1. 후가공

하... 필자가 구현하면서 제일 골치 아팠던 것 중에 하나다. Textricot에서 후가공이라고 하면 여러가지가 있는데, 보통 회사에서 후가공 이라는 이야기가 나온다면 그것은 '샤링(Shearing)' 혹은 '루프기모(Raising 혹은 Loop nap. Raise와는 다르다!)' 라고 알아들으면 된다.

## 4-1-1. 샤링

샤링은 동그랗게 말린 Head를 잘라서 '뽀실뽀실한 효과'를 준다고 생각하면 된다(디자인팀 왈). 개발상으로는 Head를 자르고 여러개로 퍼뜨린 기능이라고 보면 된다.


(이미지에서 보이듯 머리를 자르고, 실이 여러개일 경우 실이 풀어지는 효과는 덤)

참고로 샤링 샘플은 트리코트 내에 있는 st-446-1 이나 st-7302 를 보면 알 수 있다.

## 4-1-2. 루프기모

이걸 왜 Raising이라고 해놔서 Raise와 헷갈리게 용어를 적어놨는지 모르겠다. 나는 그래서 함수명으로 Loop_Nap이라고 써놨다. 루프기모란 Head를 좀 길게 늘어뜨려서 이태리타올? 같은 효과를 준 것이라고 보면 될 것 같다.


(이해를 돕기 위해서 실제 홈쇼핑에서 표현한 루프기모 이미지를 가져왔다)

참고로 루프기모 샘플은 트리코트 내에 있는 tb-01 이다.

# 4-2. Filament와 Denier

더블라셀의 Yarn List를 보면 Filament와 Denier를 볼 수 있다. 실이란 하나의 실 자체로도 만들 수 있지만, a라는 실을 n개를 꼬아서 하나의 b라는 실로 만드는 경우가 있다. Filament는 여기서 n을 이야기하는 것이고 Denier는

전체의 두께이다. 예를 들어서 35 Filament에 75 Denier와 1 Filament 75 Denier는 같은 굵기다! 단순히 몇 개의 실이 엮여서 몇 Denier의 실로 만드느냐로 이해하면 된다. 이 데이터는 더블라셀의 DRMath.cs 내에 실질적인 실 굵기를 표현해준다.

# 5. Unity로 마이그레이션 된 프로젝트 설명

## 5-1. Unity 버전

Unity 2018.1.1f1로 마이그레이션을 진행했고, 작성일 기준 2018.2.4f1로 버전 업그레이드를 진행했다.

## 5-2. 폴더 구조

Unity로 마이그레이션 된 프로젝트의 폴더 구조는 다음과 같다

1. Fonts
    - UI에 들어갈 폰트가 들어있는 폴더
2. Material
    - 오브젝트에 붙일 마테리얼이 들어있는 폴더.
3. Prefabs
    - HEAD, LEG, SLEG: 실을 각 부분으로 나눠서 그린다.
    - Octopus: 실이 엮이는 곳. HEAD, LEG, SLEG를 포함한다(상속 X). 아래 기술할 Octopus Class에서 이 오브젝트에 대해서 설명할 것이다.
    - Jacquard: 자카드 그림을 올리기 위해 쓰는 판.
4. Resources
    - 프로젝트 상에서는 설계 데이터, 시뮬레이션 실행 시 스크린 샷 파일이 들어있는 폴더
    - 실제 빌드 시에는 삭제해도 무방함.
    - Tricot : 트리코트 샘플을 넣어놓은 폴더. 테스트를 위해 만들어서 넣어놓았다.
5. Scene
    - 프로젝트 씬 파일이 들어있는 폴더
6. Scripts
    - Core: 시뮬레이션을 하는데 필요한 코드들을 모아 놓은 폴더
    - Manager: 프로그램 전체를 관리하는 매니저 클래스, 데이터 파싱 클래스, UI 관리 매니저 클래스가 있는 폴더
    - Treatment : 후가공을 위해서 넣어놓은 클래스. 사실 더블라셀에서 파일사를 잘라서 파일이 널부러지는 효과를 제작하기 위해서 만들어놓은 클래스인데 쓰이진 않는다. 일단 잘라서 파일사가 퍼지는 효과는 구현을 해놓았다.
    - UI: 트리코트, 더블라셀, 자카드에 따라 바뀌는 UI들에 대한 기능들을 작성한 폴더
7. Sprites
    - 아이콘과 Box 스프라이트가 들어있는 폴더
    - 첨언하자면, 디자인을 작성자가 직접 했기 때문에 사각형으로 모든 걸 처리해야 했다.

## 5-3. 시뮬레이션 UI 구조


1. 시뮬레이션 버튼
2. 캡처 버튼(ScreenShot 폴더에 저장된다)
3. Background Color
    - 백그라운드 색상 변경. R,G,B 값을 조절해서 바꾼다.
4. Simulation Setting Menu
    - Mode • 2D : 2D 시점으로 카메라 변경. 카메라 이동, 확대 및 축소, 앞, 뒤 볼 수 있음. • 3D : 3D 시점으로 카메라 변경. 카메라 이동, 확대 및 축소, 앞, 뒤, 옆 등 원하는 방향으로 볼 수 있음. • Fabric : 일완전패턴을 n * m 으로 반복해서 전개한 모양. 2D와 같은 옵션
    - Yarn : Front/Back/Pile 중 원하는 면만 보여주거나 끌 수 있다.
    - Width : 편물의 폭.
    - CPC : 코스 밀도.
    - Fuzz : 흩뜨림.
    - Optimize : 실의 수축작용 계산 여부.
    - Pile Leg : 파일사 다리 그릴지 말지 여부.
5. Interface
    - 프로젝트 타입에 따라서 인터페이스 달라짐.

2019년 1월 기준 UI가 소폭 변경되었다.

1. Pile Leg 기능이 사라짐
2. Fabic 설정 시 기존에는 정해진 값으로 전개하는 방식에서 x, y 값을 입력 후 Fabic을 누르면 x * y 로 전개하는 방식으로 변경
3. Interface 창이 사라지고 Treatment창이 생겼다. 트리코트는 비어있고, 더블라셀은 파일사 가마 높이를 수정할 수 있도록 기능을 만들었다.

## 5-4 스크립트 설명

스크립트는 전체적인 프로그램을 관리하는 Manager 클래스와 시뮬레이션 쪽을 구현한 Core 부분, UI를 관리하는 UI 클래스로 분리했다.

### 5-4-1 Core

#### 1) Common.cs

공통적으로 참조하는 Enum 타입이나 구조체를 묶어 놓은 파일이다.

- **Common Class**
  - DOT_PER_CM : 해상도. 실험결과 1:1에서 프린트할때 가공지와 같게 나오는 값이라고 한다.
  - COUNT_REPEAT_X, COUNT_REPEAT_Y : Fabric 모드에서 M * n 형태로 전개하기 위해서 받는 변수.
  - PILE_HEIGHT : 파일사의 높이(가마 높이)를 변경하기 위해서 사용하는 변수
  - PILE_CORRECTION_FACTOR : 파일사 높이 값 수정하기 위한 보정 변수, PILE_HEIGHT가 실질적으로 높이를 변경해주는 변수라면 이 변수는 시뮬레이션 내부에서 실제와 비슷하게 맞춰주기 위한 변수라고 보면 된다. 앞으로 적힐 CORRECTION_FACTOR 시리즈는 다 이러한 용도다.
  - IS_ONE_LOOP : Fabric 모드인지 아닌지 체크하기 위한 Bool 타입 변수. 지금 당장 쓰이진 않고 나중에 추가될 ONE_LOOP 모드에서 사용될 변수
  - COUNT_SHEARING : 후가공 중 샤링(Shearing) 효과를 사용할 때, Head가 몇 가닥으로 갈라져서 표현될 것인지를 나타내는 Int형 변수. n^2 형태로 나타내야 이쁘게 나온다.
    사실 실의 데이터에서 filament 개수만큼 흩뜨려야해서 실제 filament보다 값이 클 때도 있고 작을 때도 있을 것이다. 그런데 이 정도 값이 사실 최적화와 타협한 값이라서 웬만하면 이 값으로 쓰는 게 모양도 안이상하고 이쁘게 나올 것이다. 추후에 다른 개발방식으로 최적화가 된다면 그 때는 값을 맞추는 것이 좋을 듯 하다.
  - CORRECTION_FACTOR : 실 굵기를 보정하기 위한 보정 상수
  - TRANSPARENT_CORRECTION_FACTOR : 투명값을 보정하기 위한 보정 상수
- **POINT** : x,y값을 가진 구조체. Vector2를 써도 무방하나, 코드 이식상 편리를 위해 POINT 구조체를 가져왔다.
- **HEAD_TYPE** : 코가 걸리는 모양에 따라서 나뉜다. L_HEAD의 경우 왼쪽으로 열린 코, R_HEAD는 오른쪽, O_HEAD는 열리지 않고 닫힌 코다.
  (참고로 열린 것(Open)과 닫힌 것(Close)의 개념을 모르면 설계창에서 코를 누를 때마다 열리거나 닫히는 경우를 볼 수 있는데, 그걸 보면 이해하기 편하다)
- **LEG_TYPE** : LEG의 진입과 진출 위치를 표현했다. 예를 들어 LL은 왼쪽에서 들어와서 왼쪽으로 나가는 것이다(Open과 Close에 무관).
- **LEG_STATE** : [추측] 레이어의 제일 처음 혹은 제일 마지막에 해당하는 다리가 위로 올라오거나 아래로 내려가야하는 경우를 체크하기 위해서 만든 것 같다.
  (여러가지 샘플을 돌려보았으나 NORMAL_LEG 이외에는 사용되는 것을 본 적이 없다.)
- **SLEG_TYPE** : Side LEG의 진입과 진출의 위치를 나타낸다.

#### 2) DRMath.cs

수학적 계산을 모아놓은 파일이다.

- **CalculatePentaBezierPoint()** : 점 **다섯 개**를 베지어 곡선 공식으로 보간해서 0 ~ 1사이의 좌표를 리턴하는 함수. 사실 4개 초과부터는 최적화와 관련해서 좋지 못하기 때문에 잘 안쓰는 것이 맞다. 근데 이거 안쓰면 Head가 이쁜 모양으로 나오지 못해서 어거지로 만들었다.
- **CalculateCubicBezierPoint()** : 점 **네 개**를 베지어 곡선 공식으로 보간해서 0 ~ 1사이의 좌표를 리턴하는 함수
- **CalculateQuadraticBezierPoint** : 점 **세 개**를 베지어 곡선 공식으로 보간해서 0 ~ 1사이의 좌표를 리턴하는 함수

- **DenierToDiameter()** : 실의 굵기를 계산해주는 함수. 여기서 Filament와 Denier가 사용된다.
- **lcd()** : 최소공배수
- **gcd()** : 최대공약수

#### 3) Project.cs

프로젝트 세팅값을 저장하는 클래스를 모아놓은 파일이다.

- **ProjectType** : 프로젝트의 종류에 따라 트리코트, 더블라셀, 자카드로 분류해주는 enum 값
- **Environment** : 프로젝트 종류와 course, wpi 등 프로젝트 세팅값을 저장하는 클래스
- **SimulationInfo** : [추측] 시뮬레이션 상에서 비율을 잡기 위해서 만들어놓은 클래스
- **Project** : 프로젝트 환경, 실의 종류, 체인링크, 가이드 바 등 데이터를 가지고 있는 클래스. 현재 시뮬레이션을 돌리는 프로젝트의 모든 설정 값을 가지고 있다고 보면 된다

#### 4) GuideBar.cs

가이드바에 관련된 클래스와 구조체를 모아놓은 파일이다.

- **BarFace** : Front/Back/Pile에 따라 그릴 Bar를 결정해주는 enum변수. 참고로 FaceMax는 그냥 BarFace 갯수를 체크하기 위해서 만들어놓은 거 같다. 필요는 없는데 그냥 3이라고 쓰는 거보다 낫지 않겠나 싶어서 놔뒀다.
- **Threading** : 실의 In/Out 데이터를 저장하는 클래스.
  - type : In인지 Out인지 체크
  - count : 몇 번 In인지 혹은 몇 번 Out인지
  - yarnIndex : 실 종류 중에 어떤 실을 쓸 건지
  - transparent : 실을 투명하게 보이게 할 건지 체크. 이거 기능 구현 안해놓음. 기능 구현 필요하면 해당 Material에 색깔 넣어줄 때 알파값만 맞춰서 변경해주면 된다.
- **Bar** : 프로젝트의 각 Bar 마다 필요한 데이터를 저장하는 클래스.
  - SetBarFace : 바페이스 설정
- **GuideBar** : BarList를 가지고 있는 클래스이며, 각 Bar들을 관리한다.
  - GuideBar() : 생성자. front/back/pile 바 리스트를 초기화해준다.
  - GetBarList() : 특정 바 페이스의 리스트를 가져오기 위한 함수
  - GetDrawEntry() : 특정 DrawEntry 리스트에 있는 특정 DrawEntry를 가져오기 위한 함수.
  - GetDrawEntries() : 특정 DrawEntry 리스트를 가져오기 위한 함수.
  - OrderingBar() : DrawEntryList를 세팅해주는 함수. Octopus가 Front, Back으로 존재하기 때문에 Pile사의 경우, 앞면의 경우 Front에, 뒷면의 경우 Back에 함께 그려줘야하기 때문에 이 함수를 이용해서 Pile사를 적절히 배치해준다.
  - Clear() : 데이터 삭제를 위한 함수

#### 5) Chainlink.cs

체인링크에 필요한 데이터들을 모아놓은 파일이다.

- **Course** : 설계창에서 23, 31 같은 데이터가 파싱되어 들어갈 구조체다.
- **Chainlink** : 체인링크에 대한 데이터를 모아놓은 클래스다.
  - GetCourseClamp() : ClipX, ClipY와 같이 인덱스가 넘어갈 경우 넘어간 값만큼 빼고 난 인덱스 값을 가져온다.
- **Chainlinks** : 체인링크들을 관리하는 클래스다. 체인링크 리스트를 가지고 있다.

#### 6) Yarn.cs

Yarn은 실이다. 본래 Tricot에는 여러가지 실의 효과를 줄 수 있는 Yarn Table이 존재하나 더블라셀로 마이그레이션 당시에 여러가지 이유로 Yarn Table을 이식할 수 없다고 했다. 그래서 코드 내에 YarnMaterial이나 YarnType은 사실상 쓸모가 없다. 그럼에도 남겨놓은 이유는 혹여 이 글을 읽는 당신이나 다른 사람이 개발할 수도 있기 때문에 굳이 지우지 않았다.
실제로 실은 그냥 Head나 Leg에서 마테리얼을 씌우거나 SimulationRender에서 실의 굵기를 계산해서 넣어버린다.

#### 7) Octopus.cs

Octopus 객체는 쉽게 이야기하면, 설계창에서 점과 점 사이를 직선으로 그어 사각형으로 나눴을 때 그 사각형 하나가 Octopus다 라고 보면 이해하기 쉽다.
필자가 개발 당시에 Octopus라는 이름을 지은 이유가 무엇이냐고 Tricot를 개발했던 A씨와 같이 일했던 B씨에게 물어봤는데 머리가 동그랗고 다리가 여러 개니까 문어같이 생겼다라는 이상한 답변을 받았다. 그래서 초반에는 Head + Leg가 Octopus다 라고 생각하고 개발방향을 잡았으나 전혀 다른 이야기였다.
아무튼 좌표 하나하나에 대한 사각형은 여러 개의 Head가 걸리거나 여러 개의 다리가 지나갈 수 있기 때문에, 각 Octopus에 List로 Head, SideLeg, Leg 등을 생성해 추가한 뒤, 하나하나 그려주는 방식으로 개발했다.

- **OctopusData** : 파일사를 연결해주기 위해서 현재 Octopus와 다음 Octopus의 값을 저장하기 위한 구조체다.
- **Render_XXX()** : 각각 Head, Leg, SideLeg를 그려주는 함수다.

#### 8) HEAD.cs

코에 걸리는 둥근 모양을 Head 라고 부른다. 와 진짜 여기 개발한다고 제-일 오래 걸림.

- **Render()** : ~~Head를 오른쪽과 왼쪽으로 나눠서 그리는 구조를 사용했다. 베지어 곡선을 4차이상 늘리면 연산속도가 급격하게 저하되기 때문이다(그럼에도 5차로 계산을 하는 이유는 퀄리티 상 양보할 수가 없었다). isLeft라는 변수를 사용해서 각각 왼쪽과 오른쪽을 나눠서 그린다.~~
  19.01.28 기준 코드를 바꿨다. 원래는 후가공을 고려 안하고 개발한 내용이라 저대로 구현했는데, 고려해야할 사항이 3개로 늘었다.
  - 파일사를 기준으로 앞면이 잘리는지, 뒷면이 잘리는지, 아니면 안잘리는지
  - 루프기모인지
  - 샤링인지

그래서 결국 Render()라는 큰 함수에서는 조건만 처리해주고 Normal Head, Shearing, LoopNap 함수를 따로 만들어서 포지션만 처리해주고 CreateMesh()를 이용해서 그리는 방식으로 그렸다. 간략하게 표현하자면 다음과 같다

```
Render() ──── Render_Normal() ──── CreateMesh()
          └─ Render_Sheraing() ─┘
          └─ Render_LoopNap() ──┘

와 같은 형태라고 보면 된다.
```

- **Render_Shearing()** : 샤링을 표현하기 위해서 만든 함수. x_Factor와 y_Factor라는 변수로 각각 가로 세로로 몇 개의 머리털을 그려줄 건지 고려를 해주고, 나눈 만큼 굵기를 4분의 1로 감소해서 표현해준다. 필라멘트 수만큼 굵기를 나누지 않은 이유는 너무 얇아져서 겨드랑이 털처럼 보인다는 디자인팀의 전언때문이다.
- **Render_LoopNap()** : 루프기모를 표현하기 위해서 만든 함수. 기존 Render() 함수에서 포지셔닝만 살짝 변경해주고 굵기를 3분의 1로 줄였다. 트리코트 굵기에 맞춰서 똑같이 그려주니 루프같은 느낌이 없다고 해서 저렇게 표현했다.
- **Render_Normal()** : 그 외에 기본적인 Head를 그려주는 함수
- **CreateMesh()** : 넘겨받은 포지션으로 베지어 곡선 공식을 이용해 3D Mesh를 직접 만들어서 그려주는 함수다.

#### 9) LEG.cs

코에 걸리는 둥근 모양을 제외한 나머지 선들을 Leg라고 표현한다.

- **Initialize()** : Yarn을 세팅하고 메쉬 생성을 호출해주는 초기화 함수
- **CreateMesh()** : Head와 마찬가지로 베지어곡선으로 포지션을 잡고 3D Mesh를 직접 만들어서 그려준다. 여기서 파일이냐 아니냐로 나눈 이유는 upNormal값을 바꿔줘야함이 크다. 파일사의 경우 upNormal의 값을 (0,0,-1)로 잡게되면 정말 가느다란 실처럼 표현이 되기 때문에 y값을 -1로 바꿔주어야 제대로 생성이 된다. 또한 Leg마다 y값에 랜덤값을 주어 실이 휘어지는 효과를 줬다.

#### 10) MLEG.cs

MirrorLEG. 오직 optimize processing 에서만 쓰인다. 일반 LEG의 미러 이미지이며 반대 방향의 힘을 계산하기 위한 임시 데이터다.

#### 11) SLEG.cs

Side Leg. octopus에 고정되지 않고 움직이는 다리다. 코드는 LEG와 동일함.

#### 12) OctopusSet.cs

Octopus 생성 및 실제 위치 계산, 장력 계산 등이 들어가있는 클래스다. 멘붕이 올 수도 있으니 심호흡 크게 한 번 들이마시고 읽길 권장한다. 필자도 처음에 그랬다.

- **Initialize()** : 개수에 맞게 옥토퍼스 생성을 해주는 함수다. 여기서 Octopus의 포지션은 잡지 않고 가로 * 세로의 개수만큼 생성한다.
- **Clear()** : New Simulation 기능을 사용할 때 초기화를 해주고 새로 그려줘야하기 때문에 초기화를 해주는 기능을 만들었다.
- **ClipX(), ClipY(), ClipXY()** : 프로젝트 특성상 n * m 으로 반복을 하는 경우가 많은데, 리스트의 인덱스를 넘어가면 OutofIndex에러가 발생하기 때문에, 발생하지 않도록 이 함수를 만들어서 인덱스를 참조할 때마다 호출하도록 했다.
- **GetOctopus()** : 2차원 좌표로 배치되어 있는Ocotpus를 x, y를 각각 사용하거나 POINT 좌표로 가져올 수 있다.
- **FindRealPosX(), FindRealPosY()** : [추측] 실제 X와 Y의 좌표를 가져온다. Width나 cpc 등으로 달라지는 옥토퍼스들의 실제 포지션 계산을 해줘야 정확한 계산이 나오기 때문에 만든 함수가 아닌가 싶다.
- **FindMidPoint_X(), FindMidPoint_Y()** : [추측] 각각의 실이 잡아당기는 힘을 계산해서, 실들이 교차된 한 점의 포지션을 구하는 함수다.

#### 13) Simulation.cs

[축하한다. 이제 여기가 헬 구간이다. 잘 넘어가길 바란다.] 렌더링을 제외한 전체적인 시뮬레이션 처리가 이루어지는 곳이다. 데이터를 불러오는 것부터 시작해서 Initialize, Octopus 위치 세팅, 계산 등 사실상의 매니저 클래스 역할이다.

##### 13-1) SimulationContext

시뮬레이션 전처리 코드라고 보면 쉽다. 어느 정도 비율로 그릴거고, 원리핏 개수를 설정하고 Bar 세팅 등을 하는 클래스라고 보면 된다.

- **SimulationContext()** : 시뮬레이션을 실행하기 전에 각종 정보들에 대해서 세팅하는 작업을 해준다.
- **Set_RepeatCount()** : 사실 앞서 이야기했던 Common.COUNT_REPEAT_X 와 Common.COUNT_REPEAT_Y를 전역변수로 생성해버려서 이 함수의 존재 이유가 사라지기는 했으나, 일단 있어보이게 놔둬놨다. 관리가 조금이라도 편하겠다라는 정신승리와 함께

##### 13-2) Simulation

정말 헬구간. 정신을 놓지 말고 하나하나 따라가다보면 그래도 길이 보일 것이다 힘내자 화이팅!

여기는 시뮬레이션에 대해서 직접적으로 그리는 Rendering을 제외하고는 모든 것을 담당한다. 그래서 다른 클래스에 비해서 길이가 아아아아아주 길다. 사실 분리를 할까 했다가 그게 더 가독성을 잃어버릴 듯 해서 관뒀다.

- **InitializeData()** : 제일 처음 시뮬레이션을 실행할 때 데이터를 읽어오는 작업을 진행 후 시뮬레이션을 돌린다. 이 함수는 최초 1회만 실행된다.
- **StartSimulation()** : 읽어온 프로젝트가 트리코트인지, 더블라셀인지, 자카드인지에 따라서 OctopusSet을 각각 세팅해주고 그려주는(Optimize(), Render()) 함수들을 호출한다.
  참고로 이 함수 내에서 시뮬레이션 관리 오브젝트 Scale을 (1, 1, 1)로 만들었다가 시뮬레이션이 끝나면 (-1, 1, 1)로 만드는데 이것은 시뮬레이션이 설계창과 반대로 그려지기 때문에 이를 보정하기 위해서 이렇게 코드를 구현했다.
- **InitOctopusData()** : Octopus 데이터들을 초기화해주는데, 현재 코스와 다음 코스가 어떻게 진행이 되느냐에 따라서 세 가지로 나누어서 진행된다. 각각의 case 들에 대해 소스 코드 내에 주석으로 달아놓았는데
  - case 1 : 처음부터 끝까지 모든 코스가 x1 == x2 같은 방식으로 x1과 x2가 같은 코스가 쭈욱 반복되는 케이스다.
  - case 2 : 현재 코스는 x1 != x2이지만 다음 코스가 x1 == x2 일 경우다.
  - case 3 : 모든 코스가 x1 != x2인 경우다.

all_linear 변수나 straight 변수가 저 케이스를 체크하기 위해서 만든 변수이니 너무 깊게 신경쓸 필요는 없다.

참고로

```
// calc repeat info
   int nold_iotc = 0;
   List<Threading> threadInfo = new List<Threading>();

   for (int j = 0; j < curBar.threadingList.Count; j++)
   {
```

```
        nold_iotc += curBar.threadingList[j].count;
        for (int k = 0; k < curBar.threadingList[j].count; k++)
            threadInfo.Add(curBar.threadingList[j]);
    }

    // default complete needle이 아닌 유저가 임의적으로 설정한 값을 사용할 경우
    // 설정한 값으로 나눠주고 실제 threading 카운트로 한 번 더 나눠줘야
    // 시뮬레이션이 정상적으로 출력된다.
    int temp = 0;
    if (barFace == BarFace.FaceFront) temp =
project.guidebar.complete_needle_front;
    else if (barFace == BarFace.FaceBack) temp =
project.guidebar.complete_needle_back;

    for (int j = 0; j < context.arrWidth[(int)barFace]; j++)
    {
        int nData = j % temp;
        nData %= nold_iotc;
        ...
    }
```

이 부분을 이렇게 짠 이유는 디폴트로 설정된 Complete needle 카운트가 아닌 유저 임의적으로 설정한 카운트로 리핏을 전개할 경우를 위해서 짰다.

또한 이 코드에서는 파일사를 먼저 그리고 Front나 Back을 나중에 그리는 방식으로 제작했다는 것을 기억하는 것도 후에 이 코드를 이해하는 것에 도움이 많이 된다.

- **GetOctopusSet()** : BarFace에 따라 OctopusSet을 가져오기 위해 만든 함수. 의외로 쓰는 곳이 있어서 가독성 측면에서 좋다라는 판단을 내리고 내뒀다.
- **InitChainLinear()** : 앞서 설명했듯이 모든 코스에서 x1 == x2일 때 들어오는 함수다. 구현부의 경우에는 예전에 트리코트 개발 구현부를 그대로 가져왔기 때문에 유심히 볼 필요는 없으나 심심하면 한번 분석해 보도록 하자.
- **InitChainNormalAndUpLinear()** : 앞서 설명했듯이 현재 코스는 x1 != x2이지만 다음 코스가 x1 == x2 일 때 들어오는 함수다. 솔직히 InitChainLinear() 로 들어가는 샘플을 본 적이 없기도 하고, 웬만하면 안나올 것 같다. SLEG가 생성된다면 사실상 이 함수에서 생성하는 것이니 코드 체크 시에 이 함수를 보는 것이 좋다.

  참고로 이 함수와 다음에 소개할 InitChainNormalAndNormal() 함수에는 더블라셀의 파일을 잘라서 처리하는 효과와 샤링 효과 등 후가공에 대한 전처리를 해주기 때문에 Head나 Leg를 For문을 돌려서 여러 개 생성해주는 코드가 입력되어 있다. 후가공을 제외한다면 굳이 신경쓰지 않아도 된다.
- **InitChainNormalAndNormal()** : 앞서 설명했듯이 현재 코스도 x1 != x2 이고 다음 코스도 x1 != x2일 때 들어오는 함수다. 필자 경험 상 여태까지 많은 샘플들을 테스트해보면 대부분 이 함수에서 걸린다. 그래서 많은 조건 처리가 되어있는 건 함정. 전체적으로 크게 차이가 나지는 않는다.
- **GetNextOctopusKetX()** : ~~이거 왜 만들었더라...~~ 파일사 다리를 앞 뒤로 연결해줄 때 포지셔닝을 잡기 위해서 썼다. 다음 파일사의 위치를 그릴 때 포지션을 받기 위해서 썼다고 보면 된다.

- **Optimize()** : 원사가 당겨지는 효과를 만들어주는 함수다. UI에서 Optimize를 On 하면 이 함수 내부가 돌아간다. 전체적으로 Octopus의 포지션을 실제 효과에 맞춰서 재구성해주는 듯 하다.
- **Before_Optimize()** : [추측] Optimize가 돌기 전에 전처리를 해주는 함수다. 현재 머리 다음에 다음 코가 걸리지 않는 경우 현재 코를 풀어준다든지 MLEG를 만들어서 포지셔닝을 해준다는지 그런 효과를 담당한다.
- **After_Optimize()** : [추측] 위치 결정 이후에 변경되어야할 상태를 세팅해준다. 코의 두께나 휘어짐 등을 만드는 듯 하다.
- **Render()** : 세팅이 끝난 OctopusSet들의 데이터를 가지고 실제적으로 그려주는 함수다. 별다른 내용은 없다.
- **LoadData()** : 설계 데이터를 읽어오는 함수를 호출해주는 함수
- **Show_PileLeg()** : 그냥 테스트 용으로 파일사의 다리만 On/Off 할 수 있도록 만든 함수다. 원래의 목적은 파일 다리를 없앰으로서 좀 더 넓게 원단으로 펼쳐서 보려고 테스트하는 목적이었다.
- **Clear()** : 전체적으로 생성되었던 모든 오브젝트를 초기화해준다.

##### 14) SimulationRender.cs

SimulationRender는 Simulation에서 세팅한 데이터를 기반으로 본격적으로 그려주는 클래스다.

- **LINE3D** : 그려지는 부분마다 굵기나 레이어, 포지션을 세팅하기 위해 만든 구조체. 그대로 가져왔다.
- **FindNormalVector()** : 노말벡터를 만들어준다.
- **Draw()** : 전체적으로 그려지는 것을 담당하는 함수.
- **MappingOctopus()** : 옥토퍼스마다 가지고 있는 Head나 Leg에 대해서 Layer나 시작 포지션 등을 간략하게 세팅해준다.
- **MappingXXX()** : Head, Leg, SLeg 에 대해 그려지는 포지션 세팅과 두께 등을 설정해준다.

### 5-4-2 Manager

##### 1) ProgramManager.cs

프로그램 전체를 관리하는 클래스다. 별 다른 내용은 없다.

##### 2) DataManager.cs

설계창에서 XML 형식으로 넘어오는 데이터를 파싱해서 각 클래스에 넣어주는 클래스다. 어디서든 참조할 수 있게 LoadData()를 static으로 선언했다.

한 가지 알고 넘어가면 좋은 코드에 대해서 간략하게 설명하자면

```
#region Guidebar
// Guidebar
XmlNode node_Guidebar = xmlDoc.GetElementsByTagName("guidebar")
[0].SelectSingleNode("bar_info");

project.guidebar.needle_front =
```

```
int.Parse(node_Guidebar.ChildNodes[0].Attributes[0].InnerText);
project.guidebar.needle_back =
int.Parse(node_Guidebar.ChildNodes[1].Attributes[0].InnerText);
project.guidebar.needle_pile =
int.Parse(node_Guidebar.ChildNodes[2].Attributes[0].InnerText);

if (node_Guidebar.Attributes[1].InnerText == "False")
{
    project.guidebar.complete_needle_front =
int.Parse(node_Guidebar.ChildNodes[0].Attributes[1].InnerText);
    project.guidebar.complete_needle_pile =
int.Parse(node_Guidebar.ChildNodes[1].Attributes[1].InnerText);
    project.guidebar.complete_needle_back =
int.Parse(node_Guidebar.ChildNodes[2].Attributes[1].InnerText);
}

else if (node_Guidebar.Attributes[1].InnerText == "True")
{
    project.guidebar.complete_needle_front =
int.Parse(node_Guidebar.ChildNodes[0].Attributes[2].InnerText);
    project.guidebar.complete_needle_pile =
int.Parse(node_Guidebar.ChildNodes[1].Attributes[2].InnerText);
    project.guidebar.complete_needle_back =
int.Parse(node_Guidebar.ChildNodes[2].Attributes[2].InnerText);
}
```

가이드 바 부분을 보면 `node_Guidebar.Attributes[1].InnerText` 라는 코드가 실제 XML 데이터에서는

```
<guidebar>
     <bar_info total="7" auto="True">
       <frontbar value="2" complete="8" default_complete="8" />
       <pilebar value="2" complete="8" default_complete="2" />
       <backbar value="3" complete="8" default_complete="2" />
     </bar_info>
<guidebar>
```

에서 `auto`의 값이다. 여기서 `auto`의 값이 True이면 `default_complete`의 값을 가져오고 `auto`의 값이 False라면 'complete'의 값을 가져온다. 이 데이터는 설계창에서 In/Out 코스를 설정할 때 볼 수 있는데 `default_complete`는 In/Out 코스에 맞춰서 최소 공배수로 계산이 되어 저장되는 변수이고 `complete`는 유저가 직접 설정하는 변수이다.

##### 3) UIManager.cs UI를 관리하는 클래스. 프로젝트 타입에 따라서 UI가 다르게 켜진다.
참고로 이 클래스에 있는 함수들은 Inspector창을 보면 알겠지만 직접 연결을 해줬기 때문에 스크립트 상에서 참조를 찾을 수 없다. - **Initialize()** : UI 객체들을 초기화해주는 클래스
- **Updated()** : 매번 업데이트가 이루어져야 하는 것들을 모아놓은 함수
- **Open_URL()** : 클릭 시, 회사 홈페이지로 연결해주는 함수
- **SetColor()** : 카메라 색깔 칠해줌(백그라운드 색깔 변경)
- **Capture()** : 현재 화면을 캡처해주는 함수. 지정된 경로에 폴더가 없으면 생성 후에 현재 프로젝트 이름 폴

더 안에 스크린샷이 저장된다.
- **New_Simulation()** : 이 함수를 만든 이유는 Width, CPC, Fuzz 값을 변경했을 경우 새로 시뮬레이션을 그려줘야하기 때문에 기존에 생성되었던 오브젝트들을 싹 지워주고 다시 새로 그려준다. #### 5-4-3 UI ##### 1) UIParent.cs 다형성을 사용하기 위해 만든 클래스이며, 초기화해주는 함수와 업데이트 함수를 가지고 있다.

그 외에 UI 클래스들은 함수에 각각 주석을 달아놔서 여기서 설명을 적지는 않았다.

**2) UI_Tricot.cs**

트리코트 전용 UI 클래스.

**3) UI_DoubleRaschel.cs**

더블라셀 전용 UI 클래스.

**4) UI_Jacquard.cs**

자카드 전용 UI 클래스.

# 6. 마치며

필력이 좋지 못해 좀 더 잘 설명을 해주지 못하는 것에 대해서는 미안하게 생각한다. 웬만한 변수들은 주석처리를 해놨기 때문에 여기에 적혀있지 않다고 너무 걱정할 필요는 없다. 이미 코드와 함께 보고 있기 때문에 별 상관 없겠구나 싶다.
약 3개월 간 이 프로젝트를 진행하면서 든 생각은....

뒤지기 싫으면 도망쳐라 빨리