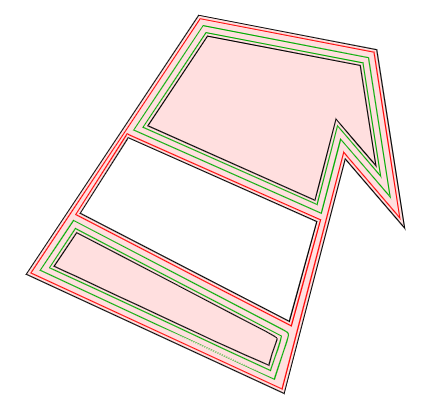
# 영역 생성

CuraEngine이 모델의 단면을 만든 후에 각 레이어를 특정 목적의 물질로 채워지는 존으로 나눈다. 이런 특징 중에 가장 중요한 것은 infill, 스킨, wall과 지원이다. 이 슬라이싱 단계의 입력은 레이어 집합이고 각 레이어에 대한 폴리곤 덩어리가 출력이다. 이들 폴리곤들은 infill, skin 등 특징에 의해 그룹핑된다.

## Wall을 분리하기

레이어를 영역으로 분리하는데 첫 단계는 wall이 되려는 파트들을 분리해내는 것이다.

결국에는 원래의 레이어 모양으로부터 나온 여러 개의 inset들이 나온다. 각 inset은 wall마다 하나이다. 아래의 이미지에서 3개의 wall이 생성된다. 하나는 빨강의 바깥 wall이고 다른 두 개는 녹색의 내부 wall이다.



첫번째 빨강의 inset은 바깥 wall을 위해 생성된다. 바깥 wall의 선 두께의 1/2의 옵셋이 있다. 결과는 바깥 wall의 중간을 관통하는 윤곽선이다. 이 윤관석의 코너들이 노즐이이 이동해야 하는 목적 좌표로서 g-code에서 마감될 것이다.

두번째 인셋(녹색)은 첫번째 내부 wall 용으로 생성된다. 이것은 첫번째 inset에 의해 윤곽이 잡힌 모양으로부터 나온 인셋이다. 인셋의 거리는 바깥 wall 선 두께의 절반에 내부 wall 선 두께의 절반을 더한 다음 또 다른 내부 wall의 선 두께의 절반이 이 첫번째 내부 wall의 중신 선에 더해진다.

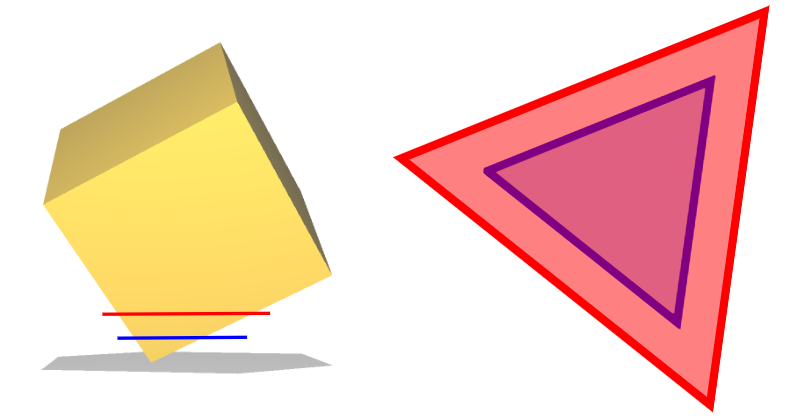
세번째 인셋과 추가적인 인셋들이 두번째 내부 wall과 그 이상을 위해 생성된다. 이것은 이전 인셋과 떨어진 한 내부 wall의 선 두께의 인셋이다. 모양은 다시 다음 내부 wall의 중간 선이다.

마지막으로 가장 내부의 wall로부터 또 하나의 인셋이 만들어진다. 이 인셋은 wall의 내부 엣지를 마킹한다. 그 모양은 skin 또는 infill로 채워져야 한다.

## Separating Skin from Infill

CuraEngine은 이 중간 인셋을 skin과 infill로 채울 필요가 있다. 이것은 skin을 어디에 놓을지 infill을 어디에 놓을지를 결정할 필요가 있다.

예를 들어 바닥 skin으로 채워야 하는 영역을 찾는 기본적인 기법은 스킨의 두께에 따라 여러분 아래의 여러 레이어들을 관찰하는 것이다. 하위 레이어에 에어가 있는 곳에는 현재 레이어에 스킨이 있어야 한다.



위의 이미지에서 스킨과 인필로 나뉘는 레이어를 빨강으로 보여준다. 이 영역에서 껍질이 될 부분과 채워질 부분을 결정하기 위해 파랑색 레이어를 살펴보고 있다. 파랑 영역을 제외한 빨강 영역 내부가 껍질이 된다 빨강과 파랑 영역에 모두 해당되는 영역이 채워질 것이다.

하지만 실제로는 좀 더 복잡하다. CuraEngine은 현재 레이어 아래의 하나의 스킨 두께 인 레이어 뿐만 아니라 그 사이의 모든 레이어도 봐야한다. 그렇지 않으면 스킨 두께보다 작은 레이어들에 있는 갭은 꺼내지지 않을 것이다.상단의 스킨에 대해서는 CuraEngine이 아래 레이어 대신 현재 레이어의 상위 레이어를 관찰해야 한다.

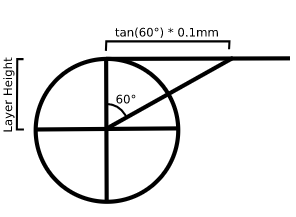
**Support**

서포트를 만드는 것은 완전히 복잡한 알고리즘이지만 여기서 요지를 설명한다.

서포트를 만드는 첫 단계는 인쇄물에서 돌출물의 위치를 결정한다. 그 지점에서 3D 격자의 표면들이 메모리에 유지되지 않으므로 Z축에 관한 특정한 각도를 가진 삼각형만 볼 수 없다. 단지 2D 레이어만 보아야 한다.

A normal, filled layer is assumed to support the layer that's above it. It will also support the layer above it fine even if the layer above it extends slightly beyond the layer below. The limit to how far the layer above can extend to still be supported by the layer below is what we'll call the "support distance". If we assume that the layer below supports the layer above if the slope between the edges of these layers is less than a certain angle (the Overhang Angle setting), then we can compute the support distance with a simple formula: tan(a) \* layer\_height

보통의 채워진 레이어는 그 위에 레이어를 지지한다고 가정한다.



CuraEngine will perform an offset of the current layer (drawn below in black) by this support distance (drawn in gray), and then subtract that from the layer above to get the area that is called the "basic support" (drawn in blue).



This type of support wouldn't print well however. It is all broken up into pieces. Typically this generates long, thin strands of areas (about 0.2mm wide) that need to be supported but gaps in between because technically that part in between is supported by the previous layer.

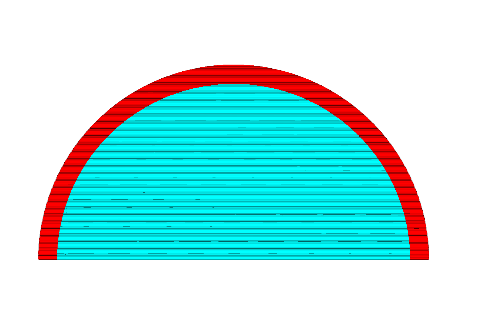
To get a more continuous support area, the basic support areas are merged. This merge is done by performing an offset of the support area by the support distance. Since the basic support areas tend to have the support distance between them, this causes adjacent support areas to overlap and merge together.



This does extend the support beyond the original part that needs to be supported, so it then subtracts everything from the support that is not in the layer above. This is called the "extended support" and forms the basis for the area that will eventually be filled with the support pattern. It needs some slight modifications though.

**Support X/Y and Z Distance**

The extended support would produce support that looks a bit like this:

[](https://github.com/Ultimaker/CuraEngine/blob/master/docs/assets/support_no_distance.png)

This looks nice but is undesirable since the support is directly adjacent to the mesh. It would stick far too well to the mesh. For that reason, some distance is created between the support and the model. Cura has several settings concerning the distance in the Z direction and in the X and Y directions.

The Z distance is maintained by not looking directly at the part that needs to be supported in the layer above the current layer, but by looking several layers higher, of which the total thickness equals the Z distance. This would then create a print like this:

If the slope is very steep or there is a bit of overhang next to a vertical wall, this would still cause problems since the support is still very close to the sides of the print. This can be seen in the image above at the bottom side of the print. For this reason, CuraEngine also maintains a horizontal distance, the Support X/Y Distance setting.

This introduces two constraints in the distance that support must keep to the model however: It must maintain a certain Z distance and a certain X/Y distance. Neither of these may be too small or too large. This is overconstrained, so one of them must supersede the other. This is determined by the Support Distance Priority setting, which can be set to prioritise either the Z distance or the X/Y distance. By default, the Z distance is prioritised. This causes our print to look a bit like this:

As you can see, the X/Y distance is only applied if the Z distance is not applicable: In the small area at the bottom where the slope was completely vertical. This does catch many cases with mechanical parts though where a vertical wall is next to a piece of overhang.

In many cases with more organic shapes this is still not desirable however, since the support will still stick to a slope that is *almost* vertical. For these cases there is another setting, Minimum Support X/Y Distance, which overrides the Z distance again. The print will then look like this:

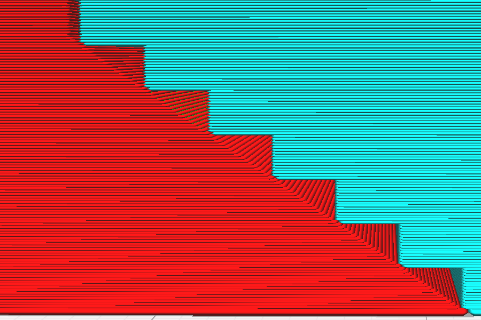
In the picture above there is a slight kink in the curve along the topside of the support if you look closely. Above the kink the Z distance is leading. Below the kink the Minimum Support X/Y Distance is leading.

**Stair-Stepping**

경사진 표면으로부터 쉽게 서포트를 제거할 수 있도록 하기 위해 계단-스테핑을 추가하였다.

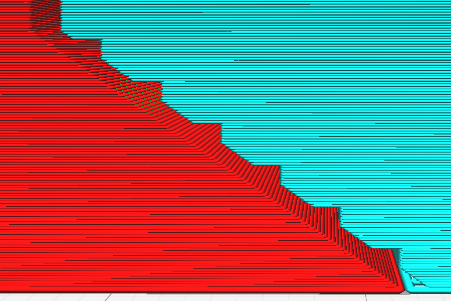
만일 서포트가 경시진 표면 위에 만들어져야 한다면 Support Stair 높이와 한 스텝이 인자로서 가질 수 있는 최대 Support Staire Step 너비로 불린다. 아이디어는 한 레이어에서 시작하고 그 위의 다음 몇 개의 레이어에서는 경사의 특정 영역에 놓이는 레이어들을 ‘고정’하는 것이다. 그 후에(스텝 높이에 도달하면)는 다음 레이어를 “풀고” 그것을 다음의 베이스로 여긴다.

In practice, this will create an inverted staircase, where the tips of the steps are closest to the model and likely the only contact-points. 실제로는 이것은 뒤집어진 계단을 만들 것이다. 계단들의

[](https://github.com/Ultimaker/CuraEngine/blob/master/docs/assets/stair_step_0.png)

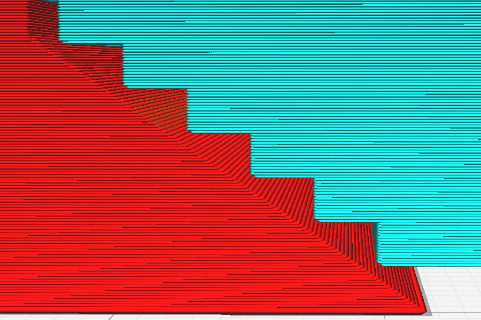
This picture shows clearly what stair-stepping is meant to do. For clarity, all support-offset distances have been set to 0.

There are some subtleties to the stair-stepping algorithm however, when it comes to the X/Y-distance offset and Z-distance offset: Namely, *if* support is higher above the sloped area because of X/Y-distance, then the additional height above the model *isn't* taken into account when the 'steps' are produced.

[](https://github.com/Ultimaker/CuraEngine/blob/master/docs/assets/stair_step_1.png)

As you can see in the picture above, in this case (Z-distance set to 0, X/Y distance increased, but lower than the step-height) it's as if the stair-stepping was done *first*, and only afterwards the X/Y distance was subtracted (that's not *quite* how it works internally, but the effect here is the same).

However, *if* support is higher above the sloped area because of the Z-distance, then the additional height above the model *is* taken into account when the steps are created.

[](https://github.com/Ultimaker/CuraEngine/blob/master/docs/assets/stair_step_2.png)

In the picture above you can see that effect in that case (Z-distance increased, X/Y-distance set to 0). Notice how the support is raised above the model, but the height of the stair-steps aren't influenced.

Of course, in practice, X/Y- and Z- distance offsets are both often greater than none. In that case it depends on the slope of the model, whether X/Y overrides Z or the other way around, and on the particular distances involved.

# Uranium - High Level Overview

우라늄은 여러 개의 모듈로 분리된 파트로 구성된다.

## Core

Core 모듈은 모든 애플리케이션에 맞는 기본 클래스와 객체들을 구성한다. 그 중에 [PluginRegistry] and [Application] 클래스가 핵심이다.

* Application 클래스: 모든 애플리케이션의 중심 객체이다. 주요 기능은 Controller와 MeshFilterHandler와 같은 다른 중요 객체에 대해 핵심적인 액세스 포인트를 제공하는 것이다.
* PluginRegistry 클래스: 플러그인과 메터데이터 로딩을 책임진다.
* Signal 클래스: 콜백을 사용하여 단순화된 이벤트 처리 메커니즘을 제공한다. 이것은 이벤트를 처리하는 라이브러이에서 사용된다. 예를 들어 Controller 클래스는 activeViewChanged 시그널을 방출한다. 이 시그널에 접속한 클래스는 이 이벤트의 공지를 받는다.
* Controller 클래스: Scene과 View클래스의 커뮤니케이션을 제공하는 객체로서 scene, view, input 장치와 툴 간의 이벤트도 처리한다.
* Event 클래스: 공통 이벤트를 제공하는 서브클래스와 함께 시스템의 이벤트에 대한 기반 클래스
* InputDevice 클래스: 툴과 뷰에게 입력을 제공하는 device를 위한 추상 기본 클래스로서 커스텀 입력 장치를 제공하기를 원하는 플러그인들에 의해 서브클래싱 되어야 한다
* Logger 클래스: 특정 출력에 로깅하는 클래스들을 위한 베이스 클래스. 대안으로 로깅 기능을 제공하는 플러그인에 의해 서브클래스화 될 수 있다.
* StorageDevice: 파일 열기와 닫기의 추상화를 제공한다. 파일 업로드와 같은 일들을 원하는 플러그인으로 서브클래싱 되어야 한다.
* Tool : scene에서 동작하는 툴을 위한 베이스 클래스이다.

## Backend

Backend는 데이터를 처리하기 위해 외부의 애플리케이션과 통신하는데 관련된 클래스를 제공한다. 이 기능의 주요 예제는 CuraEngine 슬라이싱 엔진과 통신이다.

이 기능을 제공하는 주요 클래스가 BackEnd 클래스이다. 이는 추상 클래스로서 BackEnd 기능이 필요한 플러그인에서 서브클래싱 해야 한다. BackEnd 클래스는 그 BackEnd 가 지원하는 명령 목록을 제공한다.

Command 클래스는 이러한 명령들을 포장하여 그의 send()와 receive() 메소드에서 명령의 실제 구현을 제공한다. Socket은 로컬 TCP 소켓 상에서 백엔드 애플리케이션과 대화하는 명령을 제공한다. 이 클래스는 백엔드와 통신하는데 사용하는 시스템 소켓을 만든다. Command 객체는 이 소켓에 보내진 실제 데이터의 직렬화를 담당한다.

## Math

Math 모듈은 선형대수를 다루는 클래스를 제공한다.

Matrix, Vector, Quaternion 클래스는 Numpy 배열을 기반으로 하며 기본적인 4x4 어핀 변환 로직을 구현한다. 이들은 주로 SceneNode와 그들의 변환 관련된 객체들에서 사용된다.

## MeshHandling

MeshHandling 모듈은 메시 데이터를 다루는 클래스들을 제공한다. 중요 클래스는 MeshFileHandler이다. 이것은 Application 객체가 만들며 메시의 읽기/쓰기 기능을 제공한다. MeshReader와 MeshWriter 객체 리스트를 사용하여 실제의 격자 읽기와 쓰기 로직을 구현한다. 이 두가지 클래스는 특정의 메시 포맷의 읽거/쓰기를 지원을 제공하는 플러그인에 의해 서브클래싱 되어야 한다.

메시를 읽을 때 MeshFileHandler는 메시 데이터를 저장하는 내부 데이터 객체인 MeshData 객체를 리턴한다. MeshData는 메시의 면을 형성하는 인덱스 리스트를 가진 Vertex 객체 리스트로 구성된다. 또한 메시 데이터를 다루는 편리 메소드도 제공한다.

## Scene

Scene 모듈은 scene에 있는 모든 데이터를 처리하는 클래스를 제공한다. Scene 자체는 Scene 클래스로 표현된다. Scene 클래스는 SecenNode 객체를 제공한다. SceneNode는 scene 그래프를 구성하는 트리 노드에서 한 노드이다. 각 scene 노드는 변환을 가지며 그의 부모와 상대적인 변환을 제공할 수 있다. 루트 scene 노드 외에 Scene 클래스는 활성 Camera 객체를 제공한다.

## Settings

Setting 모듈은 기게 설정을 처리하는 클래스를 제공하나. MachineSetting 클래스는 기계에 관한 모든 설정을 읽고 쓰는 핵심 객체를 제공한다. 그는 SettingCategory 객체 리스트를 제공한다. 각 Setting 카테고리는 Setting 객체의 트리를 제공하여 기계의 설정을 표현하는데 사용한다. 이 트리 표현의 말단 노드들은 더 상세한 설정을 표현 한다. 예를 들어 “Speed” 설정은 “Wall Speed”와 “Infill Speed” 설정을 자식으로 가진다. MachineSetting 클래스는 JSON 파일에서 트리 구조를 로딩하며 ini 파일에 값을 저장한다.

설정 모듈의 Validator 서브 모듈은 설정을 검증하는데 사용하는 여러 클래스를 제공한다. 이러한 검증기들은 로딩한 JSON에서 지정된 MachinSetting 객체가 만든다. 그들은 에러 범위, 경고 범위와 안정 범위를 제공한다.

## View

View 모듈에는 뷰와 렌터러를 위한 베이스 클래스가 들어있다. 뷰를 제공하는 플러그인은 View 클래스를 서브클래스해야 한다. View 클래스는 완벽한 렌더링 경로를 제어하며 모든 것이 어떻게 그려질지즐 결정해야 한다. 이것은 Application이 제공한 렌더러를 이용해야 한다. Renderer는 애플리케이션이 만들며 저변의 그래픽 API를 추상화한다.

## Toolkit Specifics

대부분의 라이브러리는 툴킷의 특정 코드를 포함하지 않는다. 대신에 대부분의 툴킷에 특정한 코드는 tookkit-specific 서브 모듈에 들어 있다. 현재는 Qt 서브 모듈에 관한 것이며 PyQt를 사용하여 대부분의 UI를 구현하는 툴킷 특정 클래스들이 들어 있다.

# Rendering

Uranium은 scene과 그의 컨텐츠에 대해 MVC와 유사한 방식을 사용한다. 이 방식에서는 모델이 Scene 클래스이고 SceneNode는 Scene를 담는다. 컨트롤러 부분은 Controller 클래스와 Tool 서브클래스에 의해 처리된다. View 클래스는 현재 scene를 보는데 관련된 모든 것을 처리한다. 이것은 현재의 활성 View가 무엇을 어떻게 그릴지를 항상 제어한다는 것을 의미한다. 특정 객체에 특정한 행동을 기본으로 할 수 있지만 View는 항상 이러한 행동을 갈아치울 수 있는 능력이 있다.

물론 View가 스스로 모든 것을 처리할 필요는 없다. 렌더링은 OpenGL을 통해 이루어며 OpenGL을 간단하게 처리하는 편리한 클래스가 있다. 또한 Renderer, RenderingBatch, RenderPass 클래스는 공통 기능을 분리한 주요 클래스들이다.

## Rendering Pipeline

렌더링은 툴킷의 이벤트 처리의 어딘가에서 시작하며 툴킷이 모든 것을 다시 페인트해야 한다고 결정할 때이다. 이것은 대개 페인트 이벤트를 메인 윈도우에 보낼 것이다. Main 윈도우는 이 이벤트를 처리하고 렌더링 프로세스를 시작한다. 이 시점에서 두 객체가 필요하다. 하나는 Renderer 인스턴스로서 애플리케이션마다 하나가 있다. 또 다른 것은 현재의 활성 View의 인스턴스이다.

1. Renderer::beginRendering()을 호출하여 OpenGL을 셋업한다.
2. View::beginRendering()를 호출하여 View가 무엇을 그릴지를 결정할 수 있게 한다. View의 beginRendering() 구현에서는 View가 Renderer::queueNode() 를 불러 렌더링할 노드들을 큐에 넣는다. 또한 View는 추가적으로 RenderPass 인스턴스를 만들어 현재 View를 렌더링하도록 할 수 있다.
3. 일단 큐에 view를 넣으면 Render::render()가 불린다. 이것은 그릴 필요가 있는 항목들을 정렬하고 그리는데 만들 필요가 있는 사후 처리를 한 다음 RenderPass 인스턴스 목록을 순회하면서 RenderPass::render()를 호출한다. RenderPass::render()가 실제의 객체 렌더링을 수행하며 최종적으로 그려져서 그려진 이미지가 scene에 놓여질 준비가 되어야 한다.
4. 끝으로 RenderPass::render()가 끝나면 먼저 View::endRendering()을 불러 View가 필요한 클린업을 할 수 있도록 해주고 그 후에 Renderer::endRendeing()을 불러 Rrender가 클린업할 기회를 준다.

## Render Passes

View가 어떤 사물을 어떻게 그릴지에 관한 최종 중재자이고 RenderPass 객체도 이 문제에 관여한다. 각 RenderPass는 scene에서 rorcemf을 렌더링하는 단일 경로를 표현한다. 이 경로들은 그리는 방식과 무엇을 그릴지에 따라 변화가 클 수 있다.

기본적으로 세가지 RenderPass 인스턴스가 있다.

* SelectionPass: 선택 가능한 모든 객체를 텍스처로 렌더링 한 다음 현재 커서 아래에있는 객체를 얻기 위해 쿼리 할 수 ​​있다
* DefaultPass: View에서 지정된 모든 것을 텍스처에 그린다.
* CompositePass: 이전 경로들은 화면에 그려질 최종의 이미지로 묶어서 최종의 RederPass가 되도록 한다.

기본적으로 DefaultPass의 컨텐츠를 풀스크린 쿼드에 그린 다음에 SelectionPass로 선택된 객체 주변의 윤곽선을 그릴 것이다. CompositePass의 shader와 바인딩은 필요할 때 자유롭게 바꿀 수 있다.

## SceneNode::render() versus Renderer::queueNode()

View 서브클래스 구현 시 한가지 주지할 것은 특정 SceneNode 서브클래스들이 자신만의 방식으로 렌더링을 결정할 수 있기를 원한다는 점이다. 이것은 SceneNode::render() 메소드의 구현을 통해 이루어진다 View들은 디폴트 렌더링 행동을 원할 때 또는 특정 SceneNode 서브클래스를 그릴때 이것을 불러야 한다. 예를 들어 Platform 클래스는 항상 Platform 메시를 플랫폼 렌더링을 위한 오른쪽 shader와 함께 큐에 넣도록 해야 한다. 만약 SceneNode::render() 가 false를 리턴하면 SceneNode는 스스로 렌더링을 제어할 필요가 없고 View가 원하는 방식대로 렌더링 될 수 있다는 뜻이다. View는 그 다음에 queueNode()로 노드를 렌더링할 수 있다. View가 여전히 제어를 하고 있다는 점을 주시하라.

# 번역 가이드

이 페이지는 Uranium& Cura 번역에 관한 절차와 방법을 설명한다. 개발자는 문자열을 번역가능으로 표시한다. 이 문자열은 소스파일에서 추출되어 하나 이상의 번역 파일에 복사된다. 이러한 번역 파일은 번역을 위해 보내진다. 일단 번역이 되면 이 파일들은 번역 카타로그로 컴파일 된다. 이 번역 카탈로그는 번역된 문자열을 찾는데 사용된다.

이 문서의 첫번째 단락은 Cura로 개발하는 모든 개발자에게 중요한다. 문서의 나머지는 번역을 추출(또는 테스트)하는 개발자를 위한 것이다.

## Mark strings as translatable in Python

Cura 또는 Uranium i18n Catalog를 import한다. 알맞은 카탈로그를 i18n\_catalog 변수에 지정한다.

from UM.i18n import i18nCatalog

i18n\_catalog = i18nCatalog("cura")

Each string in Uranium should be provided with a context. These contexts and the strings are marked up using "semantic markup", a system based on KDE's [semantic markup]. This format is intended to provide a standardised way of providing additional information about the usage and the meaning of strings. [semantic markup]: <https://techbase.kde.org/Development/Tutorials/Localization/i18n_Semantics>

###Use named arguments for string formatting When you perform string formatting with multiple arguments, you need to use named arguments instead of positional arguments.

return self.\_i18n\_catalog.i18nc("@info", "%(width).1f x %(depth).1f x %(height).1f mm") % {'width' : self.\_scene\_boundingbox.width.item(), 'depth': self.\_scene\_boundingbox.depth.item(), 'height' : self.\_scene\_boundingbox.height.item()}

###String Placeholders and Semantic Tags String contents can have placeholders where certain other elements should be inserted. For example, these are used to insert file names when displaying messages about file saving. These placeholders are use a format with a number surrounded by curly brackets ({}). Additionally, these placeholders can be surrounded by semantic tags. Semantic tags indicate the function of a placeholder.

Currently, the following two semantic tags are available:

* Used to denote a filename.
* Used to denote an error message.

###Add extra context-markers for alt-key accelerated menu-items In the menubar we use alt-key accelerators. (With alt + E the Edit menu opens; after this with alt + S you activate Delete Selection.) These strings need extra context so the translator knows in which group the chosen accelerated letter needs to be unique. We currently have the following added contact markers:

menubar:toplevel (→ all toplevel menu-items)

menubar:file (→ all items that fall under the 'file' toplevel menu-item)

menubar:edit

menubar:printer

menubar:profile

menubar:settings

menubar:help

Use the following format to add a context-marker to a menu-item:

i18n\_catalog.i18nc("@action:inmenu menubar:edit","&Group Objects")

You also add these context markers when there is only one item in the menu or when the menu-item doesn't use an alt-key accelerator (yet). If you change the string of a toplevel item; you obviously also change the context-markers of the underlying menu-items. More info on alt-key accelerators here: <http://doc.qt.io/qt-5/linguist-translators.html#changing-keyboard-accelerators>

## Mark a string as translatable in QML

Insert the i18n catalog object into your qml file. Obviously the name needs to be cura or uranium, whether you're working in Cura or Uranium.

UM.I18nCatalog{id: i18n\_catalog; name:"cura"}

For convenience we name the id 'i18n\_catalog'. Everything else works exactly the same as in python:

i18n\_catalog.i18nc("@action:button", "Back");

## Extract the messages

The following steps are pretty much the same for both Cura and Uranium, but have to be performed for both separately. The only difference is that in Cura you have to open CmakeLists.txt and make sure that URANIUM\_SCRIPTS\_DIR is set to the correct folder (the folder in Uranium that contains the scripts). Use the terminal to create the buildfiles in a new build folder in the source-directory:

mkdir build

cd build

cmake ..

Now build the build-target [extract-messages] (with the terminal; when your still in the build directory). (If it starts wining that it can't find some directory; fix it, throw away the build folder and create the build-files again.)

make extract-messages

It has created a freshly extracted POT file. Don't worry if you're not a 100% sure that the files are correct; we are going to perform a very telling test later in the process. Having said that; you do need to check the following: The script tends to break on certain characters in the JSON files. Review these files by checking if the last translatable string in the json file is also the last item in the POT file.

## Theory on translation files

First some theory on the magic translation files. These are the different files:

| Name | Long Name | File-extension | Description |
| --- | --- | --- | --- |
| POT file | Portable Template Object file | .pot | template for the PO files |
| PO file | Portable Object file | .po | contains the actual translations |
| MO file | Machine Object file | .mo | binary code; used by Cura |

As said before [extract-messages] creates a POT file. The POT file is a template for the .PO files.The PO files are based on the POT file. And the PO files are compiled into MO files.

## Creating the PO files

[extract-messages] Has already created the PO files it can automatically create (English and x-test). The other PO files are created by merging the old PO file with the new POT file. If you don't merge them; you have to get completely new translations with every new release. An added bonus is that translations that are no longer used don't get thrown away, but are commented out and placed in the bottom of the page for later use. Use the terminal to create the new PO files:

make i18n-create-po

For a specific language code use:

make i18n-create-po-<code>

msgmerge Does such a great job that I usually just rewrite the old PO file.

make i18n-update-po

Same here for a specific language:

make i18n-update-po-<code>

## Creating the MO files & testing

In the building process; the MO files are automatically created. So you only manually create the MO files that you want to test. A great example is x-test. The PO files for x-test are automatically created by [extract-messages]. It is basically an English translation with two X's before and after each string. You use it to test whether all strings are being properly translated.

Create a directory in the directory of the language (x-test is this example). The name of the new directory has to be [LC\_MESSAGES]. Create the MO file with the terminal:

make i18n-update-mo

Or for your language:

make i18n-update-mo-<code>

Setting the environment variable doesn't function. So for now you need to edit the language default in: Uranium/UM/i18n.py

def \_\_init\_\_(self, name = None, language = "x-test"):

Carefully test whether ALL strings are being properly translated, before you send the PO files to the translators. You can test this by looking if all strings have 2 X's around them.

## Finally

Send both the POT files and the PO files to the translators. Translators like to use PO editors. These are programs that allow them to translate one string at-a-time without having to look into the actual code. These programs tend to turn multi-line strings into single line strings. This is not a problem for Cura, just leave it as it is.