Smashcima Technical Documentation[1]

2024-12-30

Mgr. Jiří Mayer, Doc. Pavel Pecina, MgA. Jan Hajič jr., Ph.D.

Prague Music Computing Group

Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University

[1]This work has been done by the OmniOMR project within the 2023-2030 NAKI III programme, supported by the Ministry of Culture of the Czech Republic (DH23P03OVV008).

Contents

1 Introduction **3**
1.1 What is Smashcima, who is it for, and how is it novel? 3
1.2 Development 4
1.3 Financing 4
1.4 How to cite 4
1.5 Contact 4

2 Design Overview **5**
2.1 Synthetic data 5
2.2 Data model (the scene) 5
2.3 Exporting (and rendering) 6
2.4 Synthesis 6
2.5 Models 6
2.6 Assets 6
2.7 Other submodules 6

3 Models and service orchestration **9**
3.1 External model interface 9
3.2 Service orchestration 10
3.2.1 Service container 10
3.2.2 Service construction in a model 12
3.2.3 Pre-defined services 12
3.3 Re-configuring existing models 13

4 Scene objects **15**
4.1 Understanding the data model by designing scene types from scratch 15
4.1.1 Containers 18
4.1.2 Extensibility 19
4.2 How `SceneObject` works 21
4.2.1 Lists and `None` 21
4.2.2 Memory leaks 21
4.2.3 Detaching objects from scene 21
4.3 Relationship querying 22
4.3.1 Named queries 23
4.4 When to use the `Scene` class 23
4.5 Conclusion 24

5 Affine spaces and rendering **25**
5.1 Affine space 25
5.1.1 Space hierarchy 25
5.1.2 Constructing affine spaces 26
5.1.3 Understanding transforms 26

5.1.4 Units are millimeters . . . 27
5.2 Sprites . . . 27
5.3 View boxes . . . 28
5.4 Bitmap renderer . . . 28

6 Synthesizer interfaces **31**
6.1 Smashcima synthesizers . . . 31
6.2 Using synthesizers . . . 31
6.3 Extensibility . . . 31

7 `ColumnLayoutSynthesizer` interface **33**
7.1 Public API . . . 33
7.2 Implementations . . . 34

8 `GlyphSynthesizer` interface **35**
8.1 `Glyph` scene object . . . 35
8.1.1 Placing a glyph into a scene . . . 36
8.1.2 Glyph origin point . . . 36
8.1.3 Semantics of visual objects . . . 36
8.2 Public API . . . 37
8.2.1 Checking supported glyph labels . . . 37
8.3 Implementing a glyph synthesizer . . . 37
8.4 Implementations . . . 38

9 Glyphs **39**
9.1 Clefs . . . 39
9.2 Time signatures . . . 40
9.3 Noteheads . . . 40
9.4 Augmentation dot . . . 40
9.5 Flags . . . 40
9.6 Accidentals . . . 41
9.7 Articulation . . . 42
9.8 Rests . . . 42

10 Line glyphs **43**
10.1 List of line glyphs . . . 43

Chapter 1

Introduction

Smashcima is a library and framework for synthesizing images containing handwritten music for creating synthetic training data for Optical Music Recognition (OMR) models.

Try out the demo on Huggingface Spaces right now!
Example output with MUSCIMA++ writer no. 28 style:

Install from pypi with:

```
pip install smashcima
```

This document is for those who want to not just apply Smashcima in their OMR workflows within their visual domains (such as new manuscript datasets), but to understand how it is designed and contribute to its development (e.g. to enable it to operate without individual glyphs).

1.1 What is Smashcima, who is it for, and how is it novel?

Smashcima is a Python package primarily intended to be used as part of optical music recognition workflows, esp. with domain adaptation in mind. The target user is therefore a machine-learning, document processing, library sciences, or computational musicology researcher with minimal skills in python programming.

Smashcima is the only tool that simultaneously:

- synthesizes handwritten music notation,
- produces not only raster images but also segmentation masks, classification labels, bounding boxes, and more,
- synthesizes entire pages as well as individual symbols,
- synthesizes background paper textures,
- synthesizes also polyphonic and pianoform music images,
- accepts just MusicXML as input,

- is written in Python, which simplifies its adoption and extensibility.

Therefore, Smashcima brings a unique new capability for optical music recognition (OMR): synthesizing a near-realistic image of handwritten sheet music from just a MusicXML file. As opposed to notation editors, which work with a fixed set of fonts and a set of layout rules, it can adapt handwriting styles from existing OMR datasets to arbitrary music (beyond the music encoded in existing OMR datasets), and randomize layout to simulate the imprecisions of handwriting, while guaranteeing the semantic correctness of the output rendering. Crucially, the rendered image is provided also with the positions of all the visual elements of music notation, so that both object detection-based and sequence-to-sequence OMR pipelines can utilize Smashcima as a synthesizer of training data.

(In combination with the LMX canonical linearization of MusicXML, one can imagine the endless possibilities of running Smashcima on inputs from a MusicXML generator.)

1.2 Development

Smashcima is being developed on GitHub: https://github.com/OMR-Research/Smashcima. It is part of the `OMR-Research` organization to maximize reach within the OMR community.

Documentation specific to contributing is available directly in the software's GitHub repository.

1.3 Financing

This work has been done by the OmniOMR project within the 2023-2030 NAKI III programme, supported by the Ministry of Culture of the Czech Republic (DH23P03OVV008).

1.4 How to cite

There's a publication being written. Until then, you can cite the original Mashcima paper:

> Jiří Mayer and Pavel Pecina. Synthesizing Training Data for Handwritten Music Recognition. *16th International Conference on Document Analysis and Recognition, ICDAR 2021.* Lausanne, September 8-10, pp. 626-641, 2021.

1.5 Contact

Developed and maintained by Jiří Mayer (mayer@ufal.mff.cuni.cz) as part of the Prague Music Computing Group lead by Jan Hajič jr. (hajicj@ufal.mff.cuni.cz).

Chapter 2

Design Overview

Smashcima is a python package aimed at synthesizing training data for Optical Music Recognition (OMR). Its ultimate goal is to create synthetic images of music notation, together with the corresponding annotations (textual, visual, or both). The internal structure of Smashcima follows from this goal and is described in this file.

2.1 Synthetic data

Deep learning methods currently yield the best results for tackling OMR and these need training data in order to work. While some training data may be produced manually, this process is costly and synthesis could be used to mix, shuffle, and reuse this small amount of manually annotated data to produce much larger amount of synthetic data. Therefore the goal of data synthesis here is not necessarily to create new data out of thin air, but rather to augment existing data to prevent overfitting during model training, making the resulting recognition models more robust.

The training data for supervised methods (which are among the most prominent) comes in pairs of input images and corresponding output annotations. These input images are usually scans or photos of physical music notation documents, which are available in bitmap formats (JPG, PNG). These images can be individual musical symbols, staves, or whole pages. The annotations are, however, much more diverse and often task-specific. They could be image-classification classes, image-detection bounding boxes, image-segmentation masks, music notation graphs, sequential textual representations (aligned with the image or not), or complex notation formats, such as MusicXML, MEI, Humdrum **kern, Lilypond, ABC, and others.

The purpose of Smashcima is to synthesize such image-annotation aggregates.

2.2 Data model (the scene)

In order for the synthesizer to get a handle on the synthetic data during synthesis, it needs some internal representation of the synthesized music page - a data model. This data model is called the **scene** and it exists as a cluster of python class instances that inherit from the abstract class `SceneObject` and form an interlinked graph.

A scene should contain enough information to produce most desired annotation formats together with the image bitmap.

Scene objects live in the `smashcima.scene` module.

2.3 Exporting (and rendering)

A scene is not the image, nor the annotation. It is more: it contains all the necessary information to produce both. The process of extracting a specific annotation format (say MusicXML) from the scene is called **exporting**.

Similarly, the image bitmap itself can be exported from the scene, but since now we deal with visual data, we call this process **rendering** (terminology borrowed from computer graphics). In other words, rendering is a subset of exporting.

This distinction between the scene and the exported format lets us add exporters for additional output formats on demand.

Exporters and renderers live in the `smashcima.exporting` module.

2.4 Synthesis

The core of the actual synthesis lies in the process of constructing the scene. The scene is the data structure, and a synthesizer is the algorithm that processes it. Specifically, a synthesizer is some python code that, given some input arguments, constructs a specified subset of the scene (which can be of course the scene in its entirety as well).

> **Note:** APIs of synthesizers vary: It can either create a part of a scene and return it, or it can add something into an existing scene. It depends on the task.

Synthesizers come in lots of sizes, from small ones synthesizing individual symbols, to large ones putting together the whole page. Smashcima should act as a collection of synthesizers for you to choose and match, given the OMR task you want to solve.

Synthesizers live in the `smashcima.synthesis` module.

2.5 Models

While synthesizers do the heavy lifting, their configuration is complex and they often rely on many other synthesizers (e.g. the music notation synthesizer relies on a glyph synthesizer). Asking every single user to build their own synthesis pipeline from scratch would be impractical. Therefore, Smashcima introduces the concept of **models**.

Models act as a ready-to-use wrappers around various synthesis pipelines, being reasonably pre-configured out of the box. While a synthesizer is meant to be as general as possible, a model is built to be as specific as possible. You build your own model for your specific task domain - either by adapting existing models, or by putting together a custom synthesizer pipeline from scratch.

Models are meant to *orchestrate* synthesizers. They live in the `smashcima.orchestration` module.

2.6 Assets

The best synthetic data is partially-real data. Therefore, almost all synthesizers need some dataset, some trained generative model, or some set of tuned parameters to work. These real-world input resources are called *assets*. The Smashcima system has an assets layer responsible for their definition, download, preparation and usage.

Assets live in the `smashcima.assets` module.

2.7 Other submodules

Other Smashcima submodules that have not been covered above:

- `smashcima.geometry` Contains types for working with 2D geometry (vectors, points, transforms, polygons).
- `smashcima.loading` Classes that construct the semantic part of a scene by loading it from a music notation format (e.g. MusicXML). They are not synthesizers, because they don't create new data - they just load if from some other format.
- `smashcima.jupyter` Helper methods for working with Smashcima from Jupyter notebooks (mostly scene visualization code). The `smashcima[jupyter]` extra dependencies must be installed in order to use this module.

Chapter 3

Models and service orchestration

When you first start using Smashcima, you will mostly interact with models (extensions of the `Model` abstract class). Model (as in *generative model*) represents a pre-configured synthesis pipeline that produces synthetic data modelling a specific data domain (a specific evaluation dataset / visual style / data type).

Therefore the responsibilities of models are twofold:

1. Provide a polished external interface for synthesizing the modelled data domain.
2. Set up and configure internal synthesizers and services to facilitate the data domain modelling, while allowing for at least some re-configuration by the user (i.e. orchestrate its internal services).

Code explored in this documentation section lives in the `smashcima.orchestration` module.

3.1 External model interface

A model is constructed like any other python object:

```
import smashcima as sc

my_model = sc.BaseHandwrittenModel()
```

Constructor arguments depend on the model used, but the model should always provide default values for all of them to allow for construction without any arguments provided. This is because a model should by default already model some data domain well and customization should only be applied later if needed.

The constructed model instance is callable, so it pretends to be a data-generating function. By calling it, we generate a new sample of data:

```
new_scene = my_model("input.musicxml")
```

The returned data sample is called a *scene*, because it contains much more information than just an image or an annotation. The actual desired synthetic data must then be exported from the synthesized scene.

Arguments to the model invocation depend completely on the specific model. The `BaseHandwrittenModel` shown in this example expects a MusicXML input, either as a file, or as an XML string. But you might have models that require no arguments, or others that require some random latent vector **z** (e.g. Generative Adversarial Networks), etc.

Similarly, the type of the returned scene instance is also completely controlled by the specific model used. The `BaseHandwrittenModel` returns an instance of `BaseHandwrittenScene`, which contains representation of the music, the synthesized pages, a renderer that will be used for rasterization, and metadata about the chosen MUSCIMA++ writer style (MUSCIMA++ is the default source of assets).

For this specific scene type, getting the bitmap image is done like this:

```python
# BGRA OpenCV image
img = new_scene.render(new_scene.pages[0])
```

> **Note:** You can imagine how this is very tightly linked to the data domain of handwritten scores of music notation in the MUSCIMA++ dataset. A model designed for synthesis of isolated musical symbols, rather than entire scores, could have a completely different API: accepting a symbol class as an argument and outputting a single symbol image as output, with possible latent space embedding (or similar metadata).

The synthesized scene is also assigned to the model instance under the `.scene` field. This field is `None` before the first invocation. This is for situations when you cannot store the return value from the model invocation immediately:

```python
# an alternative way of getting the scene
# (though the first one is preferred)
my_model("input.musicxml")
new_scene = my_model.scene
```

3.2 Service orchestration

When a model is constructed, it creates a set of synthesizers and auxiliary services, configures them, and connects them to each other. It orchestrates a synthesis pipeline that will be used once the model gets invoked.

These include synthesizers for glyphs, line glyphs, stafflines, page background, page layout, music notation, or stems and beams. The auxiliary services include a random number generator and an assets repository. All of these services depend on one another, for example, the music notation synthesizer uses the glyph synthesizer and both use the random number generator and the assets repository.

Since the manual construction of these services would be tedious and modification of the construction process in this case by the user would be impossible, the model instead uses a service container for the service construction (also known as an IoC container).

3.2.1 Service container

Each model has its own service container available under the `.container` field.

The service container can be used to construct a dependency graph of service instances in two steps:

1. You register services into the container
2. You resolve the service you want to use

During the registration step, you tell the container what services you wish to be used. This can happen in roughly three ways:

- You give the container an existing instance for a given service type. So, you give it a specific `my_rng` value for the service `random.Random`. When the container is asked to resolve the service `random.Random`, it will return the `my_rng` value.
- Or you tell the container just the service type, e.g. `MuscimaPPStyleDomain`. It will then figure on its own how to construct the type when asked for its instance (by default using the argument-free constructor, or by recursively resolving all the arguments).
- Or you tell the container what specific service type it should construct when asked about an abstract service. For example, you tell the container to construct an instance of `MyFancyGlyphSynthesizer` when the user asks for a `GlyphSynthesizer`.

Once all of these registrations take place, you can that resolve a service from the container (e.g. `GlyphSynthesizer`) and it will recursively construct it, together with all of its dependencies (e.g. `random.Random`) and give it back to you.

These are the methods you can use to register services into the container:

```
import random
my_rng = random.Random(42)

# register an existing instance
container.instance(random.Random, my_rng)

# register a type
container.type(sc.MuscimaPPStyleDomain)

# register an interface implementation
container.interface(sc.GlyphSynthesizer, MyFancyGlyphSynthesizer)
```

You can then ask the container to give you the glyph synthesizer instance, which will be constructed by the container using the RNG and style domain registered above:

```
# construct a service based on type registrations
my_glyph_synth = container.resolve(sc.GlyphSynthesizer)

assert type(my_glyph_synth) is MyFancyGlyphSynthesizer # succeeds!
```

3.2.1.1 Singletons

When you resolve a service twice, the container will only construct it once and then return the same instance again:

```
# every service is constructed only once
first = container.resolve(sc.GlyphSynthesizer)
second = container.resolve(sc.GlyphSynthesizer)

assert first is second # succeeds!
```

This behaviour is called singleton registration - each service always exists only in one instance, i.e. a singleton.

It means that even if `random.Random` is requested by twenty other services, there will only be a single instance created and reused by all of them.

> **Note:** This is a slight simplification from general IoC containers, say in web applications, where the lifetime of services can be configured and there exist scoped and transient services. However, it made little to no sense in our case of constructing synthesis pipelines.

3.2.1.2 No implicit registrations

The service container cannot resolve types that have not been explicitly registered. If that occurs, the resolution raises an exception, and you need to provide the registration manually.

3.2.1.3 Complex service constructors

Sometimes, services may require arguments that cannot be automatically resolved by the container during constructions (e.g. having `str` or `int` arguments).

If possible, you can register the service as an instance:

```
my_rng = random.Random(42)
container.instance(my_rng)
```

But if the service depends on other services that will yet to be constructed by the container, you can register a factory function instead of the true service constructor:

```
def my_service_factory(dependecy: OtherService) -> MyService:
    return MyService(dependency, some_number=42)
```

```
container.factory(MyService, my_service_factory)
```

3.2.2 Service construction in a model

Now that we know how a service container works, we will look at how it is utilized within a model constructor.

At the end of the `Model.__init__` method, there are these three methods invoked:

```
self.register_services()
self.resolve_services()
self.configure_services()
```

These are designed for you to override, when you create your own model.

- `register_services` is used to register services into the container via the `.instance`, `.type`, and `.interface` methods on the container.
- `resolve_services` is used to resolve services from the container and assign them to some model fields, so that they can be used later during configuration and synthesis without calling the low-level `container.resolve` method.
- `configure_services` is used to modify services after they are constructed, altering their behaviour.

This is what these methods can look like when implemented:

```
class MyModel(sc.Model):
    def register_services(self):
        super().register_services()

        # register a service
        self.container.interface(
            sc.GlyphSynthesizer,
            MyFancyGlyphSynthesizer
        )

    def resolve_services(self):
        super().resolve_services()

        # resolve a service
        self.glyph_synthesizer: MyFancyGlyphSynthesizer \
            = self.container.resolve(sc.GlyphSynthesizer)
        assert type(self.glyph_synthesizer) is MyFancyGlyphSynthesizer

    def configure_services(self):
        super().configure_services()

        # configure a constructed service
        self.glyph_synthesizer.level_of_fancy = 999
```

3.2.3 Pre-defined services

The `Model` base class automatically registers a number of useful services into the service container for you to use. These are:

- `random.Random` instance to generate random numbers
- `sc.AssetRepository` instance to provide access to asset bundles
- `sc.Styler` instance to control style parameters for each synthesized sample

These instances are likely to be needed by almost all models, and they are very common synthesizer dependencies.

The random number generator and styler are also exposed via model fields, so that you can access them inside and outside the model:

```
model.rng     # random.Random
model.styler  # sc.Styler
```

3.3 Re-configuring existing models

There are two ways in which to modify model services:

1. change the way in which they are constructed (e.g. use a different `GlyphSynthesizer` or modify its constructor arguments)
2. change their configuration after they are constructed

The first approach is achieved by overriding container registrations at the end of the `register_services` method.

For example, let's say we want to control the seed of the `random.Random` instance. We could do the following:

```
import random
import smashcima as sc

class MyModelWithSeed(SomeBaseModel):
    def __init__(self, seed: int):
        # store the seed before we call the super constructor
        self._seed = seed

        # the super constructor will call the `register_services` method
        super().__init__()

    def register_services(self):
        super().register_services()

        # override the `random.Random` registration
        self.container.instance(
            random.Random,
            random.Random(self._seed)
        )
```

This works, because registering a type into the service container for the second time replaces the old registration with the new one. This lets us modify the dependency graph of services, what specific service types are used, and what constructor arguments they are given.

The second way of modifying services is after their creation, by modifying their configuration fields.

For example, we could modify the line width in the `NaiveStafflinesSynthesizer`:

```
class MyModel(sc.BaseHandwrittenModel):
    def configure_services(self):
        super().configure_services()

        # get the naive stafflines synth instance
        synth = self.container.resolve(sc.StafflinesSynthesizer)
        assert type(synth) is NaiveStafflinesSynthesizer, \
            "Just making sure no one changed the registered type we expect"
```

```
        # modify its configuration
        synth.line_thickness = 0.1 # mm
```

Alternatively, for services that have been resolved in `resolve_services`, you can access them directly via their field:

```
class MyModel(sc.BaseHandwrittenModel):
    def resolve_services(self):
        super().resolve_services()

        # remember service instance in a model field
        self.fancy_glyph_synth: FancyGlyphSynthesizer \
            = self.container.resolve(sc.GlyphSynthesizer)
        assert type(self.fancy_glyph_synth) is FancyGlyphSynthesizer \
            "Checking nobody changed the registered type that we expect"

    def configure_services(self):
        super().configure_services()

        # configure a service
        self.fancy_glyph_synth.level_of_fancy = 999
```

But with these explicitly resolved services, you can modify them post-creation even from outside of the model:

```
# create a model
model = MyModel()

# configure a service
model.fancy_glyph_synth.level_of_fancy = 999

# use the model
model(...)
```

Note that for most configuration changes to a model, inheriting from the model and making a custom class derivative is necessary.

Chapter 4

Scene objects

A scene is the data model produced by synthesizers (and by models), from which we can export the target image and annotation format.

A scene is made up of `SceneObject` instances, that reference each other in a graph-like structure and together describe the contents of a synthetic sheet of music.

The purpose of a scene is to contain all information about a sheet of music and to let the user query this information easily. It's also designed to be easily extensible, drawing on ideas from the Resource Description Framework (RDF) used for Open Data.

> **Note:** The inspiration from RDF is to embrace the graph-like nature of the data model and implement extensibility by extending the graph (say, instead of using class inheritance).

4.1 Understanding the data model by designing scene types from scratch

Let's learn about how `SceneObjects` work by going through the design process of representing music and music notation.

Semantically, music consists of notes. A note is a sound that has some pitch and duration.

To represent pitch, we can use the Scientific pitch notation, which is just a combination of a number and a letter. Smashcima provides the type `sc.Pitch` we can use.

Duration is usually not represented by absolute time (milliseconds), instead we use musical time consisting of beats and their subdivisions. Smashcima provides the type `sc.TypeDuration` which represents duration in the "note type units" of "whole", "half", "quarter", etc.

Putting this together, we can define a `Note` scene object:

```
import smashcima as sc
from dataclasses import dataclass

@dataclass
class Note(sc.SceneObject):
    pitch: sc.Pitch
    """Scientific pitch notation (C4, G2, ...)"""

    type_duration: sc.TypeDuration
    """Note-type duration value (whole, half, quarter)"""
```

We can create an instance of a C4 quarter note like this:

```
my_note = Note(
    pitch=sc.Pitch.parse("4", "C"),
    type_duration=sc.TypeDuration.quarter
)
```

> **Note:** Using the `@dataclass` decorator helps us auto-generate the constructor with all the arguments and verifying that all are provided. It is not necessary to use it, though.

Now we can build a **kern loader that reads a `.krn` file and constructs a list of these notes. We can now represent the music semantically (although very simplified for the purpose of this example). We can imagine that the loader returns a `list[Note]` instance which lets us hold the whole score in one variable.

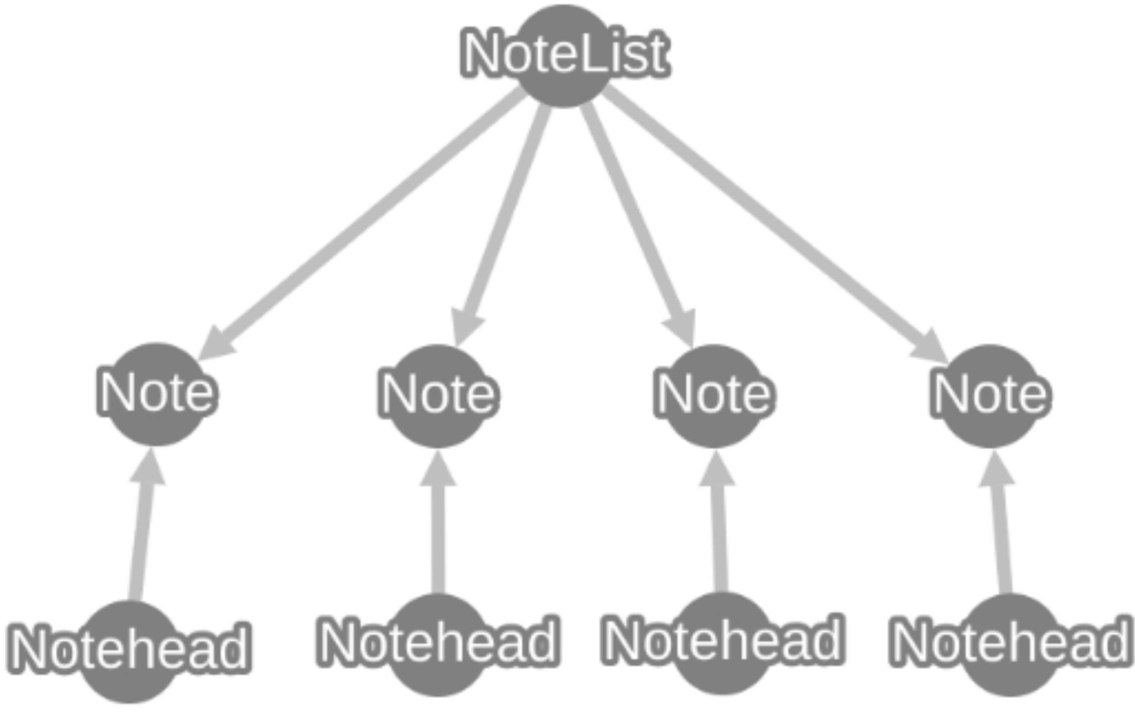
We can then build a synthesizer that generates an image for each note (notehead, stem, flag). For now, we will only consider the notehead:

```
import numpy as np

@dataclass
class Notehead(sc.SceneObject):
    note: Note
    """The semantic note that this visual notehead represents"""

    image: np.ndarray
    """The synthesized image of this notehead"""
```

Then we can build a renderer, that goes over all noteheads and puts them together into one bitmap for the whole music score.



Here, an interesting problem arises: We hold the score in a variable as a `list[Note]`. How do we get to the noteheads?

1. Currently the `Notehead` references a `Note`. This made sense so far, since we first load the notes and only then add the noteheads. But it prevents us getting to the noteheads from notes.
2. If we invert the relationship and have the `Note` reference a `Notehead`, we have to make the `Notehead` optional (being able to be `None`), otherwise the loader cannot load the notes. But that's incorrect semantically. You cannot have a note in the music score without any notehead. And relaxing this invariant just because of a technical difficulty seems incorrect.

To solve this problem (and many more), the `SceneObject` base class actually tracks all references to-and-from any `SceneObject` instance and remembers them internally. This allows us to query the inverse relationship like this:

```
print(my_note) # I have my note, I want to get its notehead

# gets the notehead of my_note via the ".note" reference
print(
    Notehead.of(my_note, lambda n: n.note)
)
```

This way, we can pass the `list[Note]` to our renderer and it is able to get to all the noteheads and use them to build the complete image.

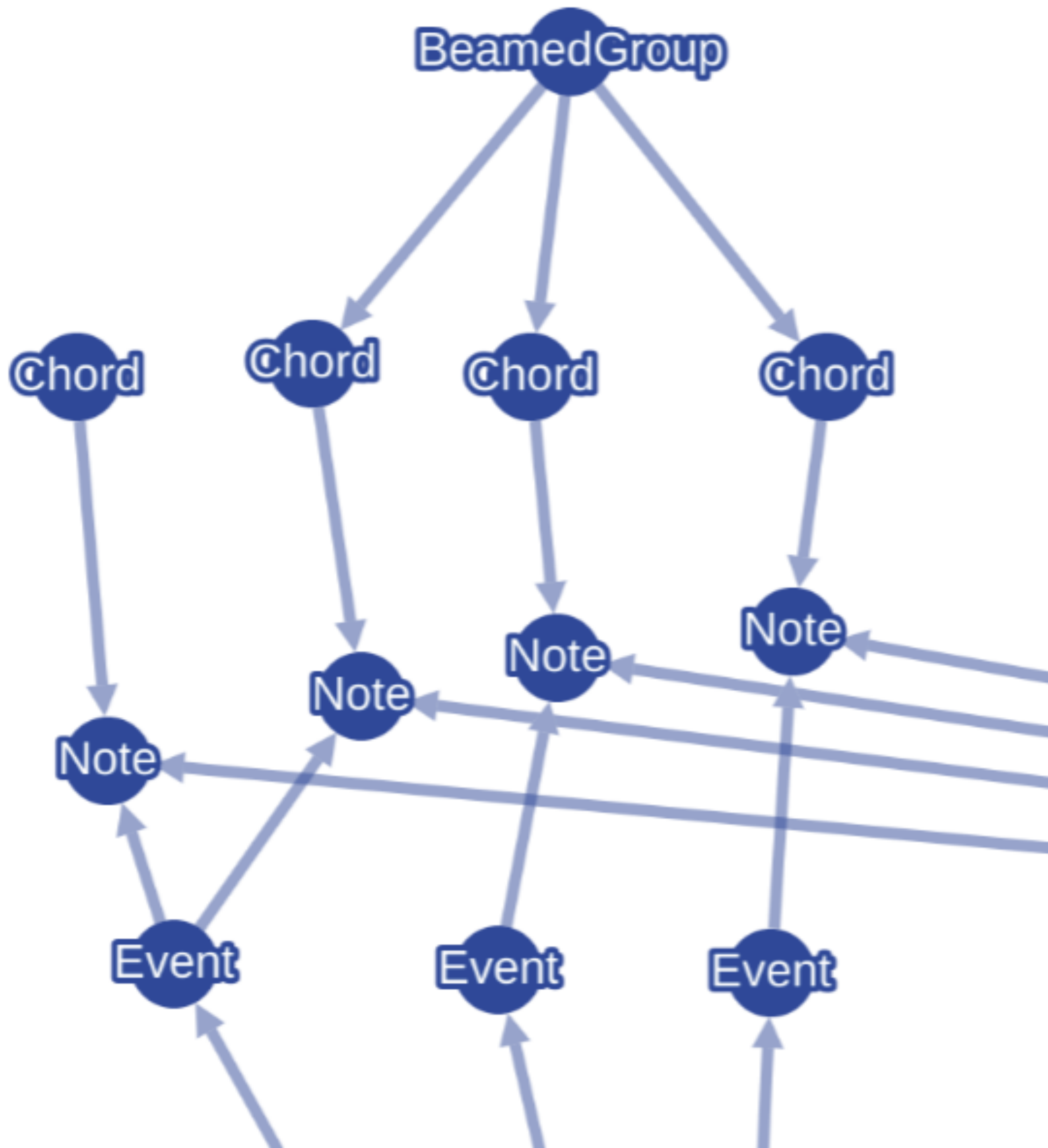
4.1.1 Containers

It also allows us to focus on the semantics of the data and don't let the technical problems get in the way. For example, in music, notes may belong to lots of different containers:

- `Event` Notes with the same onset.
- `Chord` Notes with a shared stem, same onset, and same duration.
- `Beam` Notes sharing a beam.
- `Voice` Notes in the same voice in polyphonic music.
- `Slur` Notes tied by the same slur.
- `Tuplet` Notes being part of the same tuplet.

It's the responsibility of the container to know what notes it contains, NOT the note's to remember what container it belongs to. If it was the note's responsibility, then each time we add another container, we have to extend the `Note` class. That's ugly and hinders extensibility.

At the same time, when synthesizing the music notation, the synthesizer must be able to ask a note about the chord it belongs to, about the slur it belongs to, etc. So we need this inverse querying capability. That's why we inherit from the `SceneObject` base class, which keeps track of all these references (most importantly the back-links).



4.1.2 Extensibility

Similar issue arises when we build the Smashcima library, but someone else would like to extend it. Let's say you want to build a notation synthesizer that respects notehead colors. You need to store the notehead color in the scene graph somehow.

In your own project, you would be tempted to modify the `Note` class and just add a `color` field to it. But since the `Note` class is part of the Smashcima library and you cannot change the library, you cannot do that.

Another option could be to create a `ColoredNote` class, which inherits from the `Note` class. But this also poses a few problems:

1. How do you construct the `ColoredNote` instance, if the kern loader returns `Note` instances? And if you

just copy all the values around, how do you know it won't break with a future update to Smashcima that adds new fields you didn't expect to be added?
2. What if a third person wants to also represent notehead shapes (square, slash, cross)? They would build a `ShapedNote` class. But what does the inheritance chain look like now? Does the `ShapedNote` inherit from `ColoredNote` or the other way around? And what if you two don't even know about one another, but a third person wants to use both of your extensions?

You can see how inheritance quickly leads to unmaintainable software in such situations.

Instead, we deal with extensibility in Smashcima by giving you the ability to extend the *scene graph* by adding new nodes and attaching them to existing instances.

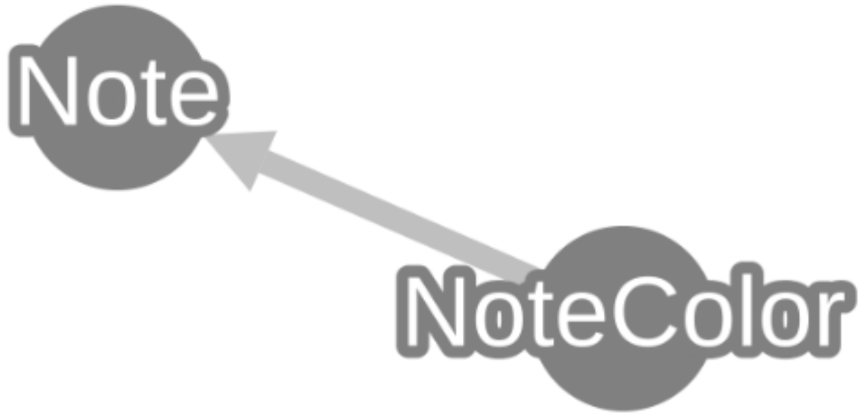
You can define a `NoteColor` scene object and link it to a note:

```
@dataclass
class NoteColor(sc.SceneObject):
    color: str
    """The color of the notehead"""

    note: Note
    """The note for which we define the color"""
```

You can add it to the scene graph simply by calling the constructor and throwing it away - the internal link tracking system will keep it referenced:

```
NoteColor(
    color="red",
    note=my_note
)
```



You can then get the color of a note by doing an inverse query:

```
print(
    NoteColor.of(my_note, lambda c: c.note).color
) # "red"
```

While inheritance is used in Smashcima in some specific cases, where this inverse querying would be unnecesarily verbose, most of the library is designed to use composition over inheritance.

For example, there is no reason why a `Voice` should be inside `Event` or why `Event` should be inside `Voice`. These two concepts are orthogonal (literally) and should not be nested artificially. The graph design of the scene data model allows for this.

4.2 How `SceneObject` works

Now that you have the motivation for the design, we can talk about how this behaviour is implemented.

The `SceneObject` class overrides the `__setattr__` magic method, so it knows about all situations when anyone is setting any field on the instance (`obj.bar = baz`). When this happens, it updates the scene object's `inlinks` and `outlinks` fields. These fields hold the list of references pointing away from this object, and the list of references pointing towards this object. Each link also knows its `name`, which means you can have more than one link between two object instances.

These links are only tracked between pairs of `SceneObjects`. If only one side is a `SceneObject`, then no link is tracked. This means that scene objects can have fields containing `str`, `int`, `Fraction`, or any other type, and these behave like any other pyhon code. It's only when a scene object "contains" another scene object. In that case you cannot really say it "contains" that object. Rather, you should think about "referencing" that object in the scene graph.

4.2.1 Lists and `None`

Scene objects can also reference other scene objects optionally, e.g. having a field with type `Optional[Note]`. The type annotation is ignored by the scene object, but the fact of setting the reference to `None` deletes the link from the scene graph.

Similarly, a scene object can reference a list of objects (e.g. a `Voice` references a list of `Notes`). When a `SceneObject` is assigned a `list` instance, the logic goes through the list and creates a graph link to all of the contained scene objects (while ignoring plain python types).

Note that, since all of this logic happens in `__setattr__` magic method, the system does not pick up `.append` or `.pop` or `del list[4]` or `+= [item]` invocations. If you want to modify a list of scene objects, you have to construct a new list and then set it using `obj.list = new_list`! In other words, you should treat the list as being immutable.

> **Warning:** If you do call `.append` on a list of scene object, you will de-synchronize the graph links from the python instance values and this will result in unexpected behaviour when querying those links. There is no safety logic that would detect that!

4.2.2 Memory leaks

Because scene objects track these back-references, it's very easy to leak memory, since objects are being referenced even when they would not be in plain python code. For example, creating the `NoteColor` instance in this way makes it attached to the `Note` instance and it will never be garbage collected, unless the `my_note` instance can also be garbage collected at the same time:

```
my_note_color = NoteColor(
    color="red",
    note=my_note
)
```

4.2.3 Detaching objects from scene

Because of this linking behaviour, if you want to throw away part of the scene (to replace it, or just to completely delete it), you cannot just forget about it. You have to also break its references to the rest of the scene.

For example, to destroy the `NoteColor` instance we created earlier, we need to set its reference to the `Note` to `None`:

```
my_note_color.note = None
```

Only now will the `my_note_color` instance be garbage collected.

Because this operation makes sense for many scene objects, but the way in which to achieve it depends on the semantics (you need to know, which reference is *the* reference that links this sub-scene to the rest of the scene), you must implement this logic manually (it cannot be provided by default by the `SceneObject` class).

When you encapsulate this detachment logic into a method, the convention is to call this method `detach()`:

```
@dataclass
class Notehead(sc.SceneObject):
    note: Note
    image: np.ndarray

    def detach(self):
        """Break all links with the rest of the scene"""
        self.note = None # type: ignore
```

Also note that you cannot really *destroy* a python class instance. Only the garbage collector can do that. And only when no live instances point to it. Doing `del my_note_color` only deletes the local variable, *not* the instance. You can still get hold of the instance via `my_note.inlinks`. That is why a dedicated `detach()` method that breaks these links is necessary.

4.3 Relationship querying

Since a `Notehead` points to a `Note`, traversing this forward link is as simple as accessing a python field:

```
# get the Note of a Notehead
my_note = my_notehead.note
```

The special syntax is only needed when we do the reverse querying:

```
# get the Notehead of a Note
my_notehead: Notehead \
    = Notehead.of(my_note, lambda n: n.note)
```

The `.of` method is a class-method defined on the `SceneObject`, meaning it will be available on every type inheriting from the `SceneObject` and it always returns the specific type (e.g. `Notehead`).

If the scene link may not exist (and it is ok for it not to exist), you can instruct the query to return `None` in such cases:

```
# get Stem for a Notehead (which may not exist)
my_stem: Optional[Stem] \
    = Stem.of_or_none(my_notehead, lambda: s: s.noteheads)
```

Note that the stem also points to a list of `noteheads` and the query also works as expected.

Lastly, you might want to ask for a list of scene objects on the other side of a link with a given name:

```
# get children of an AffineSpace
children: list[AffineSpace] \
    = AffineSpace.many_of(self, lambda s: s.parent_space)
```

4.3.1 Named queries

While using these generic inverse queries with `lambda` expressions for link names is possible, it is a little bit verbose and hard to read. Instead, when defining a new scene object type, you should provide a set of custom methods that define better names for these queries.

For example, we can extend the `Notehead` class to define the query to get a notehead for a corresponding `Note`:

```
class Notehead(sc.SceneObject):
    # ...

    @classmethod
    def of_note(cls, note: Note):
        return cls.of(note, lambda n: n.note)
```

Now we can get the notehead with much shorter and more readable code:

```
# get the Notehead of a Note
my_notehead = Notehead.of_note(my_note)
```

Similar method could be added for stems:

```
class Stem(sc.SceneObject):
    # ...

    @classmethod
    def of_notehead_or_none(cls, notehead: Notehead):
        return cls.of_or_none(notehead, lambda s: s.notes)
```

When doing this, please keep the terminology consistent:

- `X.of_Y` to get `X` that should always exist for `Y` and should raise an exception if missing
- `X.of_Y_or_none` to get `X` that may sometimes not exist and should return `None` in such case
- `X.of_many_Y` to get a `list[X]`, which may also be an empty list

4.4 When to use the `Scene` class

So far, all scene objects lived in custom variables as python instances. You don't really need a container (such as a `Scene`) when creating scene objects. However sometimes you want to return a disjoint group of `SceneObjects` as a single object. For that purpose there exists the `sc.Scene` type.

One place where this type might be used is when defining a return type of a `Model` type. Models are supposed to return scenes - the specifics of the scene depend on the model and the domain it models, but the scene class can inherit from `sc.Scene` to utilize some of its pre-defined logic (like recursive scene object addition and closure addition).

You can add scene objects into a scene like this:

```
scene = sc.Scene(
    root_space=sc.AffineSpace()
)

scene.add(my_note)
```

The scene holds one `AffineSpace` as a root affine space, which may be used as a starting point when traversing visual scene objects.

Objects added into a scene are added with all other linked objects (to and from).

We can also check that the added note also added its notehead:

```
scene.has(my_notehead) # returns True
```

> **Note:** The precise responsibility of the `sc.Scene` class is not quite defined and it might happen that it gets sharpen, modified, or dropped in the future. It's partly a relict from times when scene objects needed to be in a container.

4.5 Conclusion

This documentation page talks about `SceneObject`s in the abstract. But that's just the very core of Smashcima. You then also have a library of already pre-defined scene objects with fixed meaning that can be used to describe a synthetic sheet of music. The rest of the scene documentation talks about those.

Chapter 5

Affine spaces and rendering

In the previous documentation section we talked about scene objects and how they can represent arbitrary graph-like data. We used the concepts of a `Note` and a `Notehead` to represent semantic and visual objects. In this documentation section we will describe how the visual portion of a scene is described and rendered (delving into how a `Notehead` visual object might be actually implemented in Smashcima).

5.1 Affine space

Most 2D computer graphics software is built on the concept of a 2D affine space. This mathematical construct is used in both vector and raster computer graphics software, including Inkscape, Krita, Photoshop, Illustrator, Figma and even the SVG vector graphics data format.

In Smashcima we imagine an affine space as a 2D coordinate system, with an origin somewhere in space and coordinates defined by two basis vectors X and Y sitting at the origin. The two basis vectors can have any length and any direction, although in the majority of cases they have unit length and are orthogonal.

The affine space lets us put 2D real-valued coordinates on the underlying geometric space.

All visual objects that exist in Smashcima must be situated in some affine space (otherwise the concept of their position (and size and orientation) cannot be defined). These objects include:

- `Sprite` a bitmap image
- `ViewBox` a rectangular viewport into the scene
- `ScenePoint` a 2D point
- `Region` a polygonal area outline in the scene

For example, when you have a `Point` it consists of two coordinates: X and Y. But these coordinates have no meaning on their own. The point has to be placed into an `AffineSpace`, that defines what these numbers mean in spatial terms. Therefore, a `ScenePoint` is nothing more than a `Point` combined with an `AffineSpace` reference.

5.1.1 Space hierarchy

Each affine space can be placed into another existing affine space as a child. Its position within the parent space is defined by an affine `Transform`. An affine transform is just a linear transform (rotation, scale, skew, mirror) that also allows for translational movement (translation).

In other words, in the parent space, the child space's origin can be placed anywhere and the child space can be deformed and rotated in any way that keeps its coordinate grid as straight lines that are evenly spaced.

This nesting of affine spaces lets us take a subset of the scene's visual objects (attached to the child space) and place them anywhere within the parent space as one piece. This means that, for example, a glyph synthesizer

can operate in the glyph's local affine space, placing all the glyph strokes properly and then a music notation synthesizer can position the glyph as a whole on a staff.

An affine space that has no parent (is set to `None`) is called the root affine space, and there should only be one such space in a well-defined scene. Existence of a shared root space ensures that for any two scene objects in any two spaces, there exists a nearest ancestor space containing both of them, which allows us to translate between these two object's coordinate spaces.

5.1.2 Constructing affine spaces

You can create a root affine space like this:

```
import smashcima as sc

root_space = sc.AffineSpace()
```

Then you can create a child space, with its origin placed at `(10, 20)` like this:

```
child_space = sc.AffineSpace(
    parent_space=root_space,
    transform=sc.Transform.translate(sc.Vector2(10, 20))
)
```

If no transform is proivded, then `Transform.identity()` is used automatically.

Alternatively, if a child space was constructed before and it has no parent (or we want to change its parent), we can place it under the root space like this:

```
other_child_space.parent_space = root_space

# optionally, you can modify the transform
other_child_space.transform = sc.Transform.identity()
```

5.1.3 Understanding transforms

The affine space's transform is formally defined as a `2x3` matrix that maps from the child's coordinate system into the parent's coordinate system.

In other words, with the parent-child setup from above, with the root space and a child space placed at `(10, 20)`, if we take a point at `(-1, 0)` in the child space, and apply the child's transform to it, we will get `(9, 20)`, which is the point's coordinates in the root space:

```
point_in_child_coords = sc.Point(-1, 0)
point_in_root_coords = child_space.transform.apply_to(
    point_in_child_coords
)
print(point_in_root_coords) # prints (9, 20)
```

You can chain multiple transforms by using the `.then` method. For example, we could have the child space use the same origin at `(10, 20)` but be rotated 180 degrees. That would put our sample point at `(11, 20)` in the root coordinates:

```
child_space.transform = sc.Transform.rotateDegCC(180) \
    .then(sc.Transform.translate(sc.Vector2(10, 20)))

# now probing the same point, in a 180deg rotated child space:
point_in_child_coords = sc.Point(-1, 0)
point_in_root_coords = child_space.transform.apply_to(
    point_in_child_coords
```

```
)
print(point_in_root_coords) # prints (11, 20)
```

The order of `.then` matters. We are mapping from the child space into the parent space, so we need to first perform the rotation (in the perspective of the child space), and only after that do the translation to the correct position in the parent space.

> **Remember:** Transforms always map from child space coordinates to the parent space coordinates.

5.1.4 Units are millimeters

So far affine spaces have been defined purely mathematically without any units. While Smashcima does not track any units and you can treat the numeric values however you like, the assumption is that one unit is one millimeter.

This is because we are mostly dealing with scales in the range of a piece of paper, where a millimeter is an appropriate unit (used by many 2D computer graphics software tools).

Smashcima uses physical spatial units, as opposed to pixels, because it aims to harmonize data from various scanned source datasets, which may have been rasterized at various DPIs. Sticking to millimeters is a way to reconcile these differences.

To convert between millimeters and pixels with a given DPI, you can use these utility functions:

```
print(sc.mm_to_px(1, dpi=300)) # 12.295081967213116
print(sc.px_to_mm(1, dpi=300)) # 0.08133333333333333

# one millimeter is about 12.3 pixels under 300 DPI
```

5.2 Sprites

Now that we have the space itself covered with coordinate systems, we need to place some objects into it. Because most handwritten musical symbol datasets use raster images, we built the Smashcima visual rendering system on raster images as well.

A `Sprite` is a raster image, placed somewhere in an `AffineSpace`, with well-defined scale.

You can create a sprite like this:

```
import numpy as np

sprite = sc.Sprite(
    space=root_space,
    bitmap=np.array([...]),
    bitmap_origin=sc.Point(0.5, 0.5),
    dpi=300,
    transform=sc.Transform.identity()
)
```

The `space` is the affine space that the sprite is placed into (the parent space).

The `bitmap` is an OpenCV BGRA image with the `uint8` depth (a 3D numpy array).

The `bitmap_origin` is a 2D point in the `0.0 - 1.0` range in each coordinate, specifying where the bitmap should be overlayed with the parent space's origin. Values of `0.5` mean the center of the bitmap. So our bitmap will be centered on the space origin.

The `dpi` controls the physical size of the bitmap in the parent space. At what DPI has the bitmap been scanned.

The `transform` lets you apply an additional transform to the bitmap to position it arbitrarily in the parent space. For example, you can use a rotation transform to rotate the image.

So far I described the `bitmap_origin` and `dpi` as controlling the placement within the parent space. This is not strictly correct. There is a so-called origin space, and these arguments control the placement of the bitmap within the origin space. The `transform` property then controls the placement of the origin space within the parent space. The nesting is as follows:

```
[parent space]
      A
      |    <- controlled by `transform`
      |
[origin space]
      A
      |    <- controlled by `bitmap_origin` and `dpi`
      |
[pixel space]
```

Pixel space is the affine space, where `(0, 0)` is the top-left corner of the bitmap and `(w, h)` is the bottom-right corner of the bitmap, where `w` and `h` are width and height in the number of pixels.

You can get the lower transform by calling `sprite.get_pixels_to_origin_space_transform()` and you can get the upper transform simply by accessing `sprite.transform`. You can also get the joint transform by calling `sprite.get_pixels_to_parent_space_transform()`.

There is also a number of properties you can access about a sprite:

- `.pixel_width` and `.pixel_height` are size in the number of pixels
- `.physical_width` and `.physical_height` are size in millimeters
- `.pixels_bbox` is a helper property that returns a `Rectangle` in pixel space positioned at `(0, 0)` and with size `(pixel_width, pixel_height)`.

5.3 View boxes

Now that we have the coordinate spaces and bitmap images placed in them, we need a rectangular window that will be used as the camera that looks into the scene. This camera-like object is called the `ViewBox`. It's simply a `Rectangle` placed in an `AffineSpace`:

```python
view_box = sc.ViewBox(
    space=root_space,
    rectangle=sc.Rectangle(0, 0, 210, 297) # A4 paper in millimeters
)
```

5.4 Bitmap renderer

With the camera defined, we can now use a `BitmapRenderer` to traverse recursively all the affine spaces, find all sprites, transform them into a blank canvas and layer them on top of each other:

```python
renderer = sc.BitmapRenderer(
    dpi=300, # rasterize the scene at this DPI
    background_color=(0, 0, 0, 0) # BGRA
)
img = renderer.render(view_box)

# img is a np.ndarray OpenCV BGRA uint8 image
```

The renderer is given only the `view_box` but since it links to its affine spaces and affine spaces link to their parents, we can find the root space and iterate from there.

The whole visual scene hangs on the root space and is not garbage collected, because of the double-linking tracked by `SceneObject`s discussed in the previous documetation section.

> **NOTE:** View boxes currently assume they are placed in the root affine space. It should be supported that they can be placed in any sub-space. But they must still render the whole scene from its root space. (Specifically, it is not view boxes that assume that, it is the BitmapRenderer that assumes that.)

Chapter 6

Synthesizer interfaces

Synthesizer is any service that creates or manipulates a scene to produce synthetic data. Its API therefore completely depends on the synthetic data it produces.

Whereas a model comes pre-configured and ready to use to synthesize final data, a synthesizer should have just one narrow responsibility and allow for maximal configuration. The less work it does, the more it can be used as a LEGO piece in a larger synthesis pipeline.

To give some structure to the resulting synthesis pipelines and to allow for intechangebility of synthesizers, Smashcima defines a set of synthesizer interfaces, together with their implementations. This documentation page goes over these interfaces.

6.1 Smashcima synthesizers

This is an overview list of synthesizer interfaces in the order from the most abstract to the most concrete:

- `ColumnLayoutSynthesizer` places musical symbols onto empty stafflines
- `PageSynthesizer` produces a sheet of paper with empty stafflines with a given layout
- `StafflinesSynthesizer` produces empty stafflines
- `PaperSynthesizer` produces images of sheets of paper
- `LineSynthesizer` produces line-like music symbols (beams, stems, braces)
- `GlyphSynthesizer` produces musical symbols (notes, rests, accidentals)

6.2 Using synthesizers

If you look inside `BaseHandwrittenModel.call()` method, you can see that the model uses two synthesizers:

- `PageSynthesizer`
- `MusicNotationSynthesizer`

It loads a music score and then until there are measures remaining, it creates a new page and then fills it with measures from the score.

Looking at it from this top level makes the `BaseHandwrittenModel` a pretty thin and simple class.

6.3 Extensibility

If you're building a synthesizer for a domain not covered by these interfaces, feel free do define your own interfaces and then implement them. Having interfaces explicitly named, documented, and assigned to

implementations in models lets other developers to then create alternative implementations to them. Embrace the LEGO piece modularity of Smashcima.

Chapter 7

`ColumnLayoutSynthesizer` interface

This interface represents the (human) writer sitting at a blank piece of paper with stafflines, transcribing a piece of music onto that paper.

To create images of music symbols (glyphs), it uses as a dependency a `GlyphSynthesizer` (and a `LineSynthesizer`). Its responsibility is to create these based on the input musical score and position them on the piece of paper according to the rules of common western music notation.

7.1 Public API

It defines two methods: `fill_page` and `synthesize_system`.

They both fill the page with music you give to them, but at two leves of control - at the page level, or at the system level.

The low-level system method has this signature:

```
def synthesize_system(
    page_space: AffineSpace,
    staves: List[StaffVisual],
    score: Score,
    start_on_measure: int
) -> System:
```

You give it a list of empty staves (stafflines), whose count must match the number of staves in the music score and also the affine space that contains these empty staves. Then you give it the music score and the measure index from which it should start transcribing the music. You get back a system of music notation (system = one line with all instruments).

The high-level method has this signature:

```
def fill_page(
    page: Page,
    score: Score,
    start_on_measure: int
) -> List[System]:
```

You give it a page with empty stafflines and a music score plus a measure index from which to start transcribing, and it fills the page up with music notation.

The interface also defines these public `bool` flags that you can modify after the synthesizer is instantiated to control the music notation flow:

- `.stretch_out_columns` Analogous to text alignment, when true it behaves like "text justify" and when false, it behaves like "text align left".
- `.respect_line_and_page_breaks` When true, systems are terminated at line and page breaks in the music score. When false, these breaks are ignored.
- `.disable_wrapping` When true, music does not wrap to the next system when it starts overflowing. When false, it does.

7.2 Implementations

- `ColumnLayoutSynthesizer`

Chapter 8

`GlyphSynthesizer` interface

Glyph synthesizer represents a service intended to produce glyphs of music notation. These are, for example, noteheads, flags, accidentals, rests, etc.

8.1 `Glyph` scene object

Glyph is a visual scene object that links together other spatial scene objects that together contain information about a music notation symbol.

There are three types of information that a glyph holds:

- appearance (as a list of `Sprites`)
- classification label (as a `str`)
- semantic segmentation mask (as a `Region`)

It's a set of sprites extended with the information necessary to perform classification, object detection, and semantic segmentation on the symbol.

All of these objects are spatial in its nature, so the glyph instance also carries its owin `AffineSpace` instance, inside of which all of these objects live and which should be used to attach the glyph into the broader scene.

A glyph instance can be created like this:

```
import smashcima as sc

# new space for the glyph to live in
space = sc.AffineSpace()

# classification label for the glyph
label: str = sc.SmuflLabels.noteheadBlack.value

# dummy sprite to represent the glyph, placed in the space
sprite = Sprite.rectangle(
    space=space,
    rectangle=sc.Rectangle(-1, -1, 2, 2)
)

# create the glyph as a collection of these objects
glyph = sc.Glyph(
    space=space,
    region=sc.Glyph.build_region_from_sprites_alpha_channel(
```

```
        label=label,
        sprites=[sprite]
    ),
    sprites=[sprite]
)
```

8.1.1 Placing a glyph into a scene

The code above creates a glyph that stands outside the greater scene. Its affine space is a root space (has no parent). To place it into another space, just attach the affine space like this:

```
# some affine space of the whole scene
# (can be a root, can be only a staff or the paper space)
root_space = sc.AffineSpace()

# attach the glyph into the scene space at (10, 20)
glyph.space.parent_space = root_space
glyph.space.transform = sc.Transform.translate(sc.Vector2(10, 20))
```

8.1.2 Glyph origin point

You can see that placing the glyph into a scene is performed by placing its affine space. More specifically, by placing its affine space's origin point.

For this reason it's important to be consistent in where exactly this origin point is located for each glyph type (e.g. noteheads have it as their center, whole rests have it as the position of the staffline).

The list of glyph labels and their proper origin points is documented in the Glyphs documentation page.

8.1.3 Semantics of visual objects

The `Glyph` scene object only represents a visual glyph in the scene, but it carries no semantic information. We would like to have a `Notehead` that will have a link to its semantic `Note` and contain additional links to, for example, the `StaffVisual`.

You should **NOT** do this via inheritance! The documentation section on scene objects details the complications you would run into. Instead, you create a `Notehead` scene object and add a reference to the glyph and other scene objects:

```
from dataclasses import dataclass

@dataclass
class Notehead(sc.SceneObject):
    glyph: Glyph
    notes: List[Note]
    staff: StaffVisual

    @classmethod
    def of_glyph(cls, glyph: Glyph):
        return cls.of(glyph, lambda n: n.glyph)
```

Using the reverse reference queries, you can get the notehead for a glyph (if you know that the glyph is a notehead) and you don't need to inherit or extend the `Glyph` class in any way:

```
notehead = Notehead.of_glyph(glyph)
```

8.2 Public API

Now that you have an indea of what a `Glyph` is, we can look at how to use a `GlyphSynthesizer` service.

To just simply create a new glyph scene object, you can call the `create_glyph` method:

```
def create_glyph(label: str) -> Glyph:
```

You just give it the desired glyph class and it will construct a new glyph instance for you in the fashion very similar to the code we've seen at the beginning of this documentation page.

The returned glyph will have its own `AffineSpace`, which will have no parent. You can now do with the `Glyph` whatever you desire.

But since you often want to immediately place the glyph into some parent space, you can directly ask the synthesizer to do that for you using `synthesize_glyph`:

```
def synthesize_glyph(
    label: str,
    parent_space: AffineSpace,
    transform: Transform
) -> Glyph:
```

You specify the parent space and the transform that specifies the glyph's placement within the parent space. It also performs some additional argument checks that the previous method might not do.

Lastly, since the glyph is usually only translated, but only rarely rotated or scaled, there's the method `synthesize_glyph_at` which takes a `Point` instead of a full `Transform`:

```
def synthesize_glyph_at(
    label: str,
    parent_space: AffineSpace,
    point: Point
) -> Glyph:
```

8.2.1 Checking supported glyph labels

Since not all glyph synthesizers support all glyph classification labels, there's a method that you can use to check that the synthesizer you are about to use supports all the labels you will be asking of it:

```
def supports_label(label: str) -> bool:
```

If you're building a synthesizer that in-turn uses a `GlyphSynthesizer` as a dependency, make sure to call this method before synthesizing. This helps detect compatibility issues before they have a chance to manifest in production (for instance after 2 hours of runtime once the input data suddently contains a triple-flat accidental but the glyph synthesizer you have been using so far does not support them).

8.3 Implementing a glyph synthesizer

To implement a custom glyph synthesizer, simply inherit from the `GlyphSynthesizer` abstract base class and override the `supports_label` and `create_glyph` methods. The other two methods are already implemented for you and they internally use the `create_glyph` method you provide:

```
class MyFancyGlyphSynthesizer(sc.GlyphSynthesizer):
    def supports_label(self, label: str) -> bool:
        return True # yes, we support everything!

    def create_glyph(self, label: str) -> Glyph:
        # create a glyph from scratch
        # (or load it from some pickle and deep copy,
```

```
    # that's also an option)
    glyph = Glyph(...)

    return glyph
```

8.4 Implementations

- `smashcima.synthesis.glyph.MuscimaPPGlyphSynthesizer` Synthesizes glyphs by sampling from the MUSCIMA++ dataset.

Chapter 9

Glyphs

This documentation page lists all `Glyph` classification labels explicitly known about by Smashcima and defines their origin point position.

This does not contain `LineGlyphs`. They are documented in a separate page.

9.1 Clefs

- `smufl::cClef`, `smufl::cClefSmall`
  - Origin = vertically the defining staffline, horizontally the sprite center
  -
- `smufl::fClef`, `smufl::fClefSmall`
  - Origin = vertically the defining staffline, horizontally the sprite center
  -
- `smufl::gClef`, `smufl::gClefSmall`
  - Origin = vertically the defining staffline, horizontally the sprite center

  –

9.2 Time signatures

- `smufl::timeSig0`, `smufl::timeSig1`, `smufl::timeSig2`, `smufl::timeSig3`, `smufl::timeSig4`, `smufl::timeSig5`, `smufl::timeSig6`, `smufl::timeSig7`, `smufl::timeSig8`, `smufl::timeSig9`
  – Origin = center of the sprite

  –
- `smufl::timeSigCommon`, `smufl::timeSigCutCommon`
  – Origin = center of the sprite
  –

9.3 Noteheads

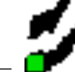
- `smufl::noteheadWhole`, `smufl::noteheadHalf`
  – Origin = center of the notehead
  –
- `smufl::noteheadBlack`
  – Origin = center of the notehead
  –

9.4 Augmentation dot

- `smufl::augmentationDot`
  – Origin = center of the dot
  –

9.5 Flags

- `smufl::flag8thDown`

  - Origin = tip of the corresponding stem
  -
- smufl::flag8thUp
  - Origin = tip of the corresponding stem
  -
- smufl::flag16thDown
  - Origin = tip of the corresponding stem
  - 
- smufl::flag16thUp
  - Origin = tip of the corresponding stem
  -

9.6 Accidentals

- smufl::accidentalFlat
  - Origin = center of the eye
  -
- smufl::accidentalNatural
  - Origin = center of the eye
  -
- smufl::accidentalSharp
  - Origin = center of the eye
  -
- smufl::accidentalDoubleSharp
  - Origin = center of the cross
  -

9.7 Articulation

- `smufl::articStaccatoBelow`
  - Origin = center of the dot
  -

9.8 Rests

- `smufl::restWhole`
  - Origin = vertically the defining staffline, horizontally the sprite center
  -
- `smufl::restHalf`
  - Origin = vertically the defining staffline, horizontally the sprite center
  -
- `smufl::restQuarter`
  - Origin = center of the sprite
  -
- `smufl::rest8th`
  - Origin = center of the sprite
  -
- `smufl::rest16th`
  - Origin = center of the sprite
  -

Chapter 10

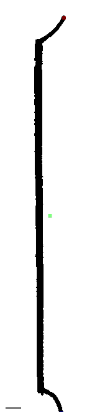
Line glyphs

This documentation page lists all `LineGlyph` classification labels explicitly known about by Smashcima and defines their start, end, and origin point positions.

Legend: - Red = start point - Green = origin point (of the glyph's affine space) - Blue = end point

10.1 List of line glyphs

- `smufl::stem`
  - Start point = bottom tip of the line
  - End point = top tip of the line
  - Origin = undefined, can be anywhere (default to sprite center)
  -
- `smashcima::beam`
  - Start point = left tip of the line
  - End point = right tip of the line
  - Origin = undefined, can be anywhere (default to sprite center)
  -
- `smashcima::beamHook`
  - Start point = left tip of the line
  - End point = right tip of the line
  - Origin = undefined, can be anywhere (default to sprite center)
  -
- `smashcima::ledgerLine`
  - Start point = left tip of the line
  - End point = right tip of the line
  - Origin = undefined, can be anywhere (default to sprite center)
  -
- `smufl::bracket`
  - Start point = top tip of the bracket
  - End point = bottom tip of the bracket
  - Origin = undefined, can be anywhere (default to sprite center)



–

- `smufl::brace`
  - Start point = top tip of the brace
  - End point = bottom tip of the brace
  - Origin = undefined, can be anywhere (default to sprite center)

  –