Smashcima User 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).

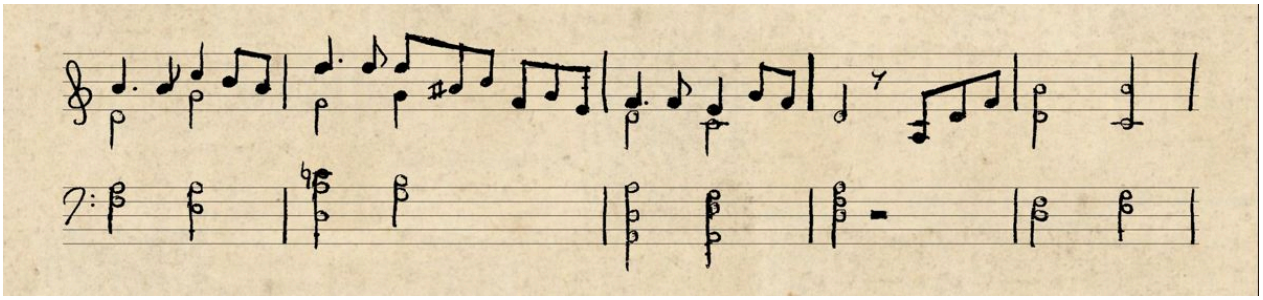
Contents

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 meant for OMR developers who want to use Smashcima to produce synthetic data for their use case. That includes adapting it to the visual domain with new annotations of music notation glyphs. (The technical documentation, on the other hand, is meant for those who want to contribute to the development of Smashcima and extends its capabilities beyond the glyph-based paradigm that it currently serves.)

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

Producing music notation images

This tutorial shows you by-example how you can use Smashcima to generate synthetic data.

2.1 Context

You typically want to use synthetic training data to solve a specific problem. You might have an existing evaluation dataset and you know what it looks like. You'd like to create synthetic data that mimics that target evaluation dataset, so that you can train a machine learning model on it.

Smashcima works with the notion of a `Model`. A model is a function that, when invoked, produces a synthetic training sample. It's called a model, because it's a *generative model*, describing the resulting synthetic dataset.

Smashcima then is a framework for building custom models, but it also comes with pre-defined models that you can use as-is, or just configure to bend them to your needs.

2.2 Invoking a model

The core model that comes with smashcima is the `BaseHandwrittenModel`. It uses the MUSCIMA++ dataset symbols to produce synthetic handwritten music notation.

The model takes in musical content in the form of MusicXML and produces the corresponding number of music notation pages. You can test it on this example music score taken from the OpenScore Lieder corpus.

Download the MusicXML score and run this code:

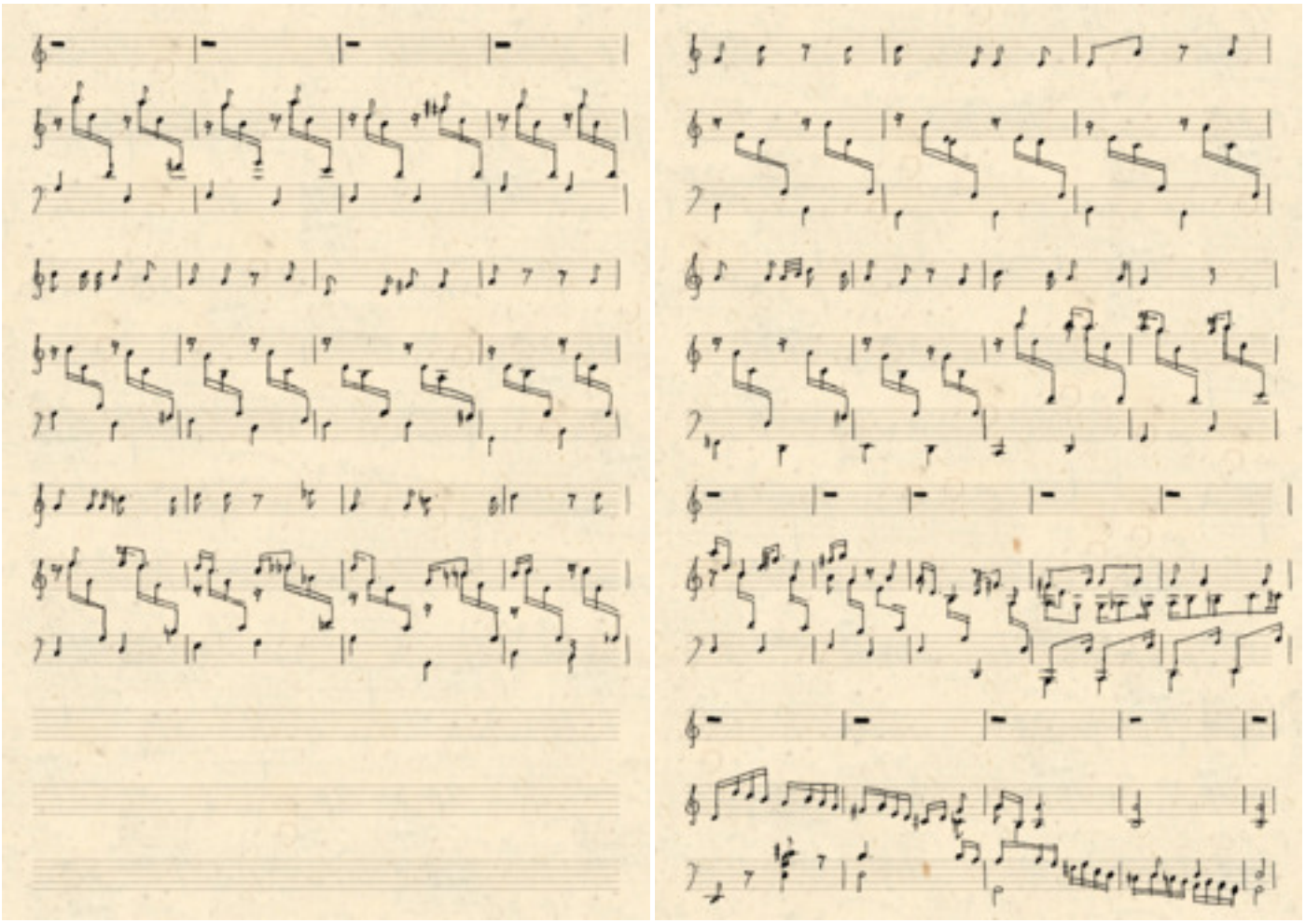
```
import cv2
import smashcima as sc

model = sc.orchestration.BaseHandwrittenModel()
scene = model("lc5003150.musicxml")

for i, page in enumerate(scene.pages):
    bitmap = scene.render(page)
    cv2.imwrite(f"page_{i}.png", bitmap)
```

First, smashcima starts downloading assets it needs for the synthesis. This happens only during the first invocation. Then these assets are reused, because they are stored in the user's cache directory (`~/.cache/smashcima` on linux).

```
[Smashcima Assets]: Installing bundle MuscimaPP...
Downloading MUSCIMA++ dataset...
```

```
Downloading https://lindat.mff.cuni.cz/repository/xmlui/bitstream/handle/11372/LRT-2372/MUSCIMA-pp_v1.0.
and saving it to /home/jirka/.cache/smashcima/assets/MuscimaPP/MUSCIMA-pp_v1.0.zip
100%|++++++++++| 21.8M/21.8M [00:17<00:00, 1.26MiB/s]
Extracting the zip...
Checking bundle directory structure...
[Smashcima Assets]: Bundle MuscimaPP installed.
[Smashcima Assets]: Installing bundle MuscimaPPGlyphs...
100%|++++++++++| 140/140 [02:17<00:00,  1.02it/s]
Writing... /home/jirka/.cache/smashcima/assets/MuscimaPPGlyphs/symbol_repository.pkl
[Smashcima Assets]: Bundle MuscimaPPGlyphs installed.
[Smashcima Assets]: Installing bundle MzkPaperPatches...
Downloading MZK paper patches...
100%|++++++++++| 10/10 [00:06<00:00,  1.52it/s]
[Smashcima Assets]: Bundle MzkPaperPatches installed.
```

The two pages are synthesized, with the layout information preserved (measures per system, systems per page):



This model by default rasterizes the scene at 300 DPI, so the two resulting images are both 2582x3652 pixels.

2.3 The scene

What inputs and outputs the model has is completely up to the model, since this depends on the domain it generates. For exmaple, you could build a model that creates MusicXML data out of thin air, in which case it would have signature `model(void) -> str`. But since smashcima focuses primarily on visual data, we call the value returned from the model a *Scene*.

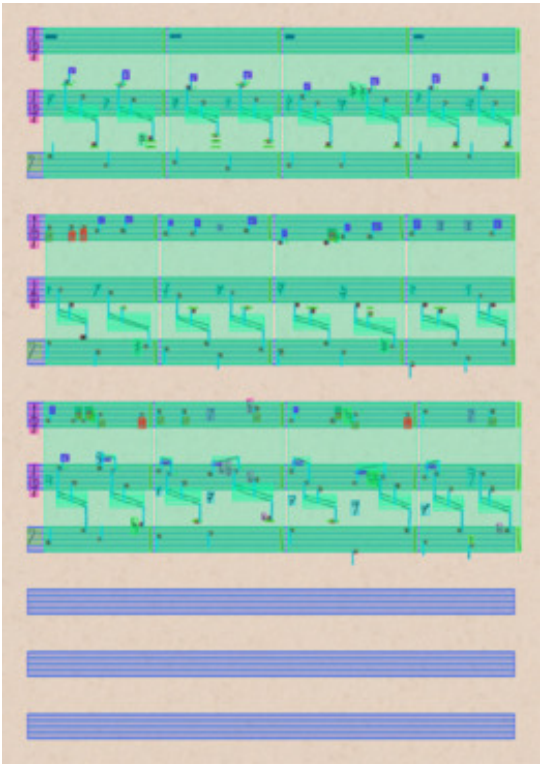
Notice that the model does not return the `np.ndarray` bitmaps directly, instead it returns a custom object that contains much more data than just the images. In fact, it contains almost all the data imaginable in the form of a graph of so-called *scene objects*. This includes the individual glyphs, their masks, bounding boxes, classification labels, stafflines and also the semantics loaded from the MusicXML file and its mapping onto the visual *scene objects*.

2.4 Exporting

For convenience, the scene defines the `render` method, which just constructs a `BitmapRenderer` exporter and invokes it on the scene, producing the final bitmap. See the `smashcima.exporting` module for a list of available exporters.

For debugging purposes there's also the SVG exporter, which you can invoke on the first page with the labeled regions overlay enabled like this:

```
exporter = sc.exporting.SvgExporter(
    render_labeled_regions=True
)
svg = exporter.export_string(
    scene.pages[0].view_box
)

# write SVG to a file
# (and you can open that in a web browser or Inkscape)
with open("page_0.svg", "w") as f:
    f.write(svg)
```



2.5 Conclusion

You've learned how to use the top-level concepts of smashcima (models, assets, scenes, and exporters). If you want to learn more about each, take a look at the documentation. Otherwise continue with the next tutorial, which will teach you how to modify the `BaseHandwrittenModel`, so that it produces only white or transparent background.

Chapter 3

Changing background texture

This tutorial shows how to modify an existing `Model` class. More specifically, how to change the background texture for the `BaseHandwrittenModel`.

3.1 Standard model use

When using the `BaseHandwrittenModel` as is, you just instantiate and invoke it:

```
import cv2
import smashcima as sc

model = sc.orchestration.BaseHandwrittenModel()
scene = model("my-input-file.musicxml")

for i, page in enumerate(scene.pages):
    bitmap = scene.render(page)
    cv2.imwrite(f"page_{i}.png", bitmap)
```

3.2 Synthesizers

Internally, a `Model` is a collection of *synthesizers* that are configured and connected together to serve a single purpose. Synthesizers come from the `smashcima.synthesis` module and they are responsible for:

- synthesizing page layout (dimensions and staff placement)
- synthesizing staves (empty stafflines)
- synthesizing music notation (musical content onto empty stafflines)
- synthesizing glyphs (individual musical symbols)
- synthesizing paper texture

As you can see, each synthesizer is like a very narrow model, that is designed to be used in conjunction with the other synthesizers, and synthesizes one tiny piece of the entire puzzle.

3.3 Using different paper synthesizer

We can modify an existing model by changing the configuration of its internal synthesizers. The model is configured inside its constructor, so in order to change it, we need to make a child class and override these methods:

```
from smashcima.synthesis import PaperSynthesizer, SolidColorPaperSynthesizer

class MyModel(sc.orchestration.BaseHandwrittenModel):
    def register_services(self):
        super().register_services()

        # for paper synthesizer use the solid color synth,
        # instead of the texture-quilting default synthesizer
        self.container.interface(
            PaperSynthesizer, # when people ask for this
            SolidColorPaperSynthesizer # construct this
        )

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

        # get the paper synthesizer instance
        self.paper_synth: SolidColorPaperSynthesizer \
            = self.container.resolve(PaperSynthesizer)

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

        # and configure paper synthesizer's properties
        # (BGRA uint8 format)
        self.paper_synth.color = (187, 221, 234, 255)
```

The model consists of a group of synthesizers and additional classes that together are called services. These services are registered into, and constructed by a service container, accessible via `self.container`. These services are set up in three methods called by the model constructor:

- `register_services` Registers types into the container, binds them to interfaces.
- `resolve_services` Asks the container to construct services we will use later.
- `configure_services` Configures resolved services instances.

  Explainer: The purpose of a service container is to construct services (also known as *resolving services*). You first tell the container what services it should know about (e.g. *When asked about a car, construct a Toyota Corolla.*). Then you resolve services that the model will use later and store them in the model in `self.my_car`. Finally, you adjust the configuration on the constrctued service instances to suit your needs. When the container resolves a service, it recursively resolves all of its dependencies (constructor arguments), which greatly simplifies the service construction process.

  Note: If you call `.resolve` on the same type multiple times, you only get one instance constructed and then returned repeatedly. Another words, all services are registered as singletons.

To learn more, see the documentation on Models.

Now we can use this new model type to do the synthesis:

```
model = MyModel()
scene = model("my-input-file.musicxml")

for i, page in enumerate(scene.pages):
    bitmap = scene.render(page)
    cv2.imwrite(f"page_{i}.png", bitmap)
```

You can use `(255, 255, 255, 255)` to get white background and `(0, 0, 0, 0)` to get transparent back-

ground.

3.4 Replacing the random number generator

Not all services in the model are synthesizers. For example, most synthesizers need a source of randomness. Therefore there is a `random.Random` instance registered as a service in the container. The instance is also stored on the model in the `self.rng` field. You can check they are the same:

```
import random

model = MyModel()
resolved_rng = model.container.resolve(random.Random)
field_rng = model.rng

assert resolved_rng is field_rng # succeeds!
```

You can replace the random number generator during service registration with your own instance:

```
my_rng = random.Random(42)

class MyRandomModel(sc.orchestration.BaseHandwrittenModel):
    def register_services(self):
        super().register_services()

        self.container.instance(
            random.Random, # when people ask for this
            my_rng # return this
        )

model = MyRandomModel()
assert model.rng is my_rng # succeeds!
```

3.5 Conclusion

You've learned how to configure model sevices for existing models. To learn more, read the documentation on Models and examine the base model you are overriding to see what services it registers and how, so that you know how to modify them. The next tutorial will teach you how to replace the `GlyphSynthesizer` of a model with your own glyph synthesizer.

Chapter 4

Using custom glyphs

This tutorial shows how you can provide your own set of glyphs for synthesis.

4.1 Glyph synthesizer

In the previous tutorial we explored how a *Model* is really just a collection of synthesizers, that are configured and wired together. One of these synthesizers is the `GlyphSynthesizer`. It's responsible for creating `Glyph`s - scene objects that consist of an image, segmentation mask, and a classification label.

The `smashcima.synthesis.GlyphSynthesizer` is just an interface (python abstract base class) with two methods to be implemented:

```
import smashcima as sc

class MyGlyphSynthesizer(sc.GlyphSynthesizer):
    def supports_label(self, label: str) -> bool:
        return # true if we can create this glyph type #

    def create_glyph(self, label: str) -> sc.Glyph:
        return # create requested glyph here #
```

The first method is an introspection API that lets the user check, whether the currently used glyph synthesizer supports all the requested glyph types (classification labels).

Since we're building a dummy synthesizer, we will return `True` always:

```
def supports_label(self, label: str):
    return True
```

4.2 Classification labels

The string argument to both functions is the classification label of the requested glyph. It's allowed to be any string, but Smashcima comes with two sets of pre-defined labels that should suite most use cases.

First is the `sc.SmuflLabels` enum. It contains most of the important classes present in the SMuFL standard. Because the enum is not a string, don't forget to ask for the `.value` when getting the underlying string value. Here's a list of the few common glyph labels:

```
print(sc.SmuflLabels.noteheadWhole.value) # "smufl::noteheadWhole"
print(sc.SmuflLabels.noteheadBlack.value) # "smufl::noteheadBlack"
print(sc.SmuflLabels.restQuarter.value)   # "smufl::restQuarter"
```

Because SMuFL is not really built to describe line-like glyphs (beams, stafflines) and because some glyphs are missing (individual flag strokes), there's also the `sc.SmashcimaLabel` enum which contains these additional glyph labels.

```
print(sc.SmashcimaLabels.beam.value) # "smashcima::beam"
print(sc.SmashcimaLabels.staffMeasure.value) # "smashcima::staffMeasure"
```

4.3 Constructing a glyph

A `Glyph` is a scene object that represents a visual unit of music notation. It carries its own `AffineSpace` which defines the glyph's local coordinate system. Then it contains a list of `Sprite`s, where a sprite is just a raster image (OpenCV BGRA uint8 bitmap as a 3D numpy array) together with its DPI (for scaling) and its placement within an affine space. Lastly it contains a `LabeledRegion`, which is a set of polygons in the affine space, encapsulating some 2D area (i.e. the segmentation mask of the glyph).

Let's explore those pieces one by one.

4.3.1 Affine space

At the core of Smashcima's visual scene objects is the `AffineSpace`. It defines a coordinate system and acts as a parent for various visual scene objects. Affine spaces can be nested in a hierarchy, similar to how most raster graphics software operates (for example SVG's `<g>` group element). Affine space contains an affine `Transform`, which describes the coordinate transformation matrix from this space to the parent's space (another words, it defines how is this space placed within the parent space). The root space has `None` parent and its transform is ignored.

A glyph has its own `AffineSpace` and when it's returned from the synthesizer, it must be a root space (have no parent) and its transform will be set later, when the glyph is positioned in the parent space (so we can leave it at default, which is identity transform).

The scene visual hierarchy has unit-less coordinates, but the convention is to assume that one unit is one millimeter. This is preserved throughout Smashcima, as the affine space hierarchy should ideally only translate and rotate, but not shear, nor scale. The scale of objects should be preserved to keep the unit corresponding to one millimeter. Various internal DPI calculations depend on this assumption.

The coordinate system is assumed to have the X axis as the first dimension, increasing to the right (on the screen) and Y axis as the second dimension, increasing down (on the screen). This is the standard 2D computer graphics setup.

This is how we can create a new affine space:

```
space = sc.AffineSpace()
```

4.3.2 Sprite

Sprite is a raster image, placed in an affine space.

Smashcima uses the OpenCV bitmap format and it always has the BGRA 4-channel, uint8 format. So the numpy array shape is `[height, width, 4]`.

We will create a 3x3 millimeter red circle sprite, centered on the origin of the parent affine space. The sprite will have the resolution of 300 DPI, which translates to 36x36 pixels.

```
import numpy as np
import cv2

# size in pixels
size = int(sc.mm_to_px(3, dpi=300))
```

```
# BGRA bitmap with red centered circle
bitmap = np.zeros(shape=(size, size, 4), dtype=np.uint8)
cv2.circle(
    img=bitmap,
    center=(size // 2, size // 2),
    radius=size // 2,
    color=(0, 0, 255, 255), # BGRA
    thickness=-1
)

# construct the sprite instance
# with the affine space created earlier
sprite = sc.Sprite(
    space=space,
    bitmap=bitmap,
    bitmap_origin=sc.Point(0.5, 0.5),
    dpi=300,
    transform=sc.Transform.identity()
)
```

The `bitmap_origin` value is a 0.0 to 1.0 in X, Y axes, which specifies where does the bitmap has its origin (where in the pixel image should the affine space origin line up). The `dpi` field spcifies the sprite scale, with the origin fixed in place.

The `transform` is an affine transform that allows you to position the bitmap origin in the parent affine space somewhere else, than over the affine space origin. If you want to place the sprite at `(50, 20)` in the affine space, you can provide a translation transform `sc.Transform.translate(sc.Vector2(50, 20))`.

4.3.3 Labeled region

Lastly, the glyph has to specify the segmentation mask with a label, to allow for automatic generation of bounding boxes from any viewport.

This can be done quickly by utilizing the alpha-channel on our sprite. There's a built-in method that uses the `cv2.findContours` method, applied to the 50% thresholded alpha-channel of the sprite. This returns a jagged polygon tracing all the glyph pixels.

```
region: sc.LabeledRegion = sc.Glyph.build_region_from_sprites_alpha_channel(
    label="smufl::noteheadBlack", # we will set the requested label instead
    sprites=[sprite]
)
```

The constructed `LabeledRegion` is created in the same `AffineSpace` as the given sprites live. It is automatically attached under that space:

```
assert region.space is sprite.space # succeeds!
assert region.space is space # succeeds!
```

4.4 Putting it all together

When we take all of this code and put it into the glyph syntehsizer, we get this:

```
class RedCircleGlyphSynth(sc.GlyphSynthesizer):
    def supports_label(self, label: str) -> bool:
        return True

    def create_glyph(self, label: str) -> sc.Glyph:
```

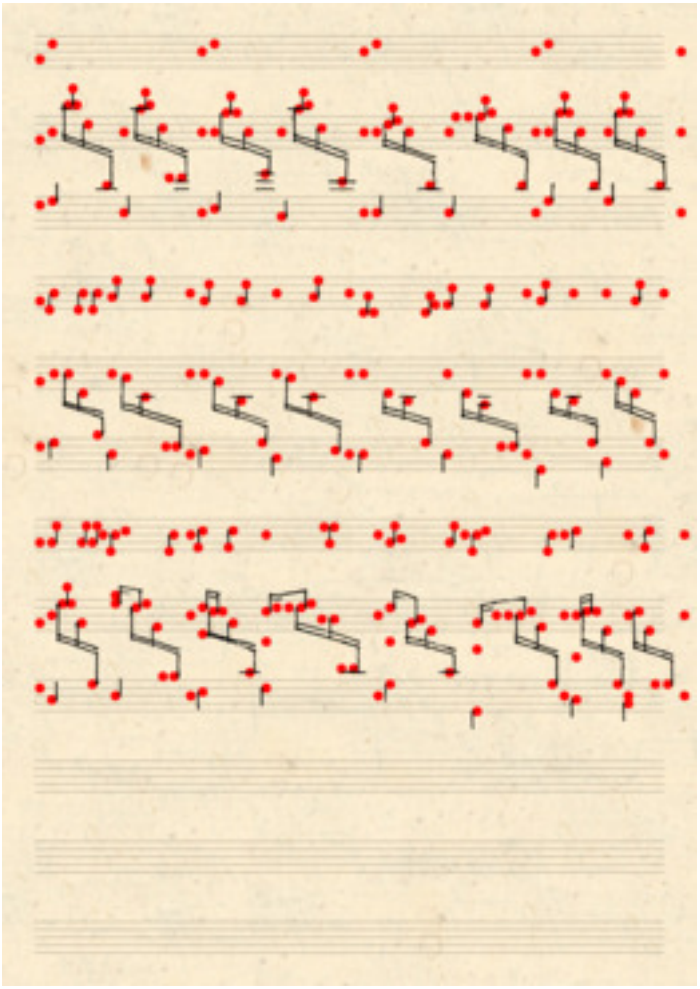
```
        space = sc.AffineSpace()

        size = int(sc.mm_to_px(3, dpi=300))
        bitmap = np.zeros(shape=(size, size, 4), dtype=np.uint8)
        cv2.circle(
            img=bitmap,
            center=(size // 2, size // 2),
            radius=size // 2,
            color=(0, 0, 255, 255), # BGRA
            thickness=-1
        )

        sprite = sc.Sprite(
            space=space,
            bitmap=bitmap,
            bitmap_origin=sc.Point(0.5, 0.5),
            dpi=300,
            transform=sc.Transform.identity()
        )

        return sc.Glyph(
            space=space,
            region=sc.Glyph.build_region_from_sprites_alpha_channel(
                label=label,
                sprites=[sprite]
            ),
            sprites=[sprite]
        )
```

Next, we modify the `BaseHandwrittenModel` to use our glyph synthesizer instead of the default MUSCIMA++ synthesizer:

```
class RedCircleModel(sc.orchestration.BaseHandwrittenModel):
    def register_services(self):
        super().register_services()

        # register a different implementation for the interface
        self.container.interface(
            sc.GlyphSynthesizer,
            RedCircleGlyphSynth
        )
```

Finally, we execute the model:

```
model = RedCircleModel()
scene = model("my-input-file.musicxml")

bitmap = scene.render(scene.pages[0])
cv2.imwrite("red-circles.png", bitmap)
```



4.5 Glyphs and line glyphs

You can see that the resulting image still contains stafflines, beams, and stems.

Stafflines are special, they are synthesized using a `StafflinesSynthesizer` and they completely side-step the `GlyphSynthesizer` interface (because they come with their own non-affine coordinate system and lots of specifics).

Beams and stems are synthesized separately, because they are `LineGlyph`s. A `LineGlyph` inherits from `Glyph`, so it has all the same properties, but it has two points - the starting and the ending point. These two points are also used to synthesize line glyphs, so the `LineGlyphSynthesizer` interface has a slightly different API. Overriding those is analogous to regular glyphs.

While line glyphs have these two special points, regular glyphs have only one - the affine space origin point. This point is used to place glyphs into the scene by the music notation synthesizer. Where exactly this origin point is located in relation to the glyph image depends on the glyph label (glyph type). Most glyphs have the origin as the geometric center (noteheads, rests), but some have it offset to some important location (the 4th line for the G-clef, the eye center of a flat, the touching line for a whole and half rests, etc.). See the documentation on Glyphs to learn more.

4.6 Conclusion

You've learned how to implement a custom glyph synthesizer and use it in a larger model. Overriding other synthesizers is analogous. Now you can read the rest of the documentation to learn in more detail about various parts of the Smashcima framework.