# The SysSon Platform

**Technical Report TR-2017-01-1**
**Institute of Electronic Music and Acoustics, Graz**
**(Status: in progress)**

Hanns Holger Rutz

January 2017

## 1 Adding an Offline Preprocessing Stage

The current workflow has been to experiment with matrix preprocessing directly in the IntelliJ IDE and outside of Mellite/SysSon. This is fine, because the IDE is much more powerful, and we have easier access to some of the utilities. However, the problem arises that we will need to make the results of these experiments available inside SysSon if we want to eventually allow the users to apply particular, newly developed preprocessing steps. We have done so with a separate menu-item to calculate anomalies, for example. We would now need to add another item for analysing the blobs. Obviously, this is not a scalable approach.

In the summer of 2016, I began to implement the next-generation FScape software, a toolkit for musical signal processing, using a UGen graph approach similar to ScalaCollider, but running offline. The back-end architecture uses the Akka-Stream framework, while the front-end API is very similar to ScalaCollider. This new FScape 2.x is already stable enough to use, as was demonstrated in various audio- and video-installations last year. Furthermore, it has a basic integration with SoundProcesses now. It is therefore an obvious choice to extend FScape with SysSon-specific UGens. A possible drawback is that the streams are weakly typed one dimensional signals (although channels can be bundled). There are a number of two-dimensional matrix and image processing UGens, but it requires that row size or width information must be explicitly passed into the respective UGens. Nevertheless, we estimate that it will be possible to implement the required UGens for SysSon matrices this way.

### 1.1 Implementation Steps

We have collected the required steps for a full implementation for processing matrices offline in SysSon below. As of February 2, the first five steps have been completed.

1. enhance SoundProcesses by "lazily" calculated objects
2. implement such a lazy object interface for FScape output
3. implement a caching mechanism for these objects
4. make it possible to use these cached instances as input to the real-time playback
5. implement a simple SysSon matrix reader in FScape
6. make `Sonification` instance matrices available in FScape
7. add commonly required meta data UGens, such as rank, size, dimensional information
8. add dimensional operators, such as transposition
9. add the possibility to *output* matrices from FScape
10. make it possible to use these cached matrices as input to the real-time sonification
11. add UGens necessary to complete existing processes (anomalies, blobs)

#### 1.1.1 Lazily Calculated Objects

*SoundProcesses* now makes a simple provision to add support for lazily calculated objects (as would occur in *FScape*). A new `Obj` type `Gen` was added that acts as an opaque container for an eventually calculated peer object.

The only way to get hold of that value is to create a `GenView` instance. This instance, once created and until it is disposed, acts as an acquired lock on the caching/cached resource. A `GenView` is created by passing a `Gen` instance and an implicit `GenContext`. This context ensures that rendering processes are shared within a workspace between multiple occurrences of the `Gen` objects.

The `GenView` is similar to a transactional future. It has a method `value` that returns a `Option[Try[Obj[S]]`—corresponding to a `future.value`—a state that indicates the current progress or completion, and it is observable for state updates.

### 1.1.2 Lazily Objects in FScape

Similar to aural views, gen views are created by factories which are globally registered for a specific peer data type. In `FScape` we created a new object type `FScape.Output` that is a sub-type of `Gen`, and a default gen-view factory for this type. The view type is `OutputGenView` and it combines an instance of `FScape.Rendering` with an `FScape.Output`. We ensure that at maximum one rendering instance is running for an `FScape` object. All outputs, i.e. all lazily created by rendering the same graph function, are thus grouped together.

### 1.1.3 Caching Objects

When requesting an instance of `FScape.Rendering`, the graph function is converted into a cache key. The graph expansion is now explicitly divided into two stages, the first giving the collection of UGens, the second creating the stream nodes in the Akka framework. In order to calculate the cache key, we only have to execute the first stage. The structure captures the positions of the constants and UGens in the graph, and collects for each UGen an "auxiliary" object that represents the structural data uniquely identifying the environmental inputs of the UGen. For example, this may contain the path name, length and modification date of an input file, or a constant such as an integer or a string obtained during the first graph expansion stage.

Each entry in the cache is associated with a value structure that contains the peer data for all `FScape.Output` instances. When graph elements provide a coupling to an `FScape.Output`, they call `requestOutput` on the graph builder, providing the a `Output.Reader` that can *de-serialize* the peer data stored in a cache value. Correspondingly, `requestOutput` returns an `OutputRef`, eventually passed to the stream graph element, that has a method `complete` to be invoked once that stream graph element has determined the peer data. The value passed to `complete` is an instance of `Output.Writer`, capable of adding serialised data to the cache value when it is written.

So, when requesting an instance of `FScape.Rendering`, either a valid cache entry is found (through the structure key), and the "rendering" is actually not a running rendering but an encapsulation of the already known result, or if not, an actual rendering will begin, and when it terminates, the cache entry is written.

To summarise, `FScape.Rendering` denotes an ongoing or finished (read from cache) rendering process, it encapsulates the peer data of *all* connected outputs of an `FScape` object. The outputs are opaque objects, by way of `GenView(output)` one obtains `OutputGenView` instances for these, which in turn extract their particular peer data by querying the common `Rendering` instance. By calling `genView.dispose()`, access to that data is nominally lost, and the system may decide to wipe the cache data if needed (for example, if a given maximum cache size is exceeded).

### 1.1.4 Using Cached Objects in Real-Time Playback

Cached objects appear as values in another object's attribute map, so for example, the result of calculating a number might be a `Gen` whose peer data is an integer or a double, and this `Gen` instance will now appear in an attribute map instead of an "eager" `IntObj` or `DoubleObj`. This requires that especially the implementation of `AuralProc` recognises these new objects.

While a `UGenGraphBuilder.Value` has a `async` flag that could be useful here, we can first focus on input that must be resolved at graph expansion time, including scalar attributes without default value (and thus without a default number of channels). This is a simpler approach: When encountering a `Gen` during a `requestInput` call, we get hold of a `GenView`. We maintain a new map inside the aural-proc from attribute keys to gen-view instances. So, in `requestInput`, when this map does not contain the attribute key, we create the `GenView` and store it there. We then call `value` to see if the peer value is already available. If so, we look at this value and proceed as normal. If not, it means we need to wait for the rendering, and at this time, we cannot determine required properties such as the number-of-channels of a scalar input. We can throw a `MissingIn` exception, an established mechanism for deferring the playback of a proc until all required attribute inputs are present. Additionally, we need to track the completion of any `GenView` we have thus created. Similar to the `attrAdded` method which is invoked when an attribute is added to the proc's attribute map, we check if the completed view corresponds to one of the missing keys in the incomplete UGen builder state. If so, we attempt the build again.

A bit of complexity is added by the fact that we now must support `Gen` in a number of scenarios, from scalar values to in-memory buffers to streamed buffers. We have implemented all of these cases, but a future overhaul should look at a more general and DRY solution.

### 1.1.5 Simple SysSon Matrix Reader UGen

TODO:

### 1.1.6 Next Steps

The remaining six steps will be covered in the next technical report.