Motivation
Enhance image quality in interactive environments by adding fast, convincing visual effects such as Ambient Occlusion or Depth of Field, calculated in screen space, provide means to connect filters and thus create a "sandbox" environment.

Related Work
Gegl [1]: A framework for graph-based image manipulation, not suited for interactive/realtime processing.
Pandore [2]: Based on precompiled binary filters, also providing a graph representation and even a graph editor (Ariane), but also not suited for realtime processing.
Osppu [3]: Supports graphs and CUDA, but is directly tied to OpenSceneGraph.

Our Approach
We present a flexible method to add screen space effects by describing filter graphs in XML files and processing them with filter plugins. These plugins can explicitly be implemented using compute frameworks such as NVIDIA CUDA, thus providing real-time performance and flexibility regarding implementation details. In addition to color and depth information, our framework also supports the use of per-pixel normals and position information.

Interface and Program Flow
Our framework provides a simple, template-based interface for setting up and getting data from its central data structure. The programmer who wants to attach our framework to his application does not need to care for and administrative details regarding graph execution. Everything is handled internally, automatically by the GraphRunner-class. The main steps executed by this class are the following:

1. Load the XML-based graph description
2. Generate a sequential traversal accounting for all dependencies, determined by directed edges
3. Load the according filter plugins dynamically
4. Visulize the graph and generate the GUI
5. Traverse the graph sequentially and run each node

Access to static and dynamic filter parameters is provided via specific interfaces: An application-wide container, where all dynamically created data is stored and a wrapper for the concrete graph representation.

Some Implemented Filters (using CUDA)

Depth Darkening [4]: Consisting of nodes for blurring the depth buffer (ordinary gauss filter), subtracting the original depth buffer from the blurred result and then using that data for darkening depth discontinuities.

Directional Occlusion [5]: Implemented for a single, screen space-defined point light. Consisting of steps to generate light vectors, calculate the visibility factors and multiply the input image with these factors. Depth of Field: Based on our own approach, implemented in a single node as well as a filter graph. The latter uses several fully gauss-filtered images and interpolates their pixel values in a depth-dependent way, yielding the appropriate values for the given depth (see f3, f4).

More: Gauss, fog, arithmetic operations, ...

(f1) Plain rendering without any effects
(f2) Directional Occlusion for Point Lights, Light is on the right, behind the observer
(f3) Depth of Field only, based on a brute force approach with separate fully filtered images and linear interpolation
(f4) Brute force Depth of Field filter graph
(f5) Depth Darkening, Depth of Field applied iteratively, followed by a slightly noticeable fog effect

(f6, f7, f8) XML Code snippet, graph visualization and automatically generated parameterizer GUI.

(f9) Benchmarks dependent on the amount of blur in a DoF filter; post processing times and whole frame calculation times, GeForce GTX 280 vs. GTX 480. Note the considerably high overhead between pure post processing time and frame calculation time, originating from the fact that the CUDA device responsible for the post processing was a separate graphics card in the system. The data was copied via the host to that device. Removing this constant overhead by directly mapping buffers is likely to improve the framerate a lot.

References: