GrIP: A Framework for Experiments with Screen Space Algorithms

Thorsten Roth, André Hinkenjann

Labor für Computergrafik
Hochschule Bonn-Rhein-Sieg
Grantham-Allee 20
53757 Sankt Augustin
Tel: +49 (0)2241 / 865 - 229
Fax: +49 (0)2241 / 8658 - 229
E-Mail: {thorsten.roth|andre.hinkenjann}@h-brs.de

Abstract: We present the extensible post processing framework GrIP, usable for experimenting with screen space-based graphics algorithms in arbitrary applications. The user can easily implement new ideas as well as add known operators as components to existing ones. Through a well-defined interface, operators are realized as plugins that are loaded at run-time. Operators can be combined by defining a post processing graph (PPG) using a specific XML-format where nodes are the operators and edges define their dependencies. User-modifiable parameters can be manipulated through an automatically generated GUI. In this paper we describe our approach, show some example effects and give performance numbers for some of them.

Keywords: Post Processing, Screen Space, Framework, Sandbox, CUDA

1 Introduction

Screen space post processing can be used as a tool for enhancing image quality in interactive environments by adding fast but still convincing visual effects. Examples of these effects are shadows, ambient occlusion and depth of field as well as simpler additions such as enhancing contrast or blending a logo. We present a flexible way of combining the single steps of aforementioned effects in PPGs: The framework GrIP (Graph-Based Image Processing). PPGs can be created by means of a specific XML description, consisting of post processing nodes (PPNs) connected by directed edges. Unlike other frameworks (e.g. GEGL [Osgb]), the presented implementation is well-suited for interactive applications as it supports the use of state-of-the-art parallel architectures such as NVIDIA CUDA. Also, it is not tied to a specific application, as osgPPU [Tev] is to OpenSceneGraph. In addition to color and depth information, we also support normal and position information (in world coordinates) for each pixel. More per-pixel information can be easily added. A visualization component for PPGs is generated, as well as a component for interactive parameterization of PPNs. The latter, based on FLTK [Bil], is automatically built from data provided by the loaded PPG and
supports the user in experimenting with newly implemented operations or finding parameter
settings which lead to satisfying visual results. Static parameters are simply displayed, while
dynamic parameters can be modified with a slider. Visualization is done using a node-link-
diagram, showing dependencies between nodes. Advanced effects are separable into several
simple (e.g. arithmetic) operations, which can in turn be used as components for other
PPGs. As a result of the ease of use of our approach, the framework can be used as a
"sandbox" for experiments in screen space.

2 Related Work

2.1 Post Processing Methods

One effect that can be implemented using screen space algorithms is antialiasing. Recently,
several approaches for screen space antialiasing have been proposed, e.g. by Reshetov (Mor-
phological Antialiasing, MLAA) [Res09] or Chajdas et al. (Subpixel Reconstruction An-
tialiasing, SRAA) [CML11]. The former has also been ported to the GPU and extended by
adding topological reconstruction algorithms [Bir11].

Another category of algorithms that are of great interest for screen space approximation
is illumination. While recently there have been huge advances in ray and path tracing, both
are still very demanding when it comes to computational resources. Approximating these
algorithms in screen space removes their dependence on geometric complexity, hence making
the rendered resolution the sole real influencing factor. A possible approximation of global
illumination is described by Ritschel et al. [RGS09], extending screen space ambient occlu-
sion (SSAO, e.g. [BS08]) by directional information and also diffuse indirect illumination.
Another approach creating a visual appearance similar to SSAO is presented by Luft et al.
in [LCD06]. Cf. section 4.2 for our implementation and a further description.

To achieve a realistic image impression it is desirable to simulate real lens-based camera-
systems by adding a depth of field effect instead of relying upon the basic pinhole camera.
Several methods have been proposed to achieve this effect, starting with distributed ray
tracing [CPC84]. In screen space, layered scenes [KTB09] can be used to create a con-
vincing effect and remove artifacts originating from simple depth-dependent blurring, while
performance improvements can be achieved by using summed area tables [HSC+05].

Other effects that can be achieved using screen space calculations include motion blur,
advanced scaling-algorithms, denoising methods as well as more "basic" image processing
algorithms such as convolution, brightness and contrast changes or gamma correction.

2.2 Related Frameworks

There are post processing frameworks similar to GrIP, but each of them differs in one or
another way. One of the most interesting systems in existence is GEGL [Osgb]. It is quite
similar to GrIP in terms of the underlying concept of graph processing, but while in GEGL
each node represents an image, in GrIP it is just an operation with arbitrary output, hence
yielding a higher degree of extensibility when it comes to adding and combining new node types. However, both share the concept of using a plugin system for PPNs. In general, GEGL is not meant to be used in real-time applications, but as an offline-tool, while GrIP is specifically designed for interactive applications. Therefore we also provide a very simple GUI for parameter changes to make experimenting easy.

Another relevant framework is osgPPU [Tev], which is specifically designed for working with OpenSceneGraph by linking post processing units (similar to PPNs in GrIP) directly to the scene graph. While osgPPU provides the means for working with interactive applications by supporting the use of shaders as well as CUDA for post processing, its dependence on OpenSceneGraph is a major disadvantage in comparison to GrIP, which can easily be combined with arbitrary applications. Other frameworks for image processing include GpuCV (GPU-based implementations of OpenCV algorithms) [AHAS] and Pandore [Clo10], where the latter is also important because of its advanced graph system, e.g. also supporting several kinds of conditional execution.

3 Framework and Interface

In this section, we will describe GrIP’s underlying concept, including its external XML-interface as well as the concrete implementation. PPGs, internal data/memory management and the graphical user interface are explained.

3.1 Post Processing Graphs

PPGs serve the purpose of connecting several different operations, making it possible to divide complex calculations into smaller, simpler steps. These can in turn be designed to be as generic as possible, allowing for reusability of frequently used operations. Conceptionally, PPGs are based on the mathematical definition of a graph consisting of nodes and directed edges, which in turn represent operations and dependencies, thus yielding information about the prerequisites of single node executions. Data flow is not directly determined through the graph’s edges, but instead by supplying the nodes separately with input and output parameters. Other (node specific) parameters such as kernel size for a gaussian blur are also supplied. At first glance, it seems as if node execution in the correct order should also be possible by just looking at the nodes’ input and output parameters. However, considering the possibility of using the same input data in multiple nodes and modifying it in-place renders this approach useless and makes the additional declaration of adjacencies a necessity. While PPGs are generally acyclic, loops are allowed via special delimiter nodes (loop start and end node). In this context, the loop start node has to be supplied with information about the succeeding first node inside the loop, so that the graph can be correctly parsed. A possible loop looks as follows, also giving an impression of how PPNs in general are parameterized (cf. Figure 1):

<node_loop_start name="gauss_loop_start">
As shown in the listing, each PPN has its own unique name which is used to refer to this particular node when constructing the edges. It can be easily seen that this general concept is applicable for arbitrary compositions of PPNs. Note that all specific node parameters in the listing have been supplied with the mutability attribute set to static. This can be altered to be dynamic, requiring the user to provide several additional attributes such as the data type (floating point or integer), minimum and maximum parameter values, step size between values and the initialization value. This information, together with a parameter name instead of the specific value used in the static case, is then used to generate a graphical user interface using FLTK, enabling the user to modify parameters at runtime and thus to comfortably experiment with the PPG. An example for a dynamic parameter in the XML representation looks as follows (kernel size for gaussian blur):

```xml
<size mutability="dynamic" datatype="dt_uint32_t" min="1" max="127" step="2" defaultvalue="3"> kernel_size </size>
```

This parameter will then be made accessible in the GUI using a slider labeled *kernel_size* and a domain of \{1, 3, 5, …, 127\}. An example of an automatically generated GUI is given in
Figure 2. If the same parameter name reoccurs in a PPG, only one slider will be generated. This allows for the variation of different PPNs’ input with just one interaction element. Of course, data types have to match for this to work properly. The framework accesses the graph structure via a wrapper class, so that in principle different representations are possible, as long as the wrapper interface is implemented correctly. This wrapper class provides methods for getting PPNs, their types and connecting edges as well as for accessing parameters, getting their mutability values and several other useful functions. This means that loading a PPG is independent of the specific XML-representation, but relies on a wrapper that serves as a "translation" layer.

3.2 Data Storage and Communication

All PPNs need an implementation, which is dependent on a specific interface. Also, data has to be communicated between nodes and the framework. To achieve this, a generic class, called UniversalContainer, is provided. This container is used to store all dynamic data incurring at runtime, as well as input and output parameters. Parameters with static mutability are accessed through the aforementioned wrapper class. To ensure that only supported types are stored in the UniversalContainer, template specialization is utilized, so that interaction with the container is limited to a predefined set of types. These predefined types are concrete and smart pointer types for all primitive C++ types plus additional user-defined types based upon them. The latter are only supported as smart pointers to reduce the programmer’s responsibilities regarding memory management. In addition to basic object information, the UniversalContainer also stores further information which is retrievable without template parameters, permitting type-related flexibility at runtime. This information includes a type identifier describing the data type, a dirty flag used for memory management and a device pointer for support of an additional address space (GPU memory in this case). The dirty flag can currently have one of three states, providing information on whether data has been modified in device memory, host memory or none of both since the last synchronization. By considering this information before copying data between host and device memory, time consuming copy operations can be avoided by keeping the data where it is actually used and not exchanging it without purpose.

3.3 Code Generation

Extending the list of supported types can be easily done by modifying an included AWK script, which is responsible for generating all the type dependent code in GrIP. Here, the basic type, the name of the resulting typedef and the smart pointer type’s name are user-definable. Since GrIP is completely header-based, adding or modifying types does not have any consequences like re-compilation or the like. The following code snippet provides support for a concrete instance of the template-based type UniversalImage, using the primitive type dt_float_t and four components:
dependent_types_base[0] = "UniversalImage<dt_float_t, 4>"
dependent_types_type[0] = "UniversalImage_float_4"type_suffix
dependent_types_sptr[0] = "UniversalImage_float_4"sptr_suffix
dependent_types_cuda[0] = "float*"

Note that the bottom line is used to provide direct support for NVIDIA CUDA by adding information on how the data type is represented when processed by this GPGPU framework. This information is especially necessary for being able to automatically free allocated device memory after the graph execution loop has been aborted (e.g. on exiting the application). This behaviour can be easily adapted to other computing frameworks such as OpenCL. Type specific code is used in several components of GrIP, including typedefs, integral type identifiers using RTTI, enumerations for type identifiers, user-retrievable type names, the UniversalContainer class and memory management regarding the GPU.

3.4 Implementation of Post Processing Nodes

Implementing a PPN is straightforward. Each node needs to implement the node interface, ensuring that the necessary data is passed. This means that a node always gets information on its instance name, the current GraphWrapper instance for accessing static parameters and names, and the currently used UniversalContainer instance for gaining access to dynamic (runtime) data. Additionally, utility macros are provided for automating procedures such as distinguishing between dynamic and static node parameters, as well as supporting the implementation of CUDA-based PPNs by automatically managing device memory. Most implementation details for PPNs are left to the user, as long as the data is handled correctly when it comes to storing it in the UniversalContainer and keeping information on device memory up to date. The user is aided in both tasks by aforementioned helper functionalities. PPNs are compiled into shared libraries and stored in a common place where the GrIP framework can find them.

3.5 PPG Execution

Execution of a PPG is handled using a very simple interface defined by the GraphRunner class, simply specifying which graph to load, whether parameters should be modifiable via the automatically generated GUI and switching the graph visualization on or off. Input parameters can then be set directly via a wrapper interface for the UniversalContainer, while output parameters can also be obtained through such a wrapper interface. This means that almost no implementation details are exposed to a programmer who uses GrIP in his application. The basic scheme of graph execution incorporates loading the XML-based graph description, generating a sequential traversal accounting for all dependencies, loading the according plugins, visualizing the graph and generating the GUI (if activated) and finally traversing the graph sequentially and calling each node's run method. Regarding graph traversal, note that GrIP is designed for putting parallelism into nodes, meaning that nodes
are never executed concurrently, but may (and should) be implemented with multicore architectures in mind. Also, this is why we chose CUDA for all node implementations presented in this paper. Of course, future improvements could also incorporate parallel graph execution.

4 Exemplary Applications

In this section, we present our test setup used to benchmark exemplary PPN implementations as well as visual results and a short description for some of these operations. For actual benchmarks, see section 5.

4.1 Test Setup

Due to the relatively easy implementation and its broad support for useful techniques, we chose OpenSceneGraph [OSGa] as the foundation for our example application. Note that this is in no way a dependency, but just an arbitrary choice for an exemplary implementation. A simple object viewer serves as a utility for demonstration of our implemented PPGs. It makes use of methods such as render to texture, multipass rendering and multiple render targets for providing the necessary data and using it in the application after it has been processed. Figure 3 shows the details of our setup and emphasizes the simple integration of GrIP into the scene graph by rendering all necessary input for GrIP in a PRE_RENDER pass, then processing it and putting it back into a texture, which is bound to a screen filling quad rendered in a POST_RENDER pass. Note that the presented test setup relies on copying the data between graphics and host memory, justifiable by thinking about the generic applicability offered by GrIP: It is not only meant for hardware rendering systems, as given in this example, but also for software rendering solutions such as ray tracers and the like. In addition, direct memory mapping between OpenGL and CUDA would also be possible with recent CUDA versions, but did not yield better results in our tests, because GrIP’s operations were executed on a separate CUDA-capable GPU anyway. Nevertheless, setting the device pointer to known information and adjusting the dirty flag accordingly enables using the device data directly without having to cope with a host indirection.

4.2 Depth Darkening

One of the perception-enhancing effects presented in [LCD06] is Depth Darkening. This is based on employing the depth buffer for darkening depth discontinuities, using less distant objects to darken adjacent objects which are further away. For achieving this effect, first a blurred depth buffer is created by calculating \( D_G = D \ast G \), denoting a gaussian convolution filter with \( G \) as the filter kernel and \( D \) as the depth buffer. Subsequently, we calculate \( \Delta D = D - D_G \) as the per-pixel difference and only take the negative values from this, so that \( \Delta D^-(x, y) = \Delta D(x, y) \) for \( \Delta D(x, y) < 0 \) and 0 otherwise. \( \Delta D^- \) can then be used for a per-pixel addition to the color-buffer, weighted with a coefficient \( \lambda \), to achieve darkened edges towards more distant objects. See section 5 for a sample image and benchmarks.
4.3 Shadows

Our implementation of screen space shadows is loosely based on [RGS09], where directional occlusion is presented as an improved alternative to ambient occlusion by also accounting for directed illumination when it comes to occlusion sampling. Contrary to their approach where an environment map is sampled for its contribution to scene illumination with respect to individual rays, we only sample the direction of one screen-space-defined point light at the moment. Similar to shadow rays in ray tracing, we then traverse the pixel grid of the image, comparing the linearly interpolated depth values on the ray with the pixels’ depth values. If a pixel is found to be in front of the ray, traversal is aborted and the original pixel is set to be shadowed. While this approach suffers from inaccurate shadows, originating from missing thickness information and thus cannot render e.g. the shadow of a column in a room correctly, it is still quite visually convincing in many scenarios. Additionally, the maximum sampling distance may be set by the user, who can also activate a linear, distance-dependent interpolation of shadow intensity as well as a blur effect for creating a soft shadow appearance. The whole shadow implementation is divided into three node types: The first node type calculates direction vectors pointing to the light source for each pixel, the second type checks for occlusion, creating an occlusion buffer as its output, which is then used by the third node type, representing a simple multiplication operation. A sample image is shown in section 5.

4.4 Depth of Field

Depth of field is an effect well-known from real lens-based systems. In contrast to a pinhole camera, depending on parameters like aperture and the focused object, the image appears blurred. Aside from scenarios where this appears as a disturbing artifact, it is often used as an artistic effect or for directing the viewer’s attention to certain parts of an image, e.g. in movies. We provide two implementations, both based on sampling individual pixels of the image with a depth-dependent gaussian, determined by considering user-defined values for a near and far plane defining the ”sharpness range” and comparing these with the pixel depth.
value. Outside of these areas, maximum blur is also individually definable and individual pixel blur is calculated by linearly interpolating between no blur and maximum blur for the near or far range. Our first approach filters an image completely with several kernel sizes and then interpolates them. Consider a pixel which needs a blur radius of $k$. This can be calculated by considering the nearest kernel sizes $k_i$ and $k_j$, $k_i < k < k_j$ and interpolating $k$, resulting in a $c_0$-continuous behaviour.

The interpolation component is implemented by a separate PPN, used once for each kernel size interval. Each of these nodes then gets the respective input images for its interval of desired kernel sizes and only creates an output for the appropriate pixels. Thus, all instances of this PPN share the same output buffer, as each pixel will be filled by exactly one of these nodes, creating a completely processed output image. For $n$ kernel sizes (including kernel size 1 for the original image), the PPG consists of $2(n - 1)$ nodes.

The obvious disadvantage is the number of superfluous filter operations: With $n$ kernel sizes used for pre-filtering the image and only 2 kernel sizes needed for the interpolation of a pixel, there are $n - 2$ unneeded filter operations per pixel, resulting in a huge computational overhead. As this implementation is only meant to be an example for what can be achieved with a PPG, this seems to be still acceptable. Our second approach is simply based on a single node, also using the interpolation method, but filtering pixels individually. Although our separated gaussian blur is usually not applicable in this case, as each pixel is potentially blurred with a different kernel size, visual results are still convincing. Artifacts that can occur in some scenarios are badly filtered corner points, especially at greater depth variations. Visual results and benchmarks are presented in section 5.

4.5 Other

Other operations we created for GrIP include arithmetic filters (addition, subtraction, multiplication, division, weighted average), fog and the aforementioned separated gaussian. Lately, GrIP has also been used for dynamic tone mapping, gamma correction and a bloom filter in our recently-developed path tracing software.

5 Results and Discussion

Visual results for depth darkening, shadows and depth of field can be seen in Figure 4, Figure 5 and Figure 6, while Figure 7 shows performance tests for screen space shadows, comparing a GeForce GTX 280 and a GeForce GTX 480. Here, $t_{calc}$ is the pure calculation time without any overhead, which means that copy operations as well as pure rendering time for providing the necessary data for GrIP processing are not included. Adding these times up results in $t_{frame}$, the complete frame rendering time.

It is obvious from the measurements that there is a huge overhead, mostly originating from copy operations. For a filter of multiple nodes, the hardware-rendered buffers (color, depth, normal and position buffer) are passed to the host and then copied to CUDA memory.
as needed. In the end, the result has to be copied back to graphics memory of the displaying card, so it can be used as a texture for the screen-filling quad. Note that using the same GPU for rendering and filtering could avoid these overheads, thus making the difference between \(t_{\text{frame}}\) and \(t_{\text{calc}}\) much smaller. Nevertheless, it is easy to see that the performance gain from using a GeForce GTX 480 is huge. When varying the maximum sampling distance for "shadow rays", the real performance achieved by the GTX 480 is better than the theoretically achievable performance for the GTX 280 from a sampling distance of 80 upwards.

Figure 8 shows a non-linear correlation between rendering time and far plane distance, delimiting the focus range. Bringing this plane closer to the observer results in a bigger number of pixels to be blurred which in turn increases calculation time. This is non-linear basically because it is dependent on the visible scene geometry. The figure also shows – aside from the already explained difference between \(t_{\text{frame}}\) and \(t_{\text{calc}}\) – that the pure calculation time for the single node including copying overhead from the host to GrIP \(t_{\text{total}}(\text{dof integrated})\) as well as the complete PPG execution time \(t_{\text{total}}\) are within 12 milliseconds of the pure calculation time. This means especially that the overhead for graph execution is relatively small, which has also been confirmed by further measurements with bigger graphs, yielding an overhead time of approximately 1 millisecond per node from graph execution. However,
graph execution is still relatively unoptimized and could be improved by e.g. removing string comparisons utilizing a preprocessing step.

6 Conclusion and Future Work

We have presented GrIP, a flexible framework for post processing of visual data in interactive applications. This flexibility is achieved by pursuing a modular approach, using a plugin system for PPNs, which are then loaded at runtime. Compatible nodes can be arbitrarily combined by means of an external XML-based format for graph definition, where each node has to be supplied with information about its input and output as well as other, more specific node parameters. Accessing the external graph definition is held independent of XML by passing requests through a wrapper class. Dependencies are determined by defining directed edges between nodes. From this, a sequential graph traversal can be generated, accounting for all dependencies and thus yielding the desired results. Parallelism is put completely into node implementations, which are also supported by auxiliary methods, e.g. for easier implementation of GPU-based operations.

Future work could include a closer connection between visualization and parameterization, meaning that graph visualization could be used to select the nodes to be parameterized and only show this node’s details. Also, graph visualization could additionally account for input data, making data flow more obvious, as the current visualization only shows defined node dependencies. Visual programming of PPGs would be another huge improvement for GrIP, enabling the inexperienced user to directly create graphs instead of first incorporating into the XML structure. Asynchronous processing is a possibility of enhancing performance, so that frame rendering for the next frame can already be done while post processing is performed for the previous one. Though, this depends largely on the calling application, rather than the callee (GrIP). A general region of interest (ROI) system would be an interesting extension. This could e.g. be realized by implementing a masking system, also incorporating floating point arithmetic, enabling the user to create smooth transitions between different image areas. Finally, GrIP could also be used for processing data other than that from computer graphics applications. In principle, applicability is given for arbitrary applications with huge calculation steps which are separable into simple, reusable steps. As we have shown, additional data types can be easily added to GrIP by using the included code generation script.

References


