OPENNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering

—Extended but Unpublished Version of the Pacific Graphics 2003 Short Paper—

Nick Halper Tobias Isenberg Felix Ritter
Bert Freudenberg Oscar Meruvia Stefan Schlechtweg Thomas Strothotte

Department of Simulation and Graphics
Otto-von-Guericke University of Magdeburg
{nick|isenberg|fritter|bert|oscar|stefans|tstr}@isg.cs.uni-magdeburg.de

Abstract

The notable amount and variation of current techniques in non-photorealistic rendering (NPR) indicates a level of maturity whereby the categorization of algorithms has become possible. We present a conceptual model for NPR, on which we base a modular system, OPENNPAR*, which integrates NPR algorithms into distinct classes in which their components are modularized and consequently re-integrated for various rendering purposes.This allows* OPENNPAR *to reproduce many kinds of NPR algorithms, including the integration of 2D and 3D methods. Additionally, the system provides support for a range of users (developers, programmers, designers) according to their respective levels of abstraction, thus being available in multiple contexts. Ultimately,* OPENNPAR *holds great potential as a tool in the development, augmentation, and creation of NPR effects.*

1. Introduction

The nature of non-photorealistic rendering (NPR) in its simplest definition, is a form of visual communication. As communication is virtually endless in its possibilities, NPR attempts to succinctly define options within this scope. Particular rendering styles are capable of conveying context-specific information in an application-dependent environment. Thus, there is a need for an effective rendering system which provides support for multivariate applications.

Despite the plethora of non-photorealistic effects available, there exists a rather limited number of primitives actually employed to generate these effects. There also remains a similarly limited number of general techniques for the application of these primitives. The ingenuity of the algorithms underlying the aforementioned effects lies not in the mere application of these primitives, but rather, in their *combination*. Thus, a system could be designed wherein all modular components are freely combined and interchanged. Moreover, photorealistic rendering is a subset of non-photorealistic rendering—the term NPR is often misleading in this context. Therefore, this system could also include photorealistic capabilities.

To achieve the necessary modularity for the proposed system, NPR techniques must first be categorized according to their various properties. Specific classes of algorithms can then operate on the same sets of data—consequently sidestepping unnecessary data conversions between software projects. In addition, NPR algorithms can be individually broken down into a set of smaller algorithms, wherein an 'elementary set' of algorithms are eventually defined. Logically, keeping modules small and simple increases the range and flexibility when generating more complex algorithms. Finally, functionality is little without application. An effective means of presenting available options in the system to a variety of users will allow content creation at a level that satisfies individual requirements. Involved herein are those who actually create the modules, those who plug the different modules together to create a specific effect, and finally those who use effects to produce images.

The paper is structured based on these user classes and we describe the tasks involved at each level of abstraction. Our main contribution is an attempt to unify many NPR techniques within a single framework. We present the initial system, OPENNPAR which embodies this framework in Section 2. Developing extensions to this system is described in Section 4, programming using the system in Section 5, and using the system to design new effects in Section 6. We base our conclusion in Section 7 on our achievements.

2. OPENNPAR

In this section we outline our design goals used to structure the basic architecture for our NPR system, OPENNPAR. We first categorize our base classes for algorithms

and groups of users on which we base a conceptual framework and introduce initial components for the core system.

2.1. Classes of Algorithms and Users

The field of NPR contains a large number of different rendering algorithms that come from many areas. This diversity makes it difficult to come up with a single system that encompasses all NPR styles and techniques. However, there are similarities between algorithms that make a classification possible. For instance, DURAND proposes a classification into four parts [3]: a *spatial system*, a *primitive system*, an *attribute system*, and a *mark system*. Although useful for terminology and discussion, there is still no clear direction for a unification of algorithms into a single system.

Our basic philosophy is to allow users of the system to define their own approach for producing the final output by making available a powerful, flexible, and extensible set of tools. We aim to provide small well-identified modules for better inter-operability, centered around a small set of basic primitives. Thus, we focus on direct relationships between primitives: primitives serve as states in our system described by attributes, and processes define operations on or between those primitives. In Figure 1 we propose a classification of three main categories of primitives on which algorithms can operate. Note that the processes (arrows) in the conceptual diagram can fall into any of the four categories proposed by DURAND above.

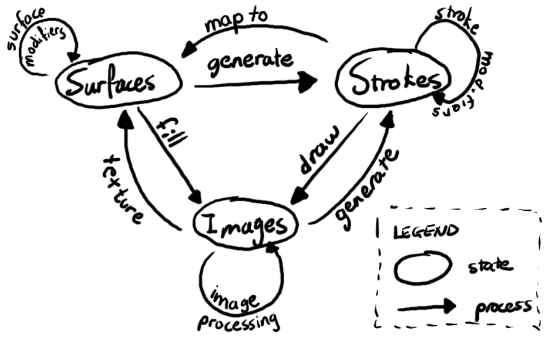


Figure 1: Our conceptual model for algorithms in NPR.

Visualizing the classification of primitives according to Figure 1 enables us to create building blocks (individual processes) that can be used in various combinations (a sequence of processes) for different rendering pipelines (the collective sequence of processes). In fact, many hybrid algorithms use combinations of two or all three of these classes (for an overview see [16, 3]). Table 1 outlines processes between primitives according to our conceptual model for several 'classic' NPR algorithms. Having such basic building blocks, the user can freely combine NPR algorithms in order to achieve a certain kind of image. However, the

Name	Description
SAITO & TAKAHASHI, Comprehensible Rendering [13]	Surfaces fill Images with shaded, z, normal data; Images process silhouette Image; Image processing (composite shaded and silhouette)
SALISBURY et al., Pen-and-Ink [14]	Surfaces fill reference Image; reference Image generates Strokes; Strokes draw into Image
MEIER, Painterly Rendering [12]	Images generate Strokes (particles); Strokes map to and Images texture Surfaces
KOWALSKI et al., Graftals [10]	Surfaces fill ID and desire Images; ID Image processing; desire Images generate Strokes; Strokes draw into Images

Table 1: A selection of classic NPR Algorithms and how they relate to the model in Figure 1.

amount of knowledge involved in the "programming" process for a rendering pipeline is different depending on the level of abstraction. Hence, a distinction into different types of users is needed. These types are—with increasing level of abstraction—developer, programmer, and designer.

At each level of abstraction, certain knowledge is needed; the higher the level, however, the less significant are technical and technological details so that the designer can concentrate on the image generation process. Thus, the goal of OPENNPAR is to embody our conceptual model for algorithms in NPR shown in Figure 1 and support user groups of various knowledge levels for creating NPR images.

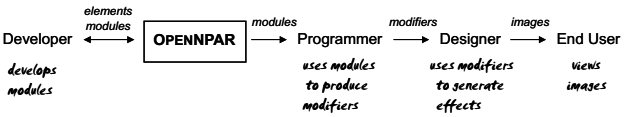


Figure 2: OPENNPAR knowledge pipeline

In Figure 2 we show the knowledge requirements and dataflow between the various user groups and OPENNPAR. Designers are given the task of creating new images by applying a selection of effects onto existing images. Thus, to work effectively designers only need knowledge of what *results* are produced once processes reach the final image. The selection of effects offered to designers are assembled by programmers. Programmers understand the relationships between processes and the various categories of primitives. From this knowledge they can assemble modules into a rendering pipeline to produce an effect. The more modules that are provided to programmers, the more operations and thus effects they can offer to designers. The developer's

task, therefore, is to broaden OPENNPAR's base functionality by extending classes of primitives (by creating elements) and adding modular operations that decide what to do with them. Before we go into details for the user categories, we first describe some initial components of OPENNPAR.

3. Basic Architecture

OPENNPAR is built on the foundation provided by OPEN INVENTOR architecture, an object-oriented graphics architecture, that allows us to use a scene-graph based approach [15] to support our conceptual framework.

The OPEN INVENTOR scene-graph consists of three classes of *nodes*: *shape nodes*, which represent 3D geometric objects, *property nodes*, which represent appearance or other qualitative characteristics of the scene, and *group nodes*, which are containers that collect nodes into graphs. The communication between nodes is handled through *fields*. In addition, nodes can access external data in stored in *elements*. These nodes are traversed by *actions* which trigger specific behaviors in each node. A typical action is the render action which causes shape nodes to render their components to the frame buffer.

With respect to our conceptual model, primitives are composed from elements, whereas processes are performed in the render action procedure in nodes during a rendering traversal. In addition, OPENNPAR restricts each node to perform one specific process only—we distinguish these nodes from normal OPEN INVENTOR operation by calling them *modules*. Thus, we execute processes in Figure 1 though the use of *modules*, and represent the classes of primitives with *elements*. *Fields* are also used to aid the propagation of data between modules.

3.1. An Initial Set of Elements

In keeping with our design goals of inter-operability, the number of elements should be minimal but offer generic functionality. We implemented a few initial extensions and additions to elements in OPEN INVENTOR to provide data structures that support our various primitives:

Winged Edge Data Structure: provides surface connectivity information for surface primitives and covers a large and efficient generality of use [1].

Images: Many NPR algorithms require storage of additional data in image buffers. Thus OPEN INVENTOR image class is extended to include floating point data—effectively storing normal, depth, and *G*-buffer data.

Stroke elements: coordinate, material, and normal elements in OPEN INVENTOR can be used to describe points along a line or curve. However, we implemented additional elements required by strokes, such as thickness and orientation values.

3.2. An Initial Set of Modules

Modules, in contrast to elements, can be numerous but succintly defined. This allows groups of modules to operate on similar sets of elements, thus facilitating their combination. Adding new elements to extend primitives requires the use of modules to either: (1) generate these elements from existing data, or (2) define data directly for pushing into the element state. Thus, we include the following modules useful for a large number of non-photorealistic algorithms:

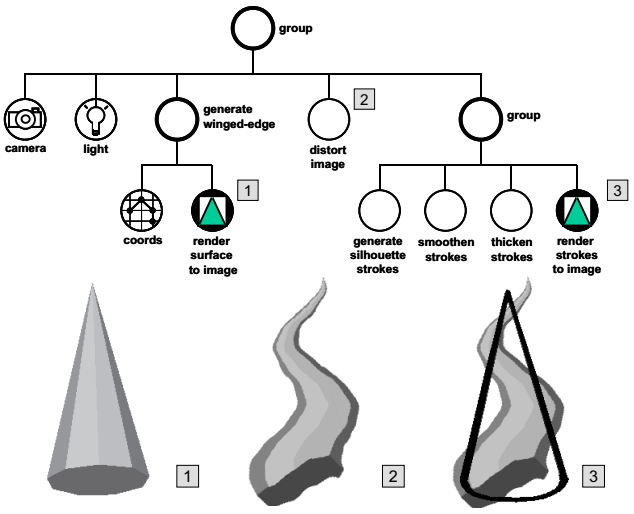


Figure 3: OPENNPAR scene graph example. The images represent contents of the frame-buffer at stages in the rendering traversal.

Render modules: We include specialized OPENGL render modules for surfaces, strokes, and images, each output defined by primitive elements describing attributes. Additional surface modules fill images with normals, UV, and object ID data. Figure 3 shows render module output at various stages during the rendering traversal.

Surface modules: OPEN INVENTOR includes many nodes that can be treated as surface modules. We add a module to generate generic descriptions of surfaces in the winged-edge elements. The most widely used NPR algorithms that require such information are silhouette algorithms, thus we include a silhouette module that reads winged-edge elements to generate stroke elements. For instance, in Figure 3 the 'generate winged-edge' module generates winged-edge elements that describe the surface of all its children, subsequently accessed by the silhouette generation module that sets stroke element data.

Image modules: A variety of image processing modules are added which propagate data through image fields. Unconnected field I/O utilize the frame-buffer image.

Thus, in Figure 3 the 'distort image' module reads in the current contents of the frame-buffer, processes it, and outputs contents back to the frame-buffer.

Stroke modules: To offer a variety of styles for stroke stylization, we introduce modules that modify the stroke primitive elements (e.g. thicken, smoothen, orientate, and perturbation). The last group node in Figure 3 contains two stroke stylization modules after the silhouette generation module: a smoothen and thicken module that read and modify the current stroke elements. Notice how the stroke render module renders undistorted strokes on top of the image since strokes were generated from surface elements.

4. Developing OPENNPAR

The developer's job is to add functionality to OPENNPAR by extending or creating new elements and modules. A primary challenge for the developer is to support interoperability between modules and encourage their re-use. This can be done by adding a great number of modules, but constraining the functionality of each to compute a single, specific task. In contrast, elements in the system should be kept generic to maintain a small set that covers a broad range of application. In this manner, a variety of modules can operate on the same set of elements which aids the interchange of data and resulting flow of computation. We now demonstrate these principles by showing examples of effective contributions to OPENNPAR.

4.1. Modularizing Stroke Operations

Our first example demonstrates how breaking down algorithms into elementary tasks adds flexibility and maintains modularity in OPENNPAR. Each elementary algorithm is encapsulated inside a module that can then be used independently or in combination with other modules.

We wanted to add functionality to OPENNPAR that would enable us to use 3D stylized strokes inside a rendered environment. To do this, there were a number of problems to overcome. First, we required that stylizations could be applied across long, smooth strokes. Second, we experienced stylization artifacts when strokes were projected to the viewport from 3D. Our final problem was that stylization modules could potentially alter positions of strokes that would then interfere with the scene.

We solved each of these problems with singular modules: (1) a module that connects strokes sharing common vertices; (2) a module that filters stylizations artifacts in projected strokes; (3) a module that performs fast hidden line removal (so that strokes could be rendered 'on top' of scenes). Details of these algorithms can be found in [8].

By keeping basic functionality in separate modules, we can apply them in a flexible variety of combinations in addition to the problem for which they were implemented. Furthermore, they are simple to use since the results of each module are tightly defined. One or more of these modules are used in each of our programming examples in Section 5.

4.2. Modifying Elements for Surface Shaders

Often developers can add modules to manipulate elements in unconventional ways. As a result, existing modules designed to use these elements will produce output differing from their initial intentions.

In our implementation, surface shaders generally modify elements so that subsequent rendering modules produce a different result. From a developer's perspective, a surface shader module simply sets values in elements used by rendering modules. For instance, we can add a module to modify texture coordinate elements based on lighting conditions. When inserted along with a texture module containing a two-tone image before the actual surface rendering module, we achieve the cartoon effect in [11].

We also take advantage of modularity present in modern graphics hardware programmability. To implement hardware features, we add elements that indicate current hardware options to use, and modules that load vertex programs and texture combiners to the graphics card. We leave the programmers to actually define what the hardware should do by allowing them to place code into a text field that is then compiled by the hardware modules. In this case, it is the hardware configuration that influences the subsequent output of rendering modules.

4.3. Re-using Elements for Skeletonization

As new algorithms are introduced, developers will often be required to support these using additional data structures. However, rather than adding new data structures for every algorithm we come across, we can map algorithms onto existing ones. This adds potential for alternative uses of elements and re-use of modules.

We demonstrate this by adding support for skeletonization in OPENNPAR, which is used in many NPR techniques (e. g., Deussen et al. [2]). The skeletonization process collapses edges in a surface definition. To do this, we make sure to load surface definitions into the winged-edge element using an existing winged-edge module. Now, we can add a skeleton module that computes using the winged-edge element. Rather than adding a new element to store the skeleton data, we leave the skeleton as represented by the winged-edge. These 'skeleton' elements can now be accessed by any subsequent modules in the scene graph. In addition, a separate module was included to generate a set of

strokes from the winged-edge data. Now we can also view and manipulate the skeleton using available stroke modules. The stroke generation module can also be re-used on 'regular' winged-edge surface definitions. This would now produce strokes from the wire-frame of a surface with equal opportunities for manipulation and stylization.

4.4. Extending Modules to Encapsulate Interaction

Developers can extend existing modules in OPENNPAR by using object-oriented strategies of deriving functionality from parent classes. We can even extend functionality beyond the representation and rendering of primitives. For instance, we added a novel form of interaction for NPR by extending a surface rendering module to evaluate whether or not object shadows on a plane are touched by the mouse pointer. To pass information about the shadow plane down the rendering pipeline, we added an element encapsulating the coefficients of the plane equation. Hence, the extended render module can test for an intersection of the ray from the mouse pointer to its shape in the shadow plane. An application that uses this interaction method is discussed in Section 5.2.

4.5. Polymorphic Rendering

Developers can present a variety of modules used for the same intent, but which produce results tailored towards a specific application requirement. For example, different rendering requirements can be supported by OPENNPAR by increasing our choice of rendering modules that act on the same element data.

In this case, we can implement a stroke rendering module that computes a particle simulation of paint along the course of a stroke and another that employs a hairy brush algorithm. These can take the same stroke elements as suggestions for using varying brush widths and sizes, speed, pressure, and so on. Furthermore, output does not need to be constrained to the visible frame-buffer. A module that takes strokes and generates a POSTSCRIPT file is possible by translating the stroke coordinates and thickness elements into the required format. As another alternative form of output, we have implemented a module with an input image field that consequently produces a video file. This allows renderings to be captured 'live' or sequenced into a steady animation.

5. Programming with OPENNPAR

The programmer accesses the functionality of OPENNPAR with the understanding of how modules can be placed into a rendering pipeline to produce desired results. Currently, a programmer can construct content by editing a scene-graph description in a text file and viewing the scene, or by calling OPENNPAR's API directly within an application.

In some cases, there are dependencies between modules and care must be taken to ensure they are placed in an appropriate order with their field connections properly setup. However, since modules in OPENNPAR are sufficiently succinct, relationships between them can be easily identified. Therefore, the real task given to the programmer is to exploit OPENNPAR's range of effects and, at times, define new algorithms by coming up with novel ways of ordering modules and interchanging data. In the following subsections, we look at how a few examples of these are constructed through different means of using the OPENNPAR API.

5.1. 3D Painter

Our first example is simply an exercise in using the OPENNPAR API directly within an application. This program allows a user to import any 3D scene and directly 'paint' on its surface. We use OPENNPAR to render the painted strokes as well as accessing many of its standard operations such as picking, interaction, and shading methods (see Figure 4). The interaction technique used is much like the algorithm shown in [9] where strokes are formed in 3D by 'drawing' on the surface of an object. The difference here is that each point picked on a surface maps its surface coordinates, normal, and color (from material or texture) to the stroke point. Additionally, the application constructed a utility that allowed for an undo-history of stroke operations by directly accessing the field data of the relevant coordinate, normal, and material modules for stroke elements.

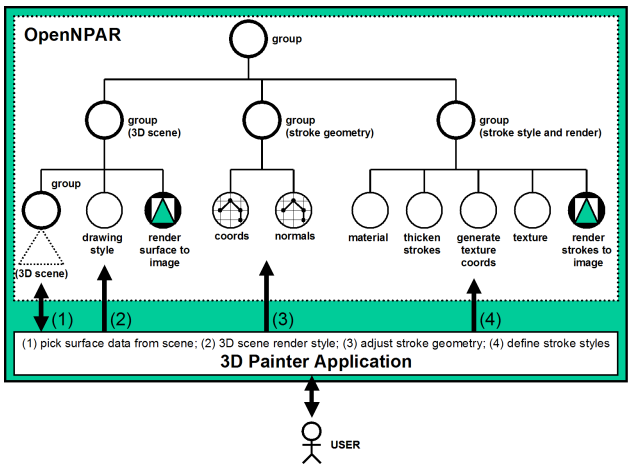
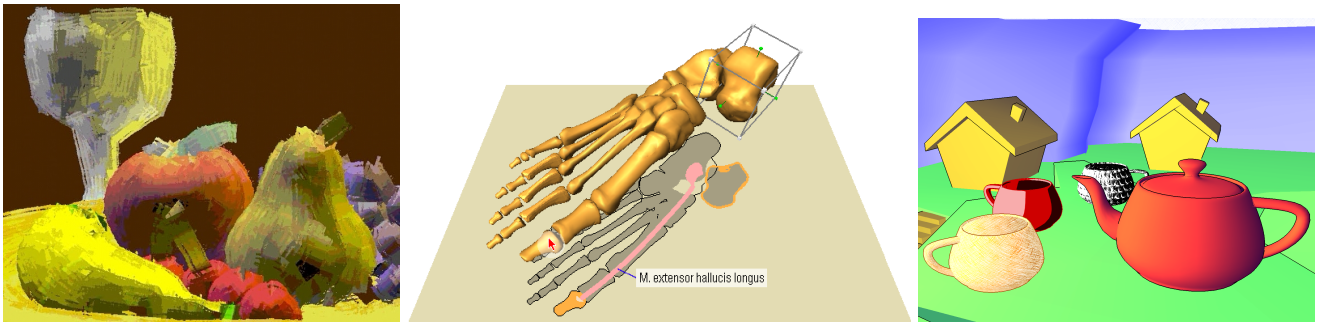


Figure 4: OPENNPAR Scene-graph for 3D Painter.

As a consequence of using OPENNPAR, the entire C++ code for this application (including comments) is less than



(a) *3D Painter*: Textured strokes 'paint' over models in this still life scene

(b) *Illustrative Shadows*: Shadows convey the current interaction context

(c) *Real-time NPR*: Modular surface shaders integrated into a game

Figure 5: Applications that use OPENNPAR.

1000 lines. Despite this remarkably small program, it allowed an animation of the painted scene in Figure 5(a) to be created by a user in a very short frame of time when given the model.

5.2. Interactive Illustration

Here, our computer generated illustrations make extensive use of non-photorealistic abstraction techniques to reduce the complexity of depicted structures. As the user interactively explores relationships in the scene, relevant details are emphasized whereas less important aspects are deemphasized or omitted to guide the focus of the viewer.

Figure 5(b) depicts the application of OPENNPAR modules to illustrate the current interaction context. Additional information about correlations between structures of a 3D model are displayed in shadows to enhance a users contextual understanding. In addition to photorealistic modules, the programmer made use of OPENNPAR's modules that cast individual shadows on a plane to enable a special kind of interaction (see Section 4.4). We also see two alternative uses of modules: first, a silhouette module placed so that stokes are aligned around the outline of shadows, rather than the actual structures, and second, modules to derive skeletons were used to guide placement of annotation anchors. In addition to color, line styles applied to the strokes of the outlines emphasize the relevance of important structures.

5.3. NPR in Games

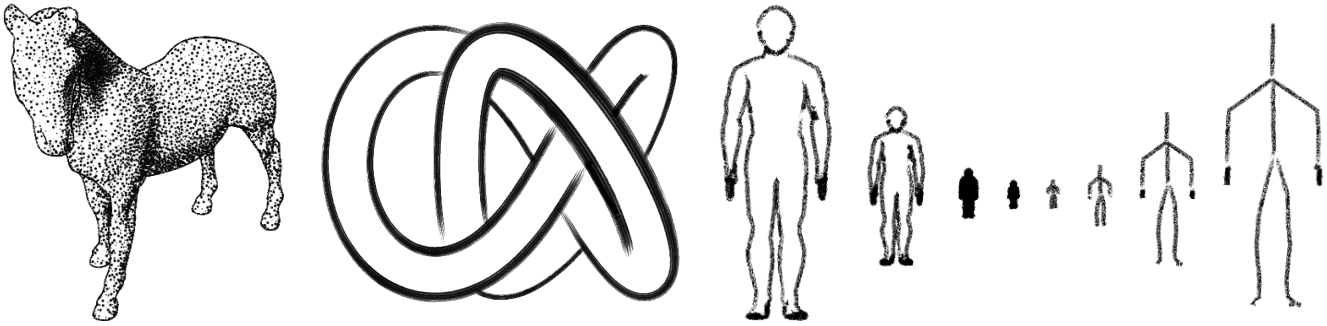
Figure 5(c) shows a sample scene from a game prototyping tool in OPENNPAR. Since games demand real-time, the programmer makes optimal use of the programmability features of hardware surface shader modules (a vertex programming and a texture combiner module, see Section 4.2) that is combined with other surface shading modules. We see examples of cool-to-warm shading, cel shading, stroke textures, and colorized hatching.

The cool-to-warm shading [5] is achieved by defining the vertex programming module to honor material colors of surfaces. As a result, this can be combined with a material module and is used for the teapot and environment. Colored hatching also reflects an object's color. However, a texture combiner module is added so that the interpolated diffuse color is blended with white based on a gray-scale image defined by a texture module. The cel shading is achieved by instructing the vertex programming module to generate texture coordinates for a one dimensional gray-scale texture based on the angle between a surface normal and light vector. This is combined with a texture combiner module programmed to use this texture to darken or lighten the object's base color (cf. Figure 10 (bottom)). The black-and-white stroke textures are rendered as described in [4].

5.4. Using an External Application for Stippling

Programmers of external applications can still use OPENNPAR's features. We take the case of frame-coherent stippling, designed and implemented outside of OPENNPAR. This is a stippling technique where most of the computation occurs as a pre-processing stage that structures a point hierarchy before the points are selectively rendered at run-time.

The point generation is done off-line independently from OPENNPAR. The output of this stage, however, can be written to a file which complies with one of OPENNPAR's file formats that defines the stipples as 'point' strokes. While the stipple generation stage still lies on the side of the producer, the rendering part now relies on OPENNPAR. Thus, potential for additional rendering features in OPENNPAR, such as its silhouette generation modules, can be used to generate the result shown in Figure 6(a).



(a) Using OPENNPAR to render stipples and silhouettes.

(b) Visible silhouette rendered in real-time with oil paint texture and depth cueing.

(c) Comparison of rendering silhouette drawings or skeleton drawings for different sizes of the object.

Figure 6: Sample algorithms implemented using OPENNPAR.

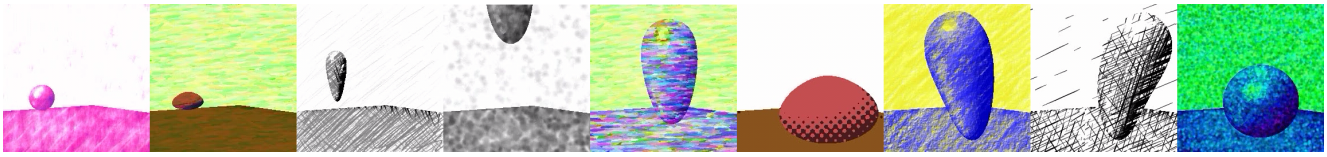


Figure 7: An animation played back with nine different image filters for artistic effect.

5.5. Editing an Animation Description to Export Filtered Video

An external 3D application can output an animation to a file that is then marked up by the programmer to produce a video including a few special effects from OPENNPAR. Note, that here *no compilation* is necessary to produce the results shown in Figure 7. A simple text editor and knowledge of OPENNPAR's API format is required.

OPENNPAR is able to read in VRML files that include additional animation nodes (currently we support vertex interpolation and group translation/rotation). The programmer edited the VRML scene description to add an image module at a specific location to read in contents of the framebuffer once the initial rendering of an animation frame was complete. A number of image processing modules were then appended to the scene graph. The effects for the image processing modules are controlled by their input image fields—in this case by typing in external image file names to use as filters—and connecting their outputs and inputs to propagate results. The last image processor module's output was connected into the image field of a video module, which was parameterized to insert images at specific time intervals to a video file. Producing this file was now simply a matter of using an available viewer application to read in the scene which would automatically run the animation.

5.6. Creating Level-of-Detail Silhouettes

Clever use of ordering modules and interchanging often allows programmers to define new algorithms. For this example, we construct a silhouette algorithm to generate stylized silhouette strokes at interactive rates that also uses a level-of-detail technique. As in the previous section, this can be achieved entirely through the use of a text file without need for any compilation.

We first insert the silhouette generation module that fills the stroke pipeline—the stroke elements—with silhouette edges. The stroke rendering module, at this moment, would simply produce thin lines for the silhouettes since no stylistic elements of the strokes have been used. To this effect, three stroke modules to support the 3D stylization of strokes (that we introduced in Section 4.1) were added directly after the silhouette module. Now, we can add stylization modules to affect the visual appearance of strokes and render these by disabling the z-buffer field in the surface renderer so that clean strokes are drawn 'on top' of the scene. The stylization achieved in Figure 6(b) combined stroke modules to influence stroke thickness elements, texture modules to read RGBA images from an external file, and a texture coordinate generator module to define the mapping of the texture onto each stroke.

For reduced level-of-detail (LOD), we could simply replace the silhouette generation module with a skeletoniza-

tion module and load the stroke pipeline with the skeleton data (see Section 4.3). This would similarly concatenate the skeleton edges to strokes and apply stylization to it. We notice that the skeleton is effective in conveying a good idea of the shape of an object when the object is far away (see comparison in Figure 6(c)). In addition, it proves computationally efficient due to viewpoint independence and typically generating less lines to convey the whole object than the full silhouette. To combine these two results, we added a LOD module that would select between the silhouette generation and skeletonization modules depending on the projected size of objects. Thus, when this scene is now loaded by a viewer application, the stylized silhouette of an object is replaced with its skeleton as it recedes into the distance.

6. Designing with OPENNPAR

Designers are given the task of creating visual results that carry a desired communicative intent. Whereas the programmer has the technical expertise to experiment with the system at a modular level, designer productivity increases when part of an entirely visual and iterative creative process. However, OPENNPAR at the modular level still imposes knowledge requirements about the inter-operability of modules—potentially hindering the creative process.

Consequently, we devised an interface to overcome impositions on designers by mimicking the designer's creative process in coming up with new images [7]. The contributions to the interactive design of effects are (1) a *method of interaction* which leaves the user unaware of the dimensionality of the data being used to create a given effect, and (2) a *method of computation* which assembles a unique pipeline of graphical operations to achieve the desired effect.

The tools created by programmers for designers are called *modifiers*. A modifier *manipulates*, *adds*, or *removes* modules in a scene graph, thus providing simple means for designers to interface OPENNPAR, whereby the modular components are abstracted by programmers into effects that produce *results*. In addition, each modifier is limited to use only its class of modules (e. g., a modifier cannot affect both a surface-shading module and an image module in the scene graph). Consequently, each modifier is a self-contained elementary block that fits into more complex, powerful structures according to our conceptual diagram.

6.1. A Method of Interaction

Modifiers abstract technical knowledge about module use in OPENNPAR. Thus, the designer is left unaware of data dimensionality and type-conversion requirements for effects composition. To take advantage of this, we implemented a simple interface to OPENNPAR wherein the designer simply 'sketches out' from existing images in a workspace and

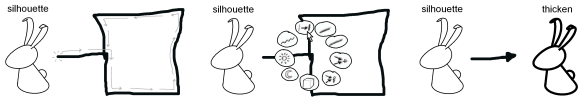


Figure 8: (left) Sketching out a new modifier. (center) Selecting a modifier from a pie-menu. (right) Result of applying a modifier.

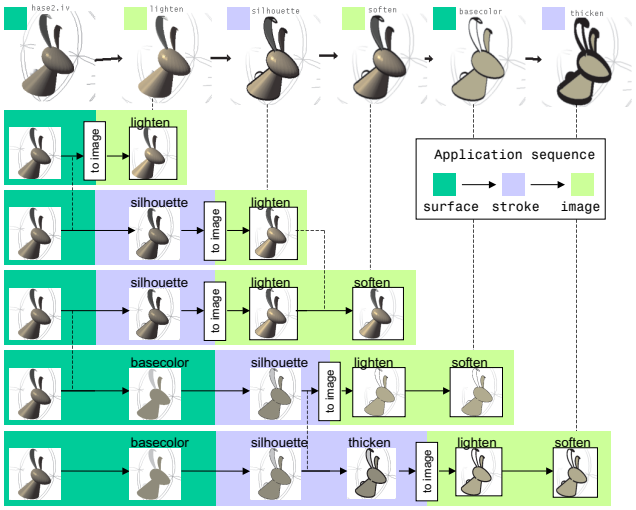


Figure 9: Designer and underlying system view of piping modifiers. Each of the displayed results at the top store a graph of modifiers (in this example a linear sequence) applied to the original scenes. Computational short-cuts are indicated by the dotted lines between successive modifier graphs.

selects a modifier effect from a pie-menu to produce a new effect (see Figure 8).

Figure 9 (top) shows an applied sequence of modifiers in which each visual result is consistent with designer expectations. Notice that from an algorithmic point of view, certain computations in this order are actually infeasible. For instance, the computation of the base color surface modifier cannot be done given a 2D image result as input (from the lighten modifier). In the next section, we show how this is made possible.

6.2. A Method of Computation

Each applied modifier is computed by first structuring a *graph of modifiers* from its inputs. Then, the applied modifier is inserted into the appropriate location within this graph of modifiers. Its field connections into other modifiers are then organized (or re-organized), and any necessary components for conversions between field connections are inserted. Thus, each modifier maintains an internalized ordering of the application of previous modifiers as its inputs.

In Figure 9 we see the actual translation of the designer's visual dataflow to the underlying system's dataflow. The

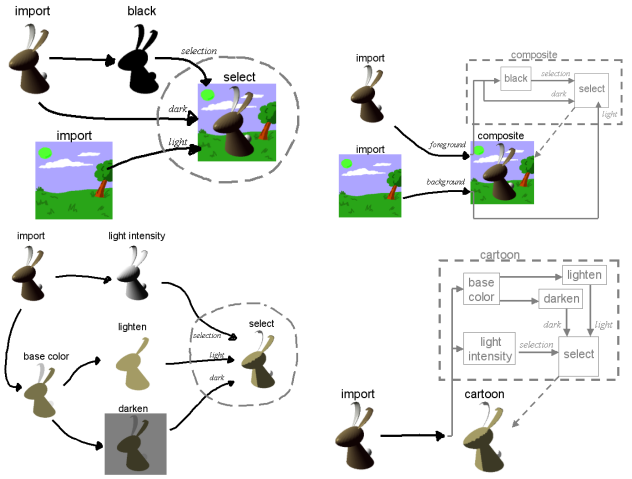


Figure 10: Circling modifiers (left) and subsequent dataflow encapsulation and remapping of inputs (right).

end result is a dataflow of modifiers that start with import modifiers. These introduce a scene graph that is first filtered through surface-shading modifiers and then stroke modifiers before converting 3D data into 2D images for input into subsequent image modifiers.

6.3. Re-use of Graphs of Modifiers

Since modifiers are self-contained effects that encapsulate a history of operations as a graph of modifiers, they can easily export and import combinations of modifiers, called *compound* modifiers, for re-use. We demonstrate this with a few examples combining both 2D and 3D effects.

In Figure 10 (top) we construct a composite effect from an image modifier 'select' and a surface modifier 'black'. Circling the select modifier collapses the dataflow and remaps input to the newly generated 'composite' modifier, mapping *foreground* and *background* inputs into the encapsulated field connections. Similarly, in Figure 10 (below), surface and image modifiers are combined to create a cartoon effect. Collapsing the select modifier in this case maps its single input to its respective encapsulated dataflow in the new 'cartoon' modifier. Finally, Figure 11 shows the combination of both the composite and cartoon modifier that integrates all the 2D and 3D effects.

To compute the saving and renaming of modifiers into new modifiers we count the number of unique import modifiers that enter the modifiers' dataflow encapsulation. For instance, the 'composite' example has two unique import modifiers that form part of its encapsulated dataflow, therefore the newly created 'composite' modifier has two inputs that can be renamed (as they are in the example). In contrast, the 'cartoon' shader has only one unique import modifier entering its dataflow, therefore requiring only one input.

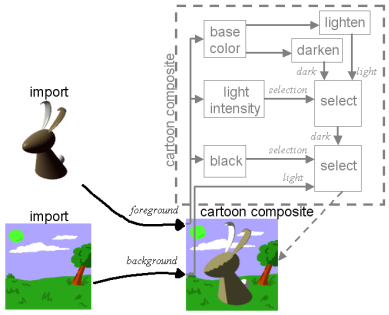


Figure 11: Combining the 'cartoon' and 'composite' modifier

6.4. Creating Complex Effects

Programmers can implement a very limited set of modifiers in a short space of time. Yet even with this limited set, enough functionality is provided so that designers can come up with interesting and diverse effects. This is made possible by the modular capabilities of OPENNPAR. For instance, the designer's task in Figure 12 was simply to play with the interface. Starting from the two images shown at the left, the designer experiments with 10 simple modifiers to come up with a 'sponge-painting' effect shown on the right. Notice, that at every point in the sequence the designer has simply taken a result and directly applied a new effect to it.

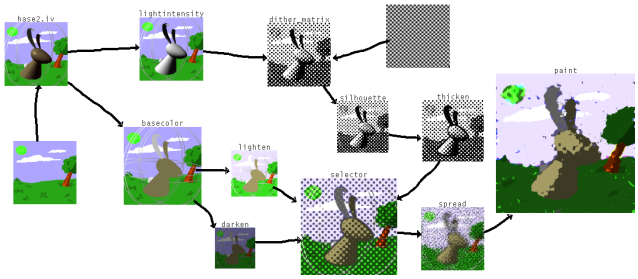


Figure 12: Producing complex effects from modifiers that manipulate modules in a rendering pipeline.

7. Conclusion

We have presented OPENNPAR, a system for creating NPR and animation. OPENNPAR appears to be the first system of its kind that allows for a range of different user classes to both reproduce a variety of algorithms as well as create new ones. This was made possible by structuring OPENNPAR

onto a conceptual framework for NPR that categorizes algorithms and primitives to support the interchange and reuse of data. Consequently, OPENNPAR offers potential for defining an effective presentation method within the wide scope of NPR.

A limitation of the system is that algorithms are constrained to formulations in the scene graph. Thus, certain NPR pipelines utilizing multiple primitives, in particular those requiring feedback loops, require atypical structuring of the scene-graph. This may invoke additional implementation overhead and loss of performance. Although we are using OPENNPAR in both advanced education and research, we have not yet conducted a broad evaluation of the usability of the tool. However, we hypothesize that any existing NPR algorithm can be created by modularizing components into OPENNPAR. The effectiveness of the system lies in the flexibility of available modules and the completeness of its elements.

Development principles have been outlined and demonstrated using examples to effectively extend OPENNPAR's functionality. Programmers can access this functionality either by linking an application directly to OPENNPAR or through textual descriptions of modules in a rendering pipeline. Due to its modular structure, pre-defined effects can be reproduced or entirely new ones created by the manipulation of interchangeable modules.

OPENNPAR also offers a top-down approach to the creation of NPR effects by allowing designers to define which algorithms should be used. In order to support this, a method of computation and interaction was introduced, separating the implementation of effects composition from the interface used to create effects. This allows designers to declaratively specify effects to apply without concern for algorithmic type restrictions or organizational issues of controlling dataflow. Whereas this interactive method sacrifices some control, we do propose that designers, at least in the experimental stage, should be unencumbered by interaction overheads.

In conclusion, OPENNPAR holds vast potential as a tool for the development, augmentation, and creation of NPR. Further details about OPENNPAR can be found in [6] and `www.opennpar.org` which also includes animations and example applications.[1]

References

[1] B. G. Baumgart. A Polyhedral Representation for Computer Vision. In *Proceedings AFIPS National Computer Conference*, volume 44, pages 589–596, 1975.

[2] O. Deussen, J. Hamel, A. Raab, S. Schlechtweg, and T. Strothotte. An Illustration Technique Using Hardware-Based Intersections and Skeletons. In *Proc. Graphics Interface'99*, pages 175–182. Morgan Kaufmann, 1999.

[3] F. Durand. An Invitation to Discuss Computer Depiction. In *Proc. NPAR'2002*, pages 111–124, New York, 2002. ACM Press.

[4] B. Freudenberg, M. Masuch, and T. Strothotte. Real-Time Halftoning: A Primitive for Non-Photorealistic Shading. In *Rendering Techniques 2002: Proc. Eurographics Workshop on Rendering*, pages 227–231, 331, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[5] A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In *Proc. SIGGRAPH'98*, pages 447–452, New York, 1998. ACM SIGGRAPH.

[6] N. Halper. *Supportive Presentation for Computer Games*. PhD thesis, University of Magdeburg, Oct. 2003.

[7] N. Halper, S. Schlechtweg, and T. Strothotte. Creating Non-Photorealistic Images the Designer's Way. In *Proc. NPAR'2002*, pages 97–104, New York, 2002. ACM Press.

[8] T. Isenberg, N. Halper, and T. Strothotte. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. *Computer Graphics Forum (Proc. Eurographics 2002)*, 21(3):249–258, Sept. 2002.

[9] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. Finkelstein. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. In *Proc. SIGGRAPH'2002*, pages 755–762, Reading, MA, July 2002. Addison Wesley.

[10] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. F. Hughes. Art-Based Rendering of Fur, Grass, and Trees. In *Proc. SIGGRAPH'99*, pages 433–438, New York, 1999. ACM Press.

[11] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized Rendering Techniques for Scalable Real-Time 3D Animation. In *Proc. NPAR'2000*, pages 13–20, New York, 2000. Addison Wesley.

[12] B. J. Meier. Painterly Rendering for Animation. In *Proc. SIGGRAPH'96*, pages 477–484, Reading, MA, Aug. 1996. Addison Wesley.

[13] T. Saito and T. Takahashi. Comprehensible Rendering of 3-D Shapes. In *Proc. SIGGRAPH'90*, pages 197–206, New York, 1990. ACM Press.

[14] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin. Interactive Pen-and-Ink Illustration. In *Proc. SIGGRAPH'94*, pages 101–108, New York, 1994. ACM Press.

[15] P. Strauss and R. Carey. An Object-Oriented 3D Graphics Toolkit. In *Proc. SIGGRAPH'99*, pages 341–349, New York, 1992. Addison Wesley.

[16] T. Strothotte and S. Schlechtweg. *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*. Morgan Kaufmann, San Francisco, 2002.

[1] Please note that this domain is no longer active.