OpenFlipper - A Highly Modular Framework for Processing and Visualization of Complex Geometric Models

Jan Möbius*
RWTH Aachen University

Michael Kremer †
RWTH Aachen University

Leif Kobbelt ‡
RWTH Aachen University

Figure 1: Left: Splat rendering of an urban 3D model. Right: Motion tracked Armadillo animation.

ABSTRACT

OpenFlipper is an open-source framework for processing and visualization of complex geometric models suitable for software development in both research and commercial applications. In this paper we describe in detail the software architecture which is designed in order to provide a high degree of modularity and adaptability for various purposes. Although OpenFlipper originates in the field of geometry processing, many emerging applications in this domain increasingly rely on immersion technologies. Consequently, the presented software is, unlike most existing VR software frameworks, mainly intended to be used for the content creation and processing of virtual environments while directly providing a variety of immersion techniques. By keeping OpenFlipper's core as simple as possible and implementing functional components as plugins, the framework's structure allows for easy extensions, replacements, and bundling. We particularly focus on the description of the integrated rendering pipeline that addresses the requirements of flexible, modern high-end graphics applications. Furthermore, we describe how cross-platform unit and smoke testing as well as continuous integration is implemented in order to guarantee that new code revisions remain portable and regression is minimized. OpenFlipper is licensed under the Lesser GNU Public License and available, up to this state, for Linux, Windows, and Mac OSX.

Index Terms: I.3.8 [Computer Graphics]: Applications; K.6.3 [Management of Computing and Information Systems]: Software Management—Software Development, Software Maintanace

1 INTRODUCTION

Most virtual reality environments represent real world objects and avatars using 3D geometry. The underlying data is often acquired using laser scanners, structure from motion, and motion tracking that usually undergo complex processing steps prior to being used in the final application. The presented software framework, OpenFlipper, aims at these geometry processing steps. Beyond that, a wide variety of modern geometry processing applications increasingly incorporate immersion technologies. Prominent examples are: gesture-driven modeling, city reconstruction applications allowing live previews, etc. In this paper, we describe the presented software's architecture that is flexible enough to allow combining both the necessary tools for geometry processing applications and the implementation of immersion techniques such as highly customized rendering stages and the integration of new input devices.

Another important aspect of software development in research environments is the minimization of coding overhead required to process, visualize, and analyze results. Usually, researchers come up with an idea, develop the mathematical and algorithmic core, and implement this on a given framework to limit the implementation overhead. Such a framework is required to ensure that basic functionality, which is required across individual projects, doesn't have to be re-invented. This applies to selection metaphors, I/O, rendering, or many other implementation details, which are not part of the current research project itself. OpenFlipper is already equipped with many commonly used functions as well as data structures for the different data including polygonal, polytopal meshes, skeletons, skeletal animations, B-spline curves and surfaces, and more.

This programming approach requires a highly modular structure of the code, which allows for adjusting the software framework to the specific requirements. OpenFlipper combines a fully modular plug-in system with a very small core implementation. The plugins available up to the current state cover many important functions ranging from file I/O, GUI metaphors, and commonly used geometry processing algorithms to rendering and post processing. The framework allows developers to access almost all parts of the system in order to implement innovative ideas in a rapid, goal-oriented fashion.

Additionally, OpenFlipper integrates a powerful scripting system, which allows for iterative development, i.e. modifying and extending the system at runtime, as well as batch processing, the creation of live demos, screenshots, videos, and many more.

To keep the software stable and avoid regressions between successive code revisions, it is important to add a quality assurance instance to the development process. This significantly reduces the time required to get from research code to commercial grade code since the framework's core, which is reused among individual research projects, is subject to continuous quality tests. The high degree of modularity combined with exhaustive code testing

*e-mail: moebius@cs.rwth-aachen.de
†e-mail: mkremer@cs.rwth-aachen.de
‡e-mail: kobbelt@cs.rwth-aachen.de

of the most essential functions bestows important properties of reusability in software development upon the presented framework. OpenFlipper uses the scripting system to run automatic tests to reveal bugs and regressions in the repositories. Furthermore, the developing process is assisted by a continuous integration system that helps detecting syntactical or logical errors in the code at an early stage, reducing the time to resolve errors in the implementations.

We show the different aspects of the framework and its development at the example of plug-ins commonly required in geometry processing and VR applications like rendering, input device management, GUI metaphors, etc.

2 Background and Related Work

Today there exist various geometry processing frameworks, however, there are still only few which simultaneously fulfill the requirements of geometry processing algorithms and visualizing the results in the context of virtual reality environments. A widely used geometry processing framework is MeshLab [13]. While being portable and extensible, it still does not offer means to use advanced rendering techniques such as stereoscopy which is a useful technique to provide a spatial sensation of the displayed data and to judge the quality of algorithms altering mesh surfaces more accurately. Furthermore, MeshLab only provides limited support for additional input devices mainly used in virtual environments such as spatial navigation devices (e.g. 3Dconnexion's SpaceMouse [1]) or infrared motion tracking systems. Furthermore, it lacks full scripting support allowing to run batch processes, create automated demos or test algorithms with a series of automatically generated parameters and/or input data.

Another component-based software framework is IP3D [16] which is designed to synthesize 3D models and real photographs in order to create realistic virtual worlds. The framework is also extendable but does not focus on animation and real-time applications. A similar approach is the open-source software Bundler [25] that implements a structure from motion technique to reconstruct realistic 3D scenes from unordered image collections. Although OpenFlipper is clearly not targeted at 2D image reconstruction, it is capable of visualizing geometry data generated by Bundler using the splat rendering method described in [26].

In [8] an authoring framework for virtual environments, Colosseum3D, is presented. This software supports rendering complex and realistic 3D scenes, incorporating rigid-body dynamics, 3D sound rendering and avatars used for the virtual identification of the user. Applications in this software can be controlled via a scripting interface using the scripting language Lua [17]. However, this framework only offers limited support for the modular implementation of geometry processing algorithms and handling objects other than polygonal meshes. The VR Juggler software framework presented in [11] is a well-established tool in the field of virtual reality that provides a development interface with several degrees of abstraction from the underlying hardware layer. These abstraction layers, called *virtual platforms*, enable developers not particularly experienced in low-level programming to effectively create applications without putting much effort into complex programming issues emerging from the underlying system. In [21] the authors present an extension to VR Juggler, called VR Jugglua. In this framework the advantages of VR Juggler are combined with OpenSceneGraph [19], an open-source scene graph data structure similar to OpenSG [23], and the widely used scripting language Lua [17]. However, similar to Colosseum3D, both frameworks offer only restricted support for the integration of geometry processing algorithms and operating on geometrical objects other than polygonal meshes (e.g. polynomial curves and surfaces).

Furthermore, CONTIGRA [15] is another component-oriented software toolkit for 3D applications that uses XML to describe the individual components of the virtual world. It is designed to facilitate the authoring and prototyping process as well as offering means for non-programmers to get involved in these development stages. In contrast, due to the inherent limitations of XML being a declarative language, the framework barely offers the possibility to develop customized control flow or low-level functions which is oftentimes needed in the deployment of real-time and/or high-performance applications.

Additionally, there are several other closed-source virtual reality software frameworks worth mentioning in this context. Among these are InstantReality [10] which is a high-performance mixed-reality system that offers many commonly used features and development interfaces, and CoVise [22] that mainly focuses on collaborative visualization of simulation results and other data. Nonetheless, neither of mentioned software systems focuses on geometry processing applications.

In contrast to frameworks such as VTK [24] we want to provide an application front-end that delivers a common look and feel and therefore decided to organize the UI and the plug-ins via the core application.

3 Plug-in Architecture

The OpenFlipper framework is semantically divided into two parts: The core application and the set of plug-ins. The core creates a Qt [5] and OpenGL context as well as the application window and some basic GUI elements. OpenFlipper's GUI is composed of the user interface elements provided by Qt. But, as we support picking, every element in the scene can act as an interaction element (like the coordinate system). Furthermore, OpenFlipper's core provides a very basic rendering system used as a fallback solution on systems with outdated graphics cards or application setups that do not incorporate any rendering plug-in (see Section 4 for further information on that topic). Apart from that, OpenFlipper's core does not contain any further advanced functionality but manages the interaction and communication between plug-ins and organizes the user interface.

Practically all functional units are added individually in terms of dynamically linked plug-in libraries. OpenFlipper's API provides a variety of plug-in interfaces from which plug-ins may inherit in order to access a set of specialized functions. These functions are used for the communication between plug-ins and the core application as well as between different plug-ins. As OpenFlipper is an event-driven architecture, all communication is accomplished by making extensive use of Qt's event system, i.e. signals and slots that are processed with the help of event queues. There exist different types of events, synchronous and asynchronous ones, that can be used in order to provide a powerful way of interoperability between the different parts of OpenFlipper even in multi-thread environments. Up to the current development state, many useful plug-in interfaces providing the following functionalities are available: Selections, rendering, I/O support, interception of input events, logging, integration of GUI elements, scripting, and many more.

In addition to these functionalities, OpenFlipper provides many commonly used data structures. These include, among others, OpenMesh [12] and OpenVolumeMesh [18] for polygonal and polytopal meshes, respectively, and implementations of data structures to handle skeletons, skeletal animations, B-spline curves and surfaces, as well as geometric primitives.

4 Rendering

Undoubtedly, one of the most important parts of a virtual reality framework is the rendering back-end. Like the other components in OpenFlipper, the rendering functionality is suspended to plug-ins as well.

The core of OpenFlipper represents the scene in a hierarchical scene graph data structure. Each object in the scene has a set of corresponding nodes in the scene graph which take care of transforming and rendering the object as well as auxiliary information

such as selections. Additionally, the scene graph contains nodes to control different OpenGL states, e.g. the current material and textures attached to the object. Due to the scene graphs hierarchical structure, the states of a node are also applied to all of its attached child nodes.

The structure is created when objects are loaded in the scene or when they are modified. E.g. if meshes are loaded, they are added to the scene graph as a node, which optimizes them for rendering (Cache optimization [6]). If they get modified, only the required parts of this optimization are recomputed (topology,geometry,selection).

Rendering this scene graph structure is done via OpenFlipper's renderer plug-ins. In order to keep the core as simple as possible, only a very simple renderer is integrated, that allows basic rendering, in case no external renderer plug-in is available. In the publicly available version of OpenFlipper, several rendering plug-ins are contained that support various rendering algorithms and strategies. They have access to the OpenGL context of the viewer and the scene graph. Prior to loading a renderer plug-in OpenFlipper checks whether the system's OpenGL version is sufficient in order to run the plug-in (each renderer plug-ins can use individual OpenGL driver revisions). If the currently installed OpenGL version is insufficient, the core will refuse to load the plug-in in order to avoid unexpected behavior or crashes due to unsupported hardware. The plug-ins are completely independent from each other, such that it is easy to develop and test new rendering algorithms without interfering with existing code.

Furthermore, OpenFlipper allows to split the screen into separate parts, each of which can be processed by a different renderer. This allows for directly comparing the results of the active renderers and using them to highlight different aspects of the objects (e.g. rendering an object using proper material and lighting simulations in one part of the screen, while the other part visualizes the object as a wire frame).

To simplify the implementation of rendering code and to support legacy graphics cards, we provide two different rendering interfaces in OpenFlipper: the classical and the advanced rendering interface. Renderer plug-ins can be derived from either of these interfaces depending on the degree of desired compatibility. The following sections provide a more elaborate description of the basic rendering interfaces.

4.1 Classical Rendering Interface

The classical interface does not support high-performance rendering. It is rather intended to provide means for visualization on legacy systems. For this mode, each node of the scene graph has to provide a draw function that takes care of rendering the object represented by that node. The renderer plug-in itself does not need to know anything about the object to be rendered, as the corresponding OpenGL code is encapsulated in the node. Therefore, it is easy to create new objects and nodes as they only need to implement the draw function and no changes to the external renderer are required.

The drawback of placing the actual rendering code in the nodes is that, if a different visualization is wanted, one has to replace or extend this code. To allow different styles of rendering the draw function in the nodes gets an additional draw mode parameter. These draw modes can be used to switch or combine the visualization (e.g. wire frame or smooth shading). But still all rendering functions reside in the nodes.

When the scene is rendered, the scene graph is traversed by the active renderer plug-in. For each node an enter function is called setting the required OpenGL states. Then the corresponding draw function of the node is called. Afterwards, a leave function resets the OpenGL states to the original ones.

The limitation of this mode is that no optimization can be performed across the objects (e.g. no sorting of objects based on depth, shaders, primitives, materials or shared rendering buffers). Furthermore, the OpenGL states changed between draw calls cannot be controlled by the renderer, which is incompatible with global rendering techniques like Dual Depth Peeling [9]. Moreover, this approach is not seamlessly compatible with shader programming in general. For example, it is unclear when to set the uniforms, as the drawing nodes are independent from the texture, shader, and material nodes, but the uniforms required might depend on all of them. To overcome these problems and to add more flexibility when programming new shaders an advanced rendering interface has been added.

4.2 Advanced Rendering Interface

With the introduction of the advanced rendering interfaces the actual rendering code moves from the scene graph nodes to dedicated rendering plug-ins. This implies that render plug-ins would have to manage the different visualization modes of the individual object types. As a result, new object types would require modifications made to *all* available renderers. To avoid this restriction, a function which returns so called *render objects* is provided by the scene graph nodes. The idea of these objects is that graphic cards only support a fixed set of primitives, i.e. triangles, lines, etc. Therefore, it would be sufficient for the scene graph nodes to generate the required primitive buffers and provide them in a unified data structure. The render objects then contain pointers to OpenGL buffers which include the data to be rendered. Furthermore, they provide information about how the data is organized in the buffer (normals, colors, etc.), which kind of primitive (triangle, point, line, etc.) should be rendered and the material, texture, or other states which have to be applied during the rendering process of an object. Therefore, one node has to provide more than one render object, e.g. if more than one texture or state is used in the object.

These render objects enable a unified draw procedure as the renderer itself can traverse the scene graph, collect the required render objects from the nodes, and start an optimization phase when all data is available. In this optimization the renderer can sort the render objects such that the number of state switches (shaders, textures,...) is minimized. Afterwards, the scene is drawn in that optimized order.

To simplify the creation of new renderer plug-ins, this process is implemented in a renderer base class. It consists of several steps:

1. The scene graph is traversed to collect the render objects.
2. The render objects are sorted and some initial OpenGL states are set.
3. Each render object is processed in newly determined order by binding its buffers, setting the required uniforms for the shaders, e.g. matrices, lighting parameters, etc., and performing the actual draw call.

Note that this setup allows full control of all state changes from within the renderer which allows for the implementation of more sophisticated, modern rendering techniques such as deferred shading. When all objects are drawn, all OpenGL states are reset to their defaults in order to prevent interference with other components of the application.

Due to these predefined functions a simple standard renderer is only a few lines of code calling the different stages. Still it is a small amount of additional code that is required to replace parts of the pipeline (e.g. replace the sorting algorithm) in order to create more advanced renderers. This flexible architecture allows for implementing highly modular renderers customized to meet the requirements of specific hardware configurations, e.g. surround-screen projection systems such as the CAVE [14], and different applications, e.g. non-photorealistic visualizations. Figure 2 shows the data flow of the advanced rendering pipeline.

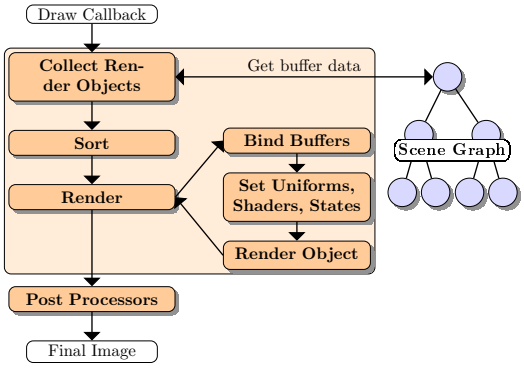


Figure 2: Data flow of the advanced rendering pipeline.

To simplify the construction of shaders, OpenFlipper provides a shader generator that uses template shader code files that can be customized by the renderer plug-ins. These shader template files contain markers which are replaced by custom code blocks at runtime via so called shader modifiers. Basic variables like the current view/projection matrices or materials are automatically added to these template files. Therefore, all required uniform variables are passed to the shader by the rendering system so that the developer does not have to be concerned with their setup. Of course, it is still possible to write entirely customized shaders and to use them in the pipeline. To avoid unnecessary switching and compilation of shaders, a shader cache manages the efficient handling of the shaders, i.e. if a shader is used twice, it will only be compiled and linked once and reused for multiple render objects to avoid overhead.

This interface allows for creating advanced renderer plug-ins like Dual Depth Peeling [9] that require to create additional render targets and shaders to compute transparency in the scene without having to render the objects back to front. Therefore, the plug-in needs full control over the shaders and buffers while rendering several passes of the same scene with different shader setups. Our implementation follows closely the one described in [9].

4.3 Post-Processors

To enable flexible rendering effects, OpenFlipper provides an additional rendering stage, the post-processing. This stage is run on the output of the renderers, i.e. frame buffers, depth buffers, etc. Post-processor plug-ins usually perform image processing algorithms executed on a rendered image but could equally be used to adapt the image to different output devices. Post-processor plug-ins have access to the OpenGL context and can therefore use all available buffers as their input. There are typically two different scenarios that require post-processing:

- Executing image based algorithms which analyze or enhance the images such as the detection of sharp features/corners that may then be accentuated in the final image.
- Reprocessing the images to be displayed on different output media.

One example for the latter is to split the image into segments and stream them to multiple displays (like a video wall consisting of an array of monitors). In this case, the post processor takes care of the splitting operation, compresses the image into a format compatible with the target platform and sends them, e.g. via network, to the display devices. Therefore, the renderers do not need to know anything about the final processing step, except for possible adaptations of the frame buffer's resolution when rendering for high-resolution targets.

4.4 Stereo Support

Another configuration in need of post-processing steps is to generate output used on stereoscopic displays. Therefore, the available rendering plug-ins support a number of techniques for stereoscopy. Currently OpenFlipper supports three different modes:

- OpenGL stereo: In this mode, one image for each eye is rendered. Depending on the available hardware these images can either be displayed at the same time using e.g. polarization filters (passive stereo) or in an alternating way via shutter glasses (active stereo).
- Anaglyph stereo: As many devices do not support direct stereo rendering, OpenFlipper provides means for anaglyphic stereoscopy (see Figure 3 left).
- Auto stereoscopic displays: This is a special mode for some auto stereoscopic displays. They take as input a color image and a depth image and compute an (approximated) 3D view. This additional mode is produced by a simple post processor plug-in which combines the color and depth buffer in one final image (Figure 3 right).

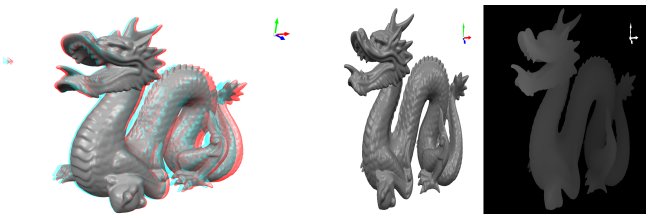


Figure 3: Left: Anaglyph stereo output. Right: Combined output of the color and depth image for auto stereoscopic displays.

One common problem of the interaction in virtual worlds when viewing geometry is the rendering of the mouse pointer. The level of immersion heavily depends on how plausible the rendering of different scene components appears for the human perception. If the scene is rendered and perceived in 3D while e.g. a mouse pointer's positions are restricted to 2D, the user perceives this as highly irritating and the sensation of depth is severely affected. OpenFlipper already includes a pointer infrastructure to render visual pointers at the correct depth. The correct depth is computed automatically such that the pointer is rendered at the same depth as the object behind it. The renderer plug-ins can also get the pointer information and replace the representation with a customized one.

5 Selection Metaphors

The selection of individual entities or groups of entities of objects is a fundamental metaphor widely used in visualization and geometry processing applications. Selections are used to determine regions of interest e.g. to be subject to further editing and/or processing of algorithms. The presented framework supports handling objects of different kinds, such as polygonal meshes, polynomial curves and surfaces (B-splines), volumetric meshes, and many more. Some selection metaphors can be transfered trivially to different kinds of objects, e.g. the selection of vertices of a polygonal mesh and the selection of control points of a B-spline curve. However, this does not apply to all metaphors in general. In many cases, each of these object types consists of characteristic entities that need special handling when it comes to selections. For instance, when selecting a point on a B-spline curve, one might want to specify whether one is interested in selecting the actual point on the curve (thus in the curve's embedding space) or rather determine the corresponding pre-image in the curve's parameter space. In practice, both metaphors require two different selection modes.

From a software-architectural point of view, we solved this issue by splitting up OpenFlipper's selection unit into a hierarchical tree of functionally differing selection layers. At the core is the base selection plug-in that implements—independent from specific object types—a set of elementary metaphors that are commonly shared among most object types, see Section 5.1 for details. In a higher level, there is a set of object specific selection plug-ins. In these plug-ins the individual functionalities tailored to the specific object types is implemented. Apart from informing the application about the supported object dependent entity types, i.e. vertices, edges, etc., they also manage which of the basic selection metaphors, provided by the selection base plug-in, should be accessible for each entity type. The actual selection is also implemented in these plug-ins. Furthermore, one may add individual, object specific selection metaphors in these plug-ins.

The two different layers are described in more detail in Sections 5.1 and 5.2. Figure 4 depicts the underlying hierarchy of the mentioned selection layers.

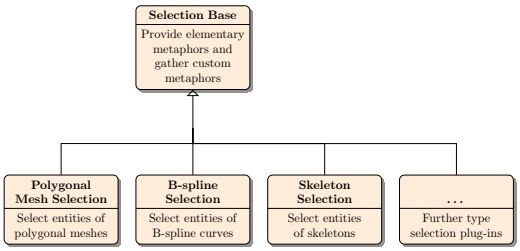


Figure 4: Hierarchy of selection layers in OpenFlipper

5.1 Basic Selection Layer

This layer is independent from specific object types. It provides basic selection metaphors that are commonly shared across multiple object types. Furthermore, it keeps track of all available primitive types as well as custom selection metaphors provided by the object specific selection plug-ins. The set of basic metaphors currently comprises the following operations: Toggle, Surface and Volume Lasso, Sphere, Flood Fill, Boundary, Connected Component.

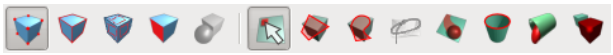
5.2 Object Specific Selection Layer

This layer contains a set of object specific selection plug-ins—one for each object type. During the initialization stage these plug-ins inform the selection base plug-in about the individual entities enabled for selection (e.g., in the case of 2D polygonal meshes, vertices, edges, and faces). In a subsequent step they inform the base selection plug-in about which basic selection metaphor should be enabled for which entity. Additionally, further custom selection metaphors can be added optionally.

Then, while the user interacts with OpenFlipper, whenever a primitive as well as a metaphor is activated for selection, all mouse events are intercepted by the selection base and propagated through all object specific selection plug-ins.

5.3 Selection Data Flow

The object specific selection plug-ins provide information about all available custom selection metaphors, i.e. metaphors not provided by the selection base plug-in. It also provides a mapping of each primitive type to the available metaphors that indicates which metaphor should be enabled for use with a particular entity type. All available primitive types and associated metaphors will then appear in OpenFlipper's GUI as buttons on a tool bar:



If the user activates a primitive type and a metaphor and clicks into the scene, the base plug-in will intercept the event triggered by the input device, determine the currently activated primitive type as well as the selection metaphor and passes this information on to all object specific selection plug-ins. The object specific plug-ins perform the actual picking and, where necessary, the algorithms used for the currently active selection metaphor. They directly modify the states of the respective objects in the scene. After the selection operation is done, they trigger a scene update in order to display the selections. Figure 5 schematically shows the underlying call sequence of a selection operation.

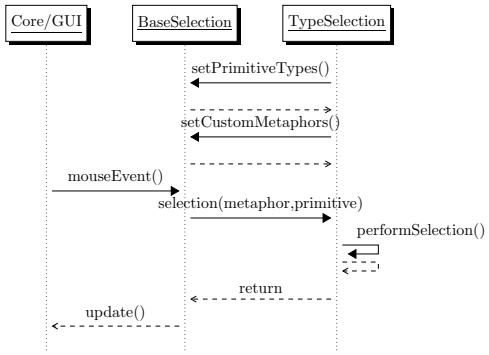


Figure 5: Data flow of a selection operation. All available primitive types and associated selection metaphors are registered in the initialization stage. Selection events triggered by input devices are then passed from the core to the base selection plug-in that triggers the actual selection operation in the object specific plug-ins.

6 Remote Control

An important feature of virtual reality applications and especially cooperative applications is to synchronize states between different instances of the system. OpenFlipper supports synchronization across networks as well as an entire remote control interface using the integrated scripting environment. Section 6.1 describes the basic features and implementation of the scripting system, while Section 6.2 shows its utilization for network synchronization.

6.1 Scripting Interface

OpenFlipper comes with a powerful scripting system. The scripting language is a JavaScript dialect following the ECMA-262 standard [7]. The Qt scripting implementation of this standard is used as a basis for our implementation. Using this interface has the major advantage that exporting functions from the C++ interface of plug-ins to the scripting system only requires the exposure of a function using the public slot qualifier.

The scripting system can be used for various tasks. First of all it is possible to start OpenFlipper in batch mode executing an automated script e.g. to evaluate a series of settings for an algorithm. Secondly, one can control the rendering algorithms to automatically create sets of visualizations or to create live user demonstrations for various output devices. This also simplifies the creation of visual results as one can script the whole process and trigger viewer snapshots in an automated fashion. These snapshots, for instance, can be converted to a video afterwards.

The user interface can also be modified by the scripting interface. It is possible to load additional user interface windows from scripts. This allows for adapting the UI to the user's individual needs without requiring to restart the application.

To simplify the creation of scripts, algorithms can emit notifications that contain information about the current state of their execution. This information contains the ready-to-use scripting commands, that would be required to run the current operation with the currently applied parameters. For example, when performing a selection on a complex model in the scene, the entire command—including the set of indices of the selected entities—is written into OpenFlipper's internal logger. This command can then be copied into a script file so that the exact selection operation on that particular model can be reproduced from within an automated script.

6.2 Network Synchronization

OpenFlipper is equipped with a network interface used to transfer scripting commands between different instances of OpenFlipper. This interface uses the TCP/IP network protocol and is thus capable of synchronizing multiple OpenFlipper instances running on separate machines and/or platforms sharing the same network as well as multiple instances on a single machine. The user determines whether an instance acts as server or client. Operations performed on server instances are sent out to the associated clients which then invoke the respective operation locally. Therefore, instances can be synchronized in various ways. A common application is the synchronization of the view. This is accomplished by simply exchanging scene parameters (such as camera and projection parameters, etc.). For collaborative work, one can also visualize the view parameters of a server instance represented as a viewing frustum being displayed within the scenes of connected OpenFlipper instances as depicted in Figure 6.

Figure 6: The current view of a connected remote viewer is rendered as a viewing frustum.

Furthermore, as the scripting also allows to send more complex commands like selection operations, execution of algorithms, or the previously mentioned modifications to the user interface, the distributed synchronization of a multitude of OpenFlipper's functions can be achieved using the network interface. If some commands are not to be sent or received, they can simply be filtered out from the stream of scripting commands.

When dealing with computationally intense operations, the network interface can be used to distribute the work load among the connected remote instances of OpenFlipper. In order for the processing to be as efficient as possible, in these scenarios, work load packages have to be delegated to the clients in a rather smart way. The flexibility and adaptability of OpenFlipper's network interface facilitates this task.

The implementation of OpenFlipper's network system offers an integrated auto discovery system for instances running in the local network. The role, i.e. server/client/both, of each instance can be configured on-the-fly.

7 Input Handling

Interaction with virtual environments can be done using a multitude of diverse input devices. As OpenFlipper is intended to be usable on a variety of platforms scaling from laptops to high-performance systems using a single displays, large high-resolution monitor configurations, or projection systems, the framework needs a flexible input device handling.

As OpenFlipper is based on the Qt library [5], the standard input of mouse and keyboard events is handled by the library. Nevertheless, the plug-in architecture of OpenFlipper enables easy integration of various other input devices which are more common in virtual environments.

There are two ways of integration for new input devices. The first possibility is to determine a mapping of the device's events to standard input events (e.g. mouse movements, key press events). At the example of the *Wiimote*, the remote control of Nintendo's Wii console, a direct mapping of its movement and button press events onto the corresponding mouse events provides a convenient way of fully integrating the Wiimote as input device in OpenFlipper. This mapping is transparent to all plug-ins in a way that no additional handling of new input events is required—they still deal with mouse events the usual way.

The second possibility is to map events of the input device directly to actions in OpenFlipper. For instance, one can map the signals of a 3D input device, such as the SpaceNavigator [1] or the Wii remote control, directly to camera or object movements. One prominent example for this is the handling of infrared head tracking systems. These systems track the users head position and orientation in 3D space via a set of reflective markers mounted on the head, e.g. a helmet or glasses. The computed position and orientation is then used to update the projection matrices accordingly. Furthermore, one can utilize additional tracking targets in the setup to move and manipulate objects within the scene. The rendering can be configured so that objects are attached to markers to provide the impression of being mounted on top of the marker in 3D space.

The integration of such extended input devices is conveniently implemented in a single plug-in that gathers all data emitted from the device, if necessary performs some preprocessing on it, and forwards the data to other plug-ins using OpenFlipper's event system. This significantly reduces computational overhead and the overall latency of the system.

8 Automated Testing

While developing applications, considerable effort has to be put in the identification and resolution of software bugs. Especially in highly interactive systems, composed of various plug-ins, this can be time consuming as the interaction between plug-ins may have unintended side effects. Additionally, these problems can become worse, if the development team is distributed over several locations and projects. To overcome this, we setup an automated testing system with two stages of quality assurance: Unit, Smoke testing and continuous integration. Section 8.1 describes the integrated testing inside the framework, section 8.2 the infrastructure configured to run the tests.

8.1 Testing Framework

The development pipeline of OpenFlipper is equipped with an integrated framework for testing many components at various implementational levels.

On the lowest level of tests, a series of *unit tests* is performed for various low level functions independent of the core. This may be the creation of spatial trees from polygonal meshes, sorting algorithms or simply a random number generator. The functions which are tested are required to run without any user interface or interaction. This level of testing uses the Google C++ testing framework [3].

In the second level of testing, *smoke tests* make sure the main application is able to start with different combinations of plug-ins. At an early stage, these tests make sure that no memory corruptions

or interferences between plug-in functions are encountered during the start-up process.

The tests are composed of two parts. First off, OpenFlipper is run in batch mode, i.e. without user interface, for the purpose of checking whether the core itself comes up correctly with the plug-ins without GUI elements. They would return an error if plug-ins cannot be loaded due to linking errors (missing symbols) or if plug-ins conflict. If this first start up succeeds, OpenFlipper is run with the user interface to see if the graphical part of the application also works correctly and whether the plug-ins can expose their user interface components to the core.

At the highest level of the testing framework lie the *integration tests*. They check the correctness of algorithms and interaction between plug-ins or even a whole work flow encoded as scripts. Again, OpenFlipper can run in batch mode without a user interface to check the basic components of algorithms or in graphical user interface mode to check the GUI and rendering results. For example, the cache optimizer class provides a smart way of caching the entities of polygonal meshes for efficient rendering. The tests on this unit can be run in batch mode to check whether the optimization algorithm works with different parameters, but nothing is actually rendered. In a next step, the algorithm is run again, but this time the results are rendered, collected and compared against a ground truth data set consisting of snapshots from previous application runs provided by a developer. As the user interface can also be included in the snapshots and is modifiable be the scripting, we can include into the analysis. This way, it is possible to narrow the possible error sources and see, if the underlying algorithm is broken or something goes wrong during the rendering.

As OpenFlipper can also take snapshots via the scripting interface, it is furthermore possible to check if rendered content suffers from regressions. To check renderings for regressions, manually generated snapshots of the expected results are taken as ground truth and compared to the images resulting during the test runs. These comparisons can be parameterized in different ways. The image is not necessarily required to be exactly the same as the ground truth. Therefore, it is reasonable to compare the images based on a threshold with different criteria (color difference per pixel, brightness, histogram of the image, etc.). If the difference exceeds the threshold, the developer is obliged to check the result. If the outlying result is still acceptable, it can be added to a pool of reference images used for successive tests. In some cases it is even reasonable to replace the initial ground truth image. This procedure allows for easy recognition of changes in the renderings and therefore detects regressions during the development of the application.

8.2 Infrastructure

To take advantage of this integrated testing environment, an automated infrastructure is required to run the tests. We use the continuous integration system *Jenkins* [4]. All check-ins into the code repository are automatically analyzed on all supported platforms. This ensures that the code compiles and executes correctly on all supported operating systems and no regressions are introduced with new code revisions.

Furthermore, the code is checked at different levels to keep it as clean as possible. The lowest level is the static code analysis (we use *Cppcheck* [2] for this). The code is analyzed with respect to possible semantic and syntactic errors, compiler warnings, and issues concerning code style. Afterwards, the code is compiled on the different platforms and compilers (MSVC, gcc, clang). The automatic testing process is schematically depicted in Figure 7.

A list of all located issues is sent to the developer who caused them. This significantly improves the code quality and portability. Due to the automatic testing, we can support a rolling release schedule as the repository is kept clean and of high quality. Additionally, the continuous integration server automatically creates setup bundles of all builds (if they succeed).

As OpenFlipper is modular, it is convenient to create different branches for each research project. These branches usually contain a different set of plug-ins. Plug-ins created in the publicly available part of the framework are simply linked to the individual branches such that updates are automatically propagated.

In addition, the framework contains a license management system which supports the creation of closed-source plug-ins and commercial applications requiring only a few lines of code which is described in more detail in [20].

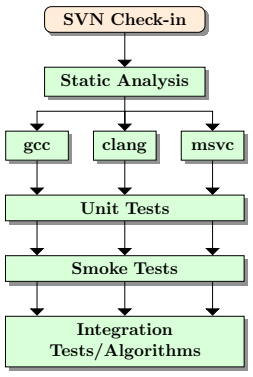


Figure 7: A schematic overview of the automated testing pipeline.

9 Results

Currently OpenFlipper is used in a variety of research and commercial projects. Most of these projects deal with geometry processing and interactive visualization of the results. To get a better insight into the data, it has become more and more important to use immersive displays and input metaphors to visualize and interact with the objects. Therefore the original framework has been extended to support the various output media used for virtual environments and the corresponding input metaphors. In this section we want to show some of the current application domains of the framework.

Generating visually plausible models of cities is usually quite complicated. To simplify the generation process, OpenFlipper is used to visualize point clouds gathered via laser scanners together with generated meshes that represent the models generated from the input data. Figure 8, left, shows a reconstruction from a laser scan.

Another application field of OpenFlipper is the processing and visualization of data captured from infrared motion tracking systems that can be used to animate models. OpenFlipper supports the visualization of this data both offline and online, i.e. real-time. Figure 1, right, shows an example of an animated Armadillo mesh.

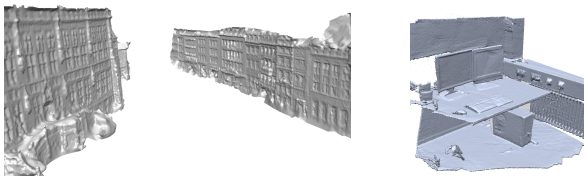


Figure 8: Left: The results of a 3D reconstruction of an urban scene. Shown is a manifold polygonal mesh generated from a laser scan. Right: The reconstruction of a desk using Microsoft's Kinect.

In addition to infrared tracking systems, OpenFlipper provides a plug-in that supports low-precision motion tracking obtained from Microsoft's Kinect. This plug-in gathers skeletal animations captured from the device and transfers them onto skeletons in the scene. Furthermore, the depth information gathered with the tracking data can be used to scan a real scene and map it onto a 3D model as depicted in Figure 8, right.

Figure 9 shows OpenFlipper's interface with multiple active views. Each of these views uses a different renderer plug-in to draw the image.

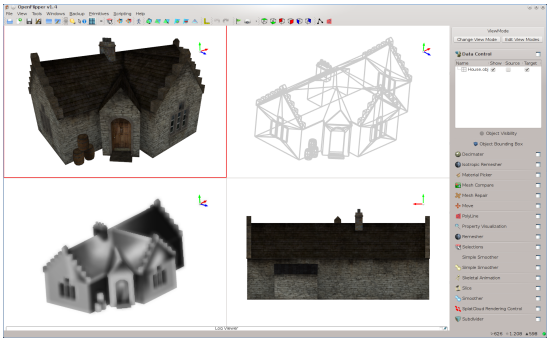


Figure 9: OpenFlipper's multi-view interface with different active renderers.

10 Conclusion and Future Work

The presented framework is a portable and highly flexible platform to easily develop powerful research and commercial applications. Its plug-in architecture provides a high degree of modularity and reusability of the implemented algorithms across different projects. Many essential algorithms are already available in the publicly available code base and can be used as a starting point for new projects.

Furthermore, OpenFlipper contains an easy-to-use scripting system to automate a variety of processes including quality assurance and continuous integration. This automation allows for rolling releases with a minimal amount of human interference.

Although a lot of functionality for VR applications has already been integrated into the framework there are still missing features. For instance, we intend to integrate more input devices such as tracking systems for head mounted displays into future releases. Next to this, we want to render the Qt menus directly into the OpenGL scene to provide a more immersed user interface experience. Furthermore, the implementation of more sophisticated rendering algorithms which improve the image quality, e.g. for our splat based rendering, is planned. Additionally, we want to write renderer modules with OpenGL ES support to be able to support mobile devices in the context of augmented reality scenarios.

In addition to the current use of OpenMP for cache optimization and other geometry processing algorithms, we intend to improve the multi-threading management inside the framework to make better use of multi-core systems in combination with the scripting system.

Acknowledgements

This project was funded by the DFG Cluster of Excellence UMIC (DFG EXC 89). We would like to thank all OpenFlipper contributors for their useful suggestions and the provided implementations. Furthermore, we thank Christopher Tenter for the implementation of various aspects of OpenFlipper's rendering infrastructure as well as Torsten Sattler for contributing point cloud data sets.

References

[1] 3Dconnexion SpaceNav. http://www.3dconnexion.de.

[2] Cppcheck, A tool for static C/C++ code analysis. http://cppcheck.sourceforge.net.

[3] Google C++ Testing Framework. http://code.google.com/p/googletest.

[4] Jenkins, An extendable open source continuous integration server . http://jenkins-ci.org.

[5] Qt cross-platform application and UI framework. http://qt.digia.com.

[6] Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), Aug. 2007.

[7] Standard ECMA-262, ECMA Script Language Specification, 5th edition, 2009.

[8] A. Backman. Colosseum3d authoring framework for virtual environments. In *In Proceedings of EUROGRAPHICS Workshop IPT and EGVE Workshop*, pages 225–226, 2005.

[9] L. Bavoil and K. Myers. Order Independent Transparency With Dual Depth Peeling. Technical report, NVIDIA Developer SDK 10, 2008.

[10] J. Behr, U. Bockholt, and D. Fellner. Instantreality - a framework for industrial augmented and virtual reality applications. In D. Ma, X. Fan, J. Gausemeier, and M. Grafe, editors, *Virtual Reality and Augmented Reality in Industry*, pages 91–99. Springer, 2011.

[11] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. Vr juggler: a virtual platform for virtual reality application development. In *Virtual Reality, 2001. Proceedings. IEEE*, pages 89 –96, march 2001.

[12] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. Openmesh – a generic and efficient polygon mesh data structure. In *OpenSG Symposium*, 2002.

[13] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. Meshlab: an open-source mesh processing tool. In *Sixth Eurographics Italian Chapter Conference*, pages 129–136, 2008.

[14] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the cave. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 135–142, New York, NY, USA, 1993. ACM.

[15] R. Dachselt, M. Hinz, and K. Meissner. Contigra: an xml-based architecture for component-oriented 3d applications. In *Proceedings of the seventh international conference on 3D Web technology*, Web3D '02, pages 155–163, New York, NY, USA, 2002. ACM.

[16] P. Grimm, F. Nagl, and D. Abawi. IP3D - A Component-based Architecture for Image-based 3D Applications. In *SEARIS@IEEEVR2010 Proceedings, IEEE VR 2010 Workshop. ISBN 978-3-8322-8989.8*, pages 47–52, 2010.

[17] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua-an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[18] M. Kremer, D. Bommes, and L. Kobbelt. Openvolumemesh - a versatile index-based data structure for 3d polytopal complexes. In X. Jiao and J.-C. Weill, editors, *Proceedings of the 21st International Meshing Roundtable*, pages 531–548, Berlin, 2012. Springer-Verlag.

[19] P. Martz. *OpenSceneGraph Quick Start Guide*. Skew Matrix Software, 2007.

[20] J. Möbius and L. Kobbelt. Openflipper: An open source geometry processing and rendering framework. In *Proceedings of the 7th international conference on Curves and Surfaces*, pages 488–500, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] R. Pavlik and J. Vance. Vr jugglua: A framework for vr applications combining lua, openscenegraph, and vr juggler. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2012 5th Workshop on*, pages 29 –35, march 2012.

[22] D. Rantzau, U. Lang, R. Lang, H. Nebel, A. Wierse, and R. Ruehle. Collaborative and interactive visualization in a distributed high performance software environment. In M. Chen, P. Townsend, and J. Vince, editors, *High Performance Computing for Computer Graphics and Visualisation*, pages 207–216. Springer, 1996.

[23] D. Reiners. A flexible and extensible traversal framework for scenegraph systems. In *Proc. 1st OpenSG Symposium*, 2002.

[24] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[25] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3d. *ACM Trans. Graph.*, 25(3):835–846, July 2006.

[26] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 371–378, New York, NY, USA, 2001. ACM.