FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Visualization in the cloud

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik

eingereicht von

Lukas Köll
Matrikelnummer 1025785

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Priv.-Doz. Dipl.-Ing. Dr.techn. Stefan Bruckner, Dipl. Ing. Johanna Schmidt

Wien, TT.MM.JJJJ

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

As currently many new visualization techniques are developed, the need for rapid prototyping systems has arisen. Current visualization prototyping software provides extensive features, however it often lacks the possibility to easily start new visualization prototypes as well as the possibility to share and collaboratively work on those prototypes. Also, existing solutions involve a cumbersome and slow development process, because hardware-near solutions often require recompilation after every development step. The availability of hardware resources (e.g. GPU) is limited, hence a remote solution is required to take advantage of them, which also solves the problem of having to transfer large volume datasets. In this thesis, a system named *VolumeShop Playground* is introduced that compensates for the above stated disadvantages while still allowing for hardware-near development of realtime visualizations. VolumeShop Playground is based on the existing VolumeShop framework, enhances it by a scripting API and provides a web frontend for simple setup and collaboration. The fact that we use a modern scripting language instead of recompiling the source code every time will supply a tremendous increase in development speed.

Kurzfassung

Da zur Zeit laufend neue Visualisierungstechniken entwickelt werden, steigt der Bedarf an Rapid Prototyping Systemen. Aktuelle Software zum Prototyping von Visualisierungstechniken stellt umfangreiche Featuresets zur Verfügung, weist jedoch häufig hohen Aufwand beim Erstellen neuer Projekte auf und stellt keine Mechanismen bereit, um diese Prototypen zu teilen und kollaborativ daran zu arbeiten. Außerdem weisen aktuelle Lösungen einen langsamen Entwicklungsprozess auf, da hardwarenahe Programmierung nach jedem Entwicklungsschritt eine Rekompilierung des Quellcodes mit sich bringt. Die Verfügbarkeit von Hardware-Ressourcen (z.B. GPU) ist limitert, daher wird eine Remote-Lösung benötigt um diese Ressourcen auszunutzen, was auch das Problem löst, dass oft große Volumsdatensätze transferiert werden müssen. In dieser Arbeit wird ein System namens *VolumeShop Playground* vorgestellt, das Lösungen für die oben genannten Probleme anbietet und trotzdem hardwarenahe Programmierung Echtzeitvisualisierungen erlaubt. VolumeShop Playground basiert auf dem VolumeShop Framework, erweitert dieses um ein eine Scripting API und bietet ein Web-Frontend an, das Kollaboration und einfaches Bootstrapping erlaubt. Die Tatsache, dass dabei eine moderne Scriptsprache benutzt wird sorgt für eine merkbare Beschleunigung während der Entwickung.

Contents

CHAPTER 1

Introduction - Problem Statement

Research in volume rendering currently develops numerous new techniques and methods. It therefore makes sense to make use of a prototyping framework to speed up the development process for those methods. There are already programs that allow quick development of volume rendering (compare [VWE05] and [SBP+08]) with a dynamic rendering pipeline as well as deferred rendering (compare [vir].

Existing frameworks usually make use of some kind of plugin system, where the rendering technique is being developed as a new component. Methods of this component are then called at defined times during execution of the program to execute certain steps or calculations for the rendering pipeline. Since these systems typically take advantage of advanced Graphics Processing Unit (GPU) features, they require special hardware. Hence, development is limited to computers which are equipped with expensive graphics cards. Also, programmers are required to being familiar with a specific programming language, mostly C++. The workflow during development is rather inefficient, because the whole plugin has to be recompiled for every change to see the resulting rendering output. There is often a high hurdle before a developer is able to start a new plugin because initial setup of the development environment is very specific and often badly documented. Sharing intermediate or final results is made hard, due to the fact that setting up the development environment is bound to high effort. While sending out video demos might be a possibility sometimes, many times it will not be viable, because modern volume visualization often offers interactive components which cannot be tested properly when only shown in a video.

These drawbacks motivate the extension of existing frameworks with a server based, cross platform usable, hardware independent rendering and development component which will presented in this thesis.

CHAPTER 2

Related work - State of the art

There is a wide variety of software tools and libraries for visualization and computer graphics and a comprehensive review would be beyond the scope of this thesis. Therefore this thesis will restrict itself to a discussion of current approaches which share some of the goals of this work. Specifically, some web based visualization applications will be reviewed and their functionality will be compared to the requirements stated in the introduction 1.

2.1 Visualization Engines and Toolkits

The Visualization Toolkit [SML07] is an open source collection of libraries for visualization, image processing and computer graphics. At it's core it is a C++ class library exposing scripting layers to languages like Java and Python. Voreen [MSRMH09] offers visual modelling of data flows through the visualization pipeline. Also, developers can specifically implement their own steps in the pipeline with modern techniques like OpenGL and GLSL to make full use of GPU based rendering. MeVisLab [mev] is a framework for image processing and visualization modules. It has a focus on medical applications. It is written in C++, scriptable via Python and Javascript and offers a generic integration of the Visualization Toolkit mentioned above. Amira [SW05] is a desktop application that offers a full featured image processing, 3D visualization and manipulation library. Amira has a focus on biomedical and industrial data processing and does also support virtual reality features. Finally, Volumeshop [Bru13] is the codebase this work will be based on. Volumeshop offers a prototyping platform for visualization research. It therefore does not provide as much ease for creation of concrete visualizations, instead it aims to provide maximum flexibility for researchers due to it's modular code architecture. Further discussion of the VolumeShop architecture will be done in the Implementation Chapter.

2.2 Remote Rendering

According to [SL07] there are three different classes of remote rendering.

When using **Client-Side Rendering**, the remote application just sends graphics data like meshes and textures to the client, where it is then rendered. This approach works well when no special hardware is required for the rendering. A popular example for this approach is the X-Server.

Server Side Rendering for 2D and Administration is mainly used for administration and remote control. The image representation of the desktop of the server is compressed and sent to the client where it can be viewed. This class has several drawbacks for 3D applications, such as insuffucient framerates and high latency for interaction. A popular example is The Virtual Network Computing Framework (VNC) [RSFWH98].

Server-Side Rendering for 3D Application has specialized abilities regarding latency and high framerates. The GPU of a server is utilized to render the image which is then compressed and sent to the client. Du to the previously stated requirements, this is the class where the proposed *VolumeShop Playground* sytem will be located.

There are also mixed approaches where the rendering is done partly on the server and partly on the client. For example [SNC12], where network latency for games is compensated by putting some of the rendering to the client, using 3D image warping in combination with the change of the point of view. Another mixed approach was presented in [DGE04]. In order to save network traffic, this approach renders lines on the client, based on features that are derived from the server component.

2.3 WebGL Playground

WebGL Playground [web] is a website that lets programmers write rendering programs in Javascript. The architecture is a thick-client and a thin server, as the programmers browser is executing all of the rendering code. A screenshot is given at Figure 2.1. Available methods are the OPENGL-ES specification, as well as the OpenGL ES Shader language (as far as supported by the programmers browser). The rendering code is executed in and by the programmers web browser. WebGL Playground offers the possibility to save and share projects as well as a reasonable documentation for starters.

WebGL Playground has no specific support for volume rendering, altough it should be possible because WebGL playground allows loading of custom resources. What is clearly missing for our purpose here is the explicit handling of volumetric data sets. WebGL Playground also does not provide any classes or data types specific to rendering like vectors, quaternions or matrices. Viewers can only influence the rendering with the mouse cursor but not set any parameters specific to a project. The browser-centered design has several advantages like easy sharing and collaboration, but the lack of libraries that can be used in the project code makes quick prototyping of volume visualizations a cumbersome task.

2.4 Bkcore Shader Editor

As the name suggests, the Bkcore Shader Editor [Des] project is a shader editor only (see Figure 2.2). It follows an approach similar to WebGL Playground with the additional limitation that the only programmer editable parts are one vertex- and one fragment shader. It allows saving

the created shaders to a file as well as creating an HTML link that contains the shader code as a base64 encoded link variable. Volume rendering plugins require usually more than just one rendering stage and also want to make optimal use of graphics hardware, because of which it would be desirable to be able to create a custom pipeline with more advanced shaders like tesselation or geometry.

2.5 SculptGL

SculptGL's [Gin] main purpose is not to create prototype visualizations, but to sculpt 3D meshes. It also has a rendering component, which is realized via WebGL and the HTML canvas element. I chose to review this project in the thesis because it has some interesting features regarding sharing. It is for example possible to import and export to various standard formats like Wavefront and STereoLithography. Further SculptGL allows direct upload of modeled meshes to Sketchfab and Verold via an API key. Sketchfab [Inca] and Verold [Incb] are Web Services that provide simple publishing of interactive 3D content, see Figure 2.3

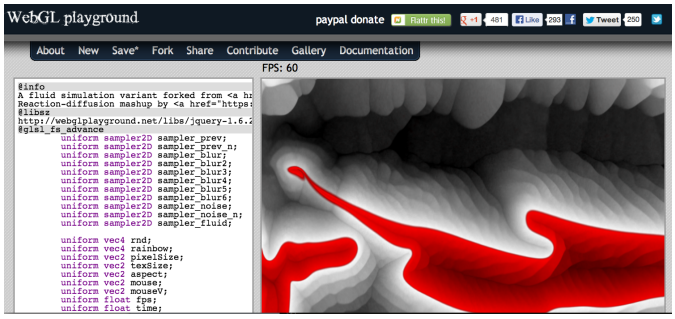


Figure 2.1: Screenshot of the WebGL Playground Website

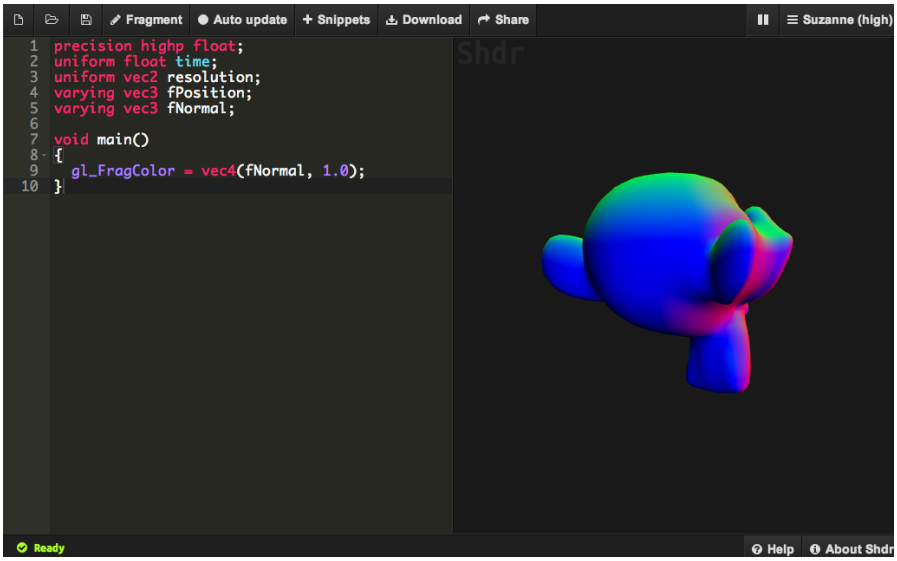


Figure 2.2: Screenshot of the bkcore Shader editor Website

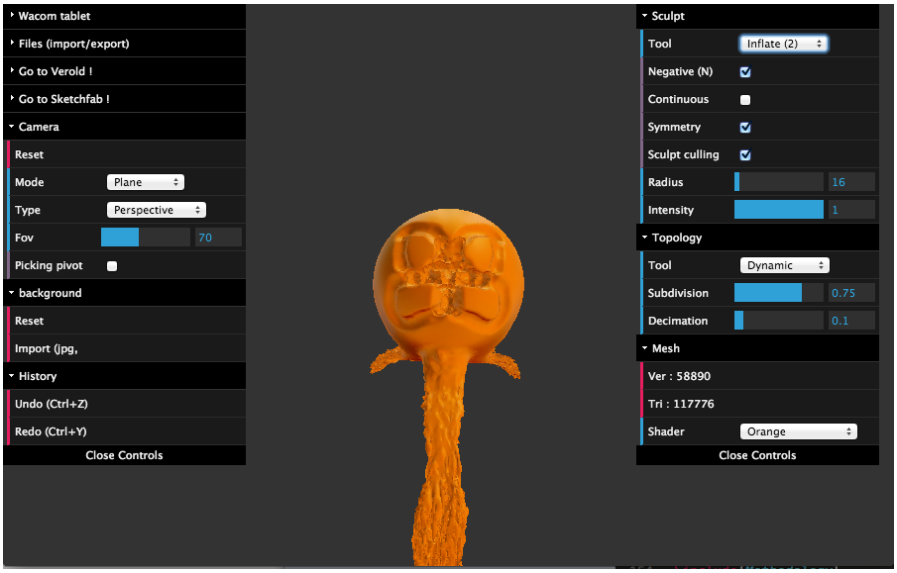


Figure 2.3: Screenshot of the bkcore Shader editor Website

CHAPTER 3

Methodology

The goal of this work was to provide an easy to setup and use development environment for volume visualization researchers with possibilities to easily share progress with other professionals. This environment should also shorten the duration of development cycles. Further, the system should operate on a centralized server to make optimal use of hardware resources and reduce the need to transfer large volume datasets. These constraints motivated a client-server based approach with the client being a modern web browser to keep constraints to a developers machine as small as possible. By providing bindings to a modern scripting language named Squirrel and processing of the scripts during runtime, it is possible to shorten development cycles. The proposed system fits well into a layered architecture, which are described in the Architecture section 3.2.

3.1 User interface development

To guarantee a high usefulness for developers of rendering techniques, the design of the user interface was created in multiple feedback loops. Screenshots of the development stages can be viewed in the Appendix.

3.2 Architecture

The architecture of the proposed extension consists of five layers, of which two (VolumeShop Service and Server) where already implemented at the start of this work. The layers left to plan and implement were the Frontend, the Javascript Remote Layer and the scriptable rendering plugin.

Frontend

On the top level the system consists of an HTML website showing the actual user interface. It is responsible for loading CSS and Javascript required for operation as well as a user friendly

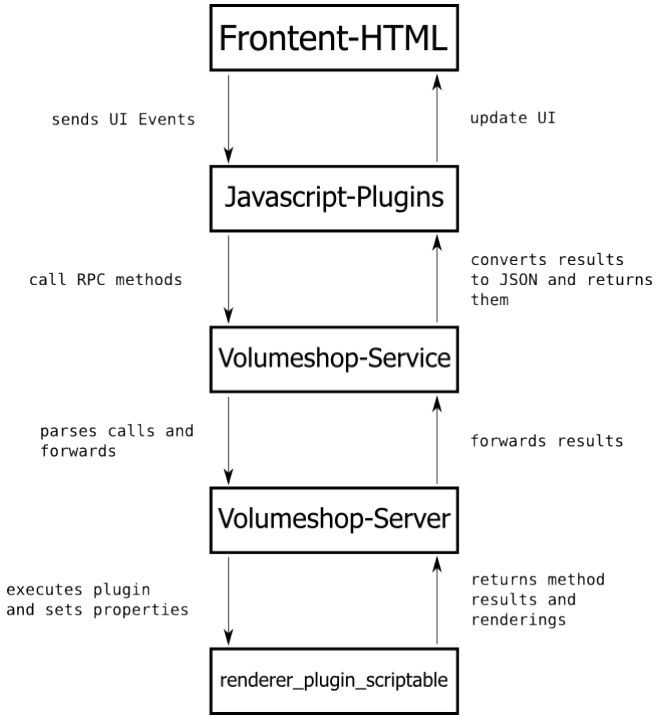


Figure 3.1: VolumeShop Playground Architecture

display.

Javascript Plugins

There are several JavaScript widgets that act as an abstraction layer for calling the VolumeShop service via asynchronous XMLHttpRequests. Most of them are wrappers around service calls to allow future changes in the service implementation without having to rewrite the entire Javascript client code. There are also wrapper methods for a modern responsive user interface.

VolumeShop service

VolumeShop Service is an HTTP RPC Interface to Volumeshop Server. It exposes project loading and saving as well as some public methods of plugins. On the first request, the service component creates a user session and token for subsequent requests to simulate a persistent connection.

VolumeShop server

The VolumeShop Server component is an object server which allows loading of VolumeShop projects and subsequent access to plugins and properties on a session base. This layer is called

by the VolumeShop Service and handles the actual coordination of objects in a project. In particular, it exposes access to the scriptable rendering plugin and its properties.

Scriptable rendering Plugin

The scriptable rendering plugin is a VolumeShop rendering plugin which executes a script for every rendering method, such as $display()$, $idle()$ and $update()$. Whenever the script is saved, it will be recompiled and the updated script methods will be called from the plugin. This allows for an interactive development process, where a recompilation will only take the fraction of a second, whereas recompiling a normal plugin in C++ would take much longer.

CHAPTER 4

Implementation

4.1 Script Binding

To avoid long compilation times during the development process, bindings to a scripting language named Squirrel were implemented.

> Squirrel is a high level imperative, object-oriented programming language, designed to be a light-weight scripting language that fits in the size, memory bandwidth, and real-time requirements of applications like video games. [Dem]

To allow calling of VolumeShop methods and usage of VolumeShop objects, it was required to bind them correctly to the Squirrel virtual machine. Although this could be done directly via the Squirrel API, a template based framework for Squirrel binding named Sqrat was used.

> Sqrat is a C++ library for Squirrel that facilitates exposing classes and other native functionality to Squirrel scripts. It is similar to SqPlus, both in functionality and syntax, but seeks to address several issues present in other binding libraries. [Haf]

This choice was mainly motivated by better readable and shorter binding code. Also, usage of Sqrat does not forbid usage of native binding code next to Sqrat bindings. So no negative implications for development are to be expected even if a future developer should decide not to use Sqrat anymore.

VolumeShop Binding

Most VolumeShop classes are bound to Squirrel in a way that VolumeShop programmers could use them inside a Squirrel script just as they would in C++. Due to the differences of C++ and Squirrel (for example, C++ is typesafe while Squirrel is not), there are some specifics when using the scripting interface which will be explained here.

Arrays Methods returning C++ arrays will be copied and returned as native Squirrel arrays instead. Conversely, methods expecting C++ arrays as an argument expect a Squirrel array in the script. The conversion from Squirrel to C++ and vice versa is done automatically in the binding code.

Listing 4.1: native C++ array

```
1    Matrix myMatrix;
2    float[] arr = myMatrix.Get();
```

Listing 4.2: Squirrel array

```
1    local myMatrix;
2    local arr = myMatrix.Get(); // arr is a native Squirrel
        array!
```

Property Variants VolumeShop saves its properties as Variants which can have various types ranging from $float$ over $Matrix$ to $Quaternion$. When requesting property values via *GetProperty()* or setting property values via *SetProperty()*, the binding will automatically convert value objects to variants and backwards. Also, all Variant Types that exist as of the writing of this thesis are bound to Squirrel. So inside the script, no further conversion is required.

OpenGL Binding

A big part of the work was setting up Squirrel bindings to the OpenGL API. Binding the whole specification would not have brought any serious advantages, so only the OpenGL ES specification Version 2.0 [Gro] was bound. A full list of the methods available in Squirrel is available in the Appendix. The native OpenGL API consists of methods and constants. The code for binding constants via Sqrat looks like in Listing 4.3

Listing 4.3: Binding of OpenGL constants to the Squirrel VM via Sqrat

```
1  Sqrat::ConstTable().Const("GL_SRC_ALPHA", GL_SRC_ALPHA);
2  Sqrat::ConstTable().Const("GL_STATIC_DRAW", GL_STATIC_DRAW);
3  Sqrat::ConstTable().Const("GL_STENCIL_ATTACHMENT_EXT",
      GL_STENCIL_ATTACHMENT_EXT);
```

Binding methods to Squirrel was a bit more elaborate, because data types between C++ and Squirrel do not always match. A showcase implementation for binding the method glUniformMatrix4fv can be seen in Listing 4.4. The method expects to be called with a Squirrel array as the third parameter, but the native method expects a native float pointer, so inside the binding code a conversion has to be made.

Listing 4.4: Binding of the OpenGL glUniformMatrix4fv method via Sqrat, converting the Matrix to a float pointer

```
1  SQInteger sq_glUniformMatrix4fv(HSQUIRRELVM v){
2    if(sq_gettop(v) == 5){
3      Sqrat::Var<GLint> location(v, 2);
4      Sqrat::Var<GLsizei> count(v,3);
5      Sqrat::Var<GLboolean> transpose(v, 4);
6
7      float arr[16];
8      Sqrat::Var<Sqrat::Array> data(v, 5);
9      for(int i = 0; i < 16; i++){
10       data.value.GetElement(i, arr[i]);
11     }
12     glUniformMatrix4fv(location.value, count.value,
           transpose.value, &arr);
13   }
14   return 0;
15 }
```

As Squirrel does not provide object references or pointers, some OpenGL functions are not possible to bind in the same way as in the native C++ API. They will be discussed in this section.

glGenBuffers and similar functions expect an output variable pointer to be passed which will then receive the generated buffers. As it is not possible in Squirrel to create a reference to an object, the binding code instead creates a temporary variable and returns that, see Listing 4.5.

Listing 4.5: Example for usage of glGenBuffers

```
1    local m_uVertexBuffer = glGenBuffers(1); //
       m_uVertexBuffer is either an int or an array of ints
```

glUniformMatrix4fv and similar functions. When passing value pointers to openGL functions, one instead has to pass a Squirrel array which will internally be converted to a native C++ array and then passed to the OpenGL API by pointer.

Similar Bindings exist already for other languages, such as the Java OpenGL Bindings [Pro14]. An extensive list of available objects and their methods can be found in the Appendix.

4.2 Creation of a scriptable rendering plugin

The scriptable rendering plugin is a VolumeShop plugin as described in [Bru13]. Such a plugin has at least three methods, namely *idle()* which is called continuously during the main loop, *display(Canvas& canvas)* which is only called when anything in the rendering pipeline changes

and *reshape(unsigned int newWidth, unsigned int newHeight)* which is called on every change of the viewport size.

These methods only contain calls to the Squirrel functions that are defined in the script. See Listing 4.6 and 4.14 for an example.

Listing 4.6: Implementation of the reshape() method in the C++ VolumeShop rendering plugin

```
1 void RendererTestPlugin::reshape( const unsigned int uWidth,
    const unsigned int uHeight )
2 {
3   sqBinding->reshapeFunc(uWidth, uHeight);
4 }
```

Listing 4.7: The corresponding reshape() method in the Squirrel script

```
1 function reshape(uWidth, uHeight){
2   if (uWidth != properties.viewportWidth || uHeight !=
       properties.viewportHeight)
3   {
4     properties.viewportWidth = uWidth;
5     properties.viewportHeight = uHeight;
6     properties.m_bReshape = true;
7     updatePlugin();
8   }
9 }
```

The scriptable renderer also has a member variable which contains an *HSQUIRRELVM* object that contains all the bindings to VolumeShop and OpenGL required for development. Upon a script change, the Squirrel VM is being replaced by a new one so that calls to script methods will be executed with the new code.

For VolumeShop Playground, two methods named *setup()* and *teardown()* have been introduced. *setup()* is called after compilation of a new script, while *teardown()* is called before destruction of the old Squirrel VM. These methods are meant to allow a script programmer to properly initialize and clean up his resources.

The scriptable renderer is controlled by exposing properties to the VolumeShop Service. Apart from metadata like the project name and description, which are shown directly in the frontend (see Section 4.3), this plugin exposes the following 3 properties:

Script Code, which is of type string, contains a base64 encoded version of the script code that the programmer wrote in the browser. It is set by the JavaScript layer whenever the programmer saves his code.

Script Compilation Errors, which is of type string, contains errors in human readable form that occured during the compilation of the Script Code. This property is read by the JavaScript layer after it received a notification that saving of the Script code was successful. Then a graph-

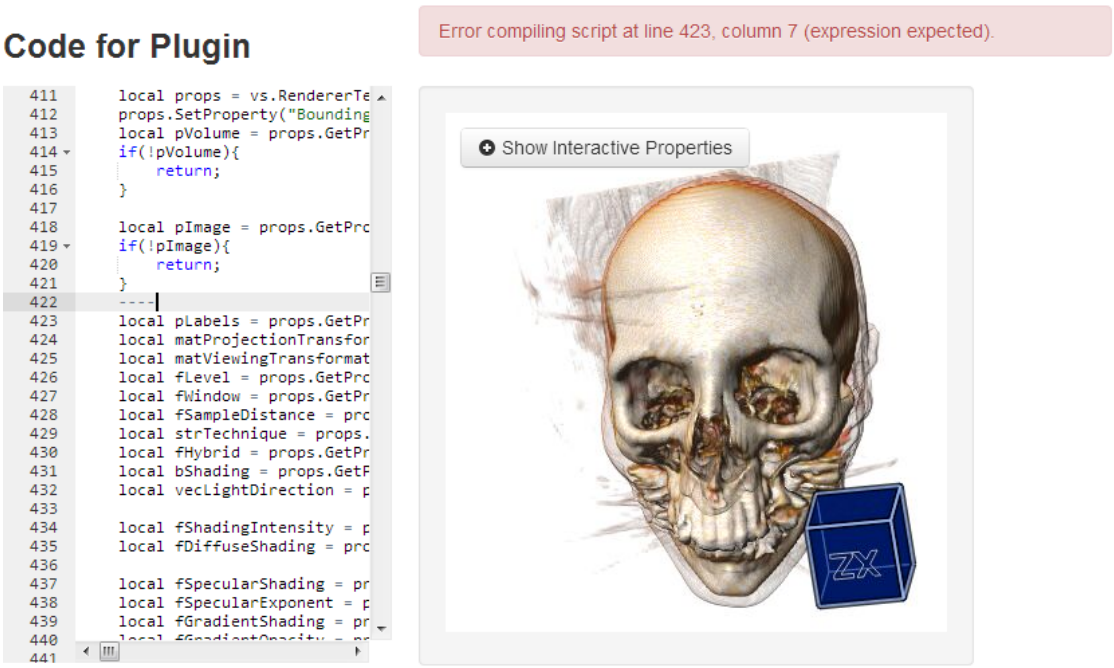


Figure 4.1: Compiling an erronous script will result in an error message showing the line, column and a hint regarding the compilation error

ical notification is created to that the programmer knows if his source compiled successfully or, if not, where to look for the error (see Figure 4.1).

Reload Script, which is of type boolean, is an indicator for the scriptable plugin to recompile its script. It is set via the Javascript layer whenever the code editor receives a save() event. In the plugins native display() method, it always checks for this property to be true, in which case the recompilation process is triggered. The recompilation involves calling the teardown-method of the old script, compiling the new script, binding script methods and calling the setup-method of the new script. In case of compilation failure, the state of the old Squirrel-VM is restored and the according error message set to the *Script Compilation Errors* property.

4.3 HTML Frontend development

The frontend has been developed utilizing the Bootstrap framework [OT]. Bootstrap offers a basic responsive, fluid, mobile ready and cross browser compatible template for websites which made it an ideal candidate for quick prototyping and further use for VolumeShop Playground. Bootstrap also comes with a variety of predefined CSS components for user interface elements, such as buttons, tables and dropdowns. Please see the web page for a comprehensive list of its features. The frontend is partitioned into multiple views, a structural overview of these views is given in Figure 4.2.

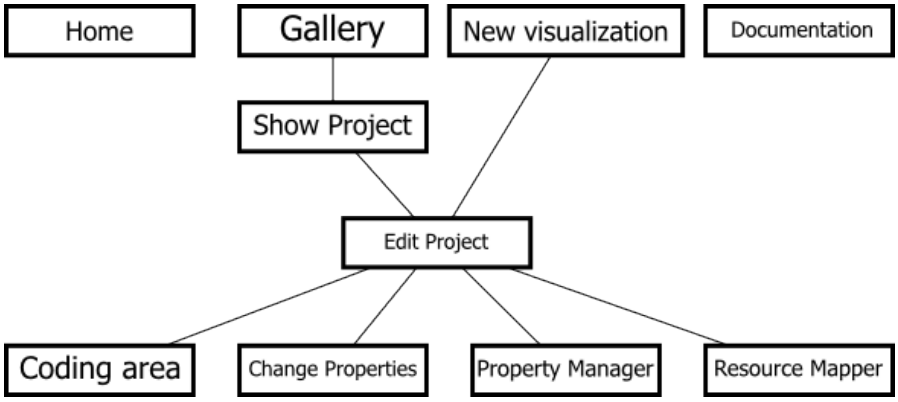


Figure 4.2: Structural Overview of the Frontend views

The gallery browser is meant to give a brief overview of existing volume rendering projects. The Javascript remote layer is calling a remote method on the VolumeShop server which returns a list of all currently existing projects. Then, for each of the projects, a new HTML element is created and inserted into the page. Clicking a project name leads to the show project view (Figure 4.3).

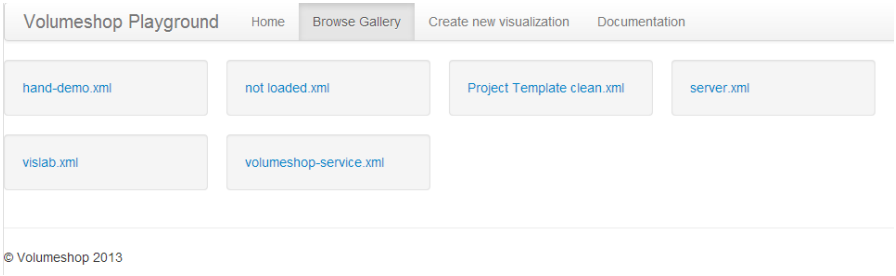


Figure 4.3: Screenshot of the Gallery Browser

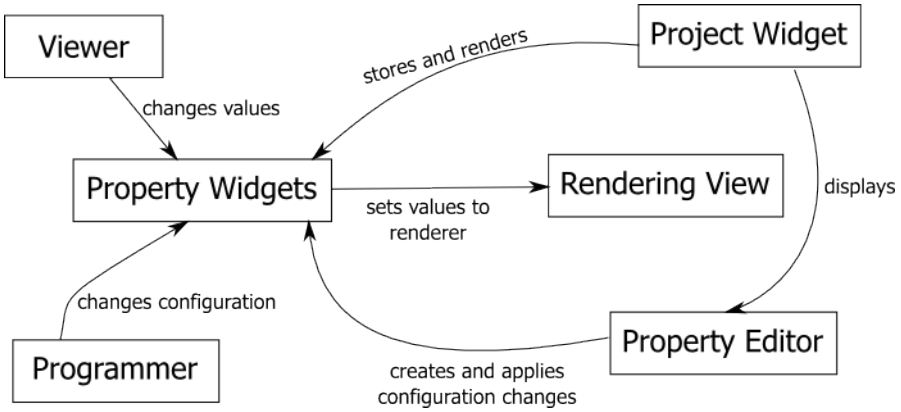
The show project view is for demonstration purposes. It shows some project metadata like the author and a description as well as an interactive rendering window. It allows a visitor to interactively explore the impact of the interactive properties that where configured by the author in the project edit view (Figure 4.4).

The edit project view is split vertically where the right part always shows the current rendering output(referenced to as the rendering panel), while the left part is navigatable via a submenu and contains one of the following elements:

Edit project settings allows the user to change project metadata such as the author and description. It makes use of the JavaScript widget described in 4.4 to save changes of the data to the VolumeShop project.

Figure 4.4: Screenshot of the Show Project View

Figure 4.5: Architecture of the Javascript Layer



Plugin code provides the interface for developing the plugins Squirrel script code. To provide a convenient code editor, it embeds the ACE-Editor jQuery plugin [B.V]. On saving the script the updated rendering output will be instantly visible in the rendering panel.

Property Manager allows editing of a plugins interactive properties. Here a plugin author can expose certain properties used in the script code to a viewer, control how those properties should be rendered and impose client side constraints to valid property values. Changes to interactive properties will be visible instantly in the rendering panel so it is easy to see the consequences of a change.

4.4 Javascript Remote Layer Development

Most of the JavaScript remote layer has been implemented as jQuery stateful plugins [jF]. The Steam- and Remote-Plugin from VolumeShop Service have been reused. The architecture of the JavaScript layer can be seen in Figure 4.5

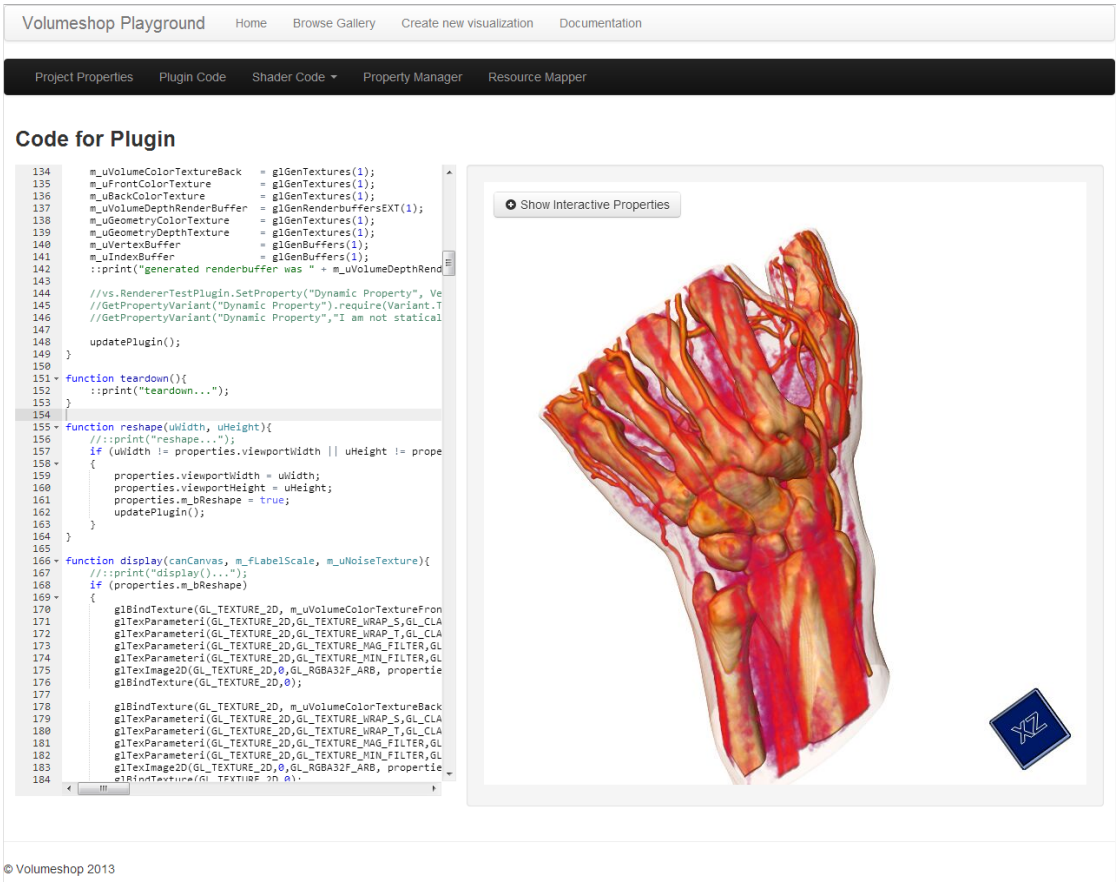
Project Widget

The project widget contains references to relevant html DOM nodes like the rendering canvas, property widgets, the property editor and the code editor. It provides methods for loading and saving projects, rerendering the view and saving the script code as well as the interactive property configuration. The code editor (see Figure 4.6) is shown via the ACE Editor jQuery plugin. The project widget then registers to be notified when the editor content is to be saved (in this case when Control-S is pressed on the keyboard) and saves the new code to the renderer plugins *Script Code* property.

Property Widgets

To make project properties interactively editable it was required to create user interface elements for these properties. Currently four types of property widgets are implemented: *Boolean* is rep-

Figure 4.6: The code editor realized via ACE Editor



resented by a checkbox that can be enabled or disabled by users to point to a 1 or 0 value. *float* is represented by a jQuery UI slider. Programmers can limit the value range in form of a minimum and maximum value. *Options* is represented by a selectbox from which a user can select one single option. Possible values are entered by programmers as a newline separated string. *Resource* is a special form of an option. It contains as possible values all volume resources connected to a project. This allows the user to quickly change between different volumes without leaving the visualization session. An example of possible influence on the rendering can be seen in Figures 4.7 and 4.8. The programmer is however able to use properties to influence his rendering script in whichever way desired.

Property Editor Widget

Another component is the property editor (see Figure 4.9) . It allows programmers to setup and configure property widgets. These properties can later be used inside the plugin script to give viewers the possibility to interactively change rendering parameters.

Figure 4.7: Rendering when the Technique property is set to Direct Volume Rendering

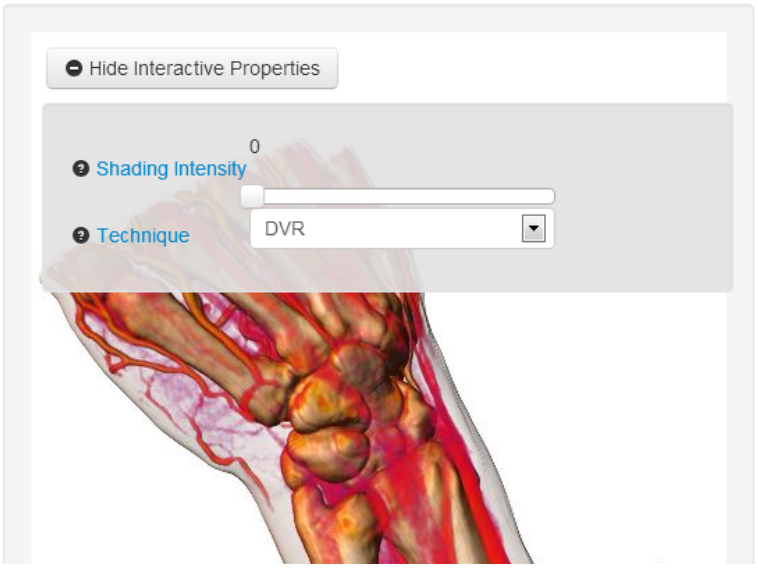


Figure 4.8: Rendering when the Technique property is set to Maximum Intensity Projection

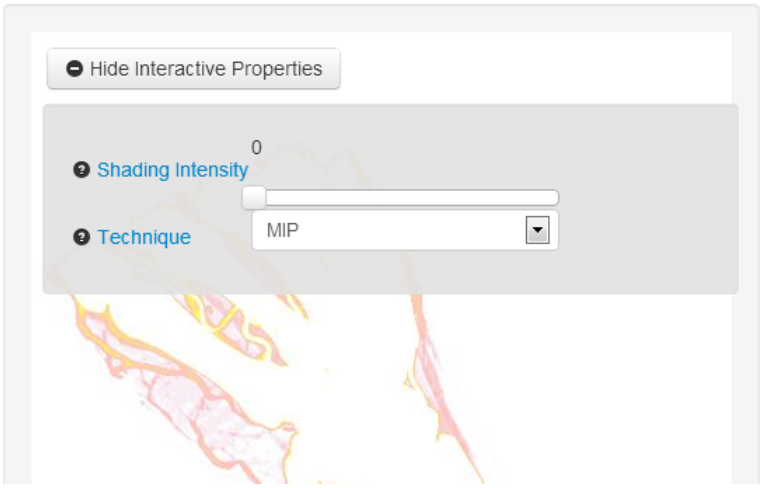
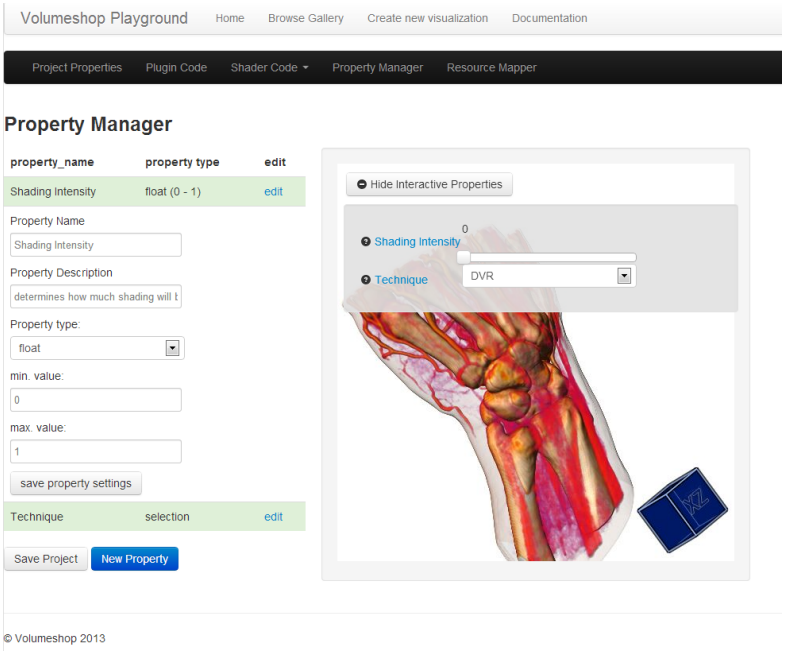


Figure 4.9: The property editor widget



4.5 How to expose more functionality to Squirrel

Exposing classes

When extending scripting functionality of the scriptable renderer plugin, the first location to look out for is the *bindSquirrel*-method in the file *src/SqBindings.h*. *SqBindings.h* includes the header files that contain the actual binding code for a specific class.

An example for Squirrel Bindings: The *Timer*-Class:

Listing 4.8: Content of vsbindings/TimerBindings.h

```
void bindTimer(){
  Sqrat::Class<Timer> timer;
  timer.Func("getFloat", &Timer::operator float);
  /* myTimer.getFloat() in Squirrel translates to
  the (float)-Operator in C++
  */
  timer.Func("start", &Timer::start);
  /* myTimer.start() in Squirrel translates to
  myTimer.start() in C++
  */
  Sqrat::RootTable().Bind("Timer", timer);
  // in Squirrel, this Class is named "Timer"
}
```

In *SqBindings.h*, it looks like this:

Listing 4.9: calling the bindTimer()-Method

```
void bindSquirrel(HSQUIRRELVM vm, RendererTestPlugin* plugin)
{
  ...
  bindTimer();
  ...
}
```

Exposing OpenGL Methods and Constants

OpenGL Bindings are stored in the *src/vsbindings/OGLBindings.h* file.

Methods are bound like this:

Listing 4.10: Binding OpenGL Methods

```
void bindOglMethods(){
  ...
```

```
3   Sqrat::RootTable().SquirrelFunc("glTexCoord2f",
       &sq_glTexCoord2f);
4   // I use SquirrelFunc to apply some parameter conversion
       where needed
5   ...
6 }
```

When using SquirrelFunc, a separate method is required like this:

Listing 4.11: Custom

```
1 SQInteger sq_glTexCoord2f(HSQUIRRELVM v){
2   if(sq_gettop(v) == 3){   // require 2 parameters
3     Sqrat::Var<GLfloat> s(v,2); // load param 1 as float
4     Sqrat::Var<GLfloat> t(v,3); // load param 2 as float
5     glTexCoord2f(s.value, t.value); // call the actual OpenGL
         method
6   }
7   return 0; // no return values
8 }
```

Constants can be bound like this:

Listing 4.12: Binding OpenGL Constants

```
1 void bindConstants(){
2   ...
3   Sqrat::ConstTable().Const("GL_TEXTURE4", GL_TEXTURE4);
4   ...
5 }
```

Dynamic behavior of Squirrel methods

When using SquirrelFunc for binding, Sqrat allows a method to behave differently, depending on the number and type of arguments that are supplied to the Squirrel function. Such distinctions have to be made at runtime of the Squirrel VM, which makes them comparatively slow.

Lets see an example on how to make a dynamic matrix multiplication, depending on wether a matrix, float or vector is the multiplicator. First, we bind the multiplication operator to our custom function *sq_Matrix_operatorMul*:

Listing 4.13: Matrix Multiplication Bindings

```
1 void bindMatrix(){
2   ...
```

```
  // default implementation of the C++ Matrix class for
     transpose
  matrix.Func("transpose", &Matrix::transpose);

  // _mul is the multiplication operator, we handle that
     ourselves
  matrix.SquirrelFunc("_mul", &sq_Matrix_operatorMul);
  ...
}
```

Then, we define that custom multiplication function:

Listing 4.14: Matrix Multiplication Bindings

```
SQInteger sq_Matrix_operatorMul(HSQUIRRELVM v){
  if (sq_gettop(v) == 2) { // one argument, (v,1) is class or
     environment

    // so now... matrix * matrix
    Sqrat::Var<const Matrix&> me(v, 1);
    Sqrat::Var<const Matrix&> multiplicator(v, 2); // multiply
       with another matrix
    if (!Sqrat::Error::Instance().Occurred(v)) {
      Matrix result = me.value * multiplicator.value;
      Sqrat::PushVar(v, result);
      return 1;
    }
    Sqrat::Error::Instance().Clear(v);

    // now matrix * float
    Sqrat::Var<float> multiplicatorF(v, 2);
    if (!Sqrat::Error::Instance().Occurred(v)) {
      Matrix result =  me.value * multiplicatorF.value;
      Sqrat::PushVar(v, result);
      return 1;
    }
    Sqrat::Error::Instance().Clear(v);

    // now matrix * vector
    Sqrat::Var<const Vector&> multiplicatorV(v, 2);
    if (!Sqrat::Error::Instance().Occurred(v)) {
      Vector result =  me.value * multiplicatorV.value;
      Sqrat::PushVar(v, result);
      return 1;
```

```
29      }
30
31      Sqrat::Error::Instance().Clear(v);
32      Sqrat::PushVar(v, "oh_noooes you cannot do that to a
           matrix!");
33      return 1;
34      }else{
35        // no arguments is invalid
36        return 0;
37      }
38    }
```

So we try loading the second argument as matrix, if an error occurs on that (i.e. loading as a matrix was not successful), we proceed to the next try - matrix * float, and if that doesn't succeed, we finally try matrix * vector. The returned value of the Squirrel function is each pushed on the stack and the number of return values (1) returned.

A full documentation of the Sqrat Binding Library [Haf] and the Squirrel Scripting language [Dem] can be found on the web.

CHAPTER 5

Conclusion

An extension to the existing VolumeShop framework was presented that allows for rapid visualization prototyping using only a modern Web Browser. A scripting API to OpenGL calls as well as VolumeShop internal classes and objects reduces the duration of development iterations by avoiding recompilation of the whole renderer plugin. Instead the plugin methods relevant to rendering can be implemented in a script which is then recompiled dynamically during runtime, thus not bringing the long compilation times of C++.

As of now, no security measures have been taken at all to ensure integrity of user input or enforce any type of access control. Also, not the whole OpenGL specification has been bound for Squirrel scripting, so that would be a point to be further elaborated. Further work could also include interactive debugging of Squirrel scripts. Breakpoints in the Squirrel scripts and live variable inspection would require a full binding of all Volumeshop classes as well as some deeper thought about senseful representation of objects like volumes, big arrays etc.

APPENDIX A

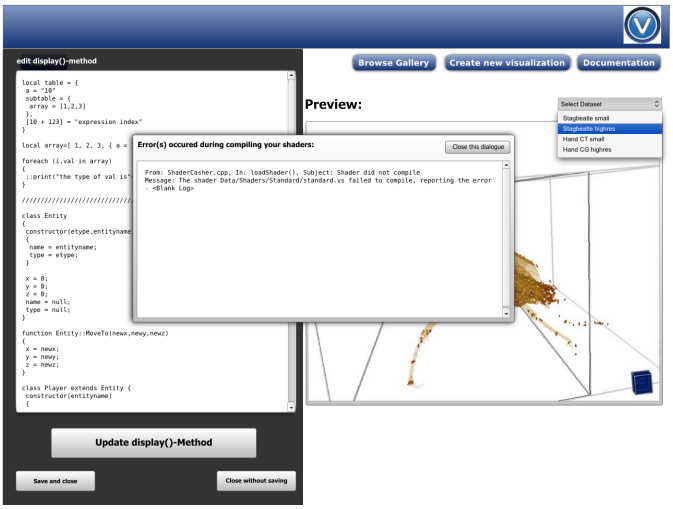


Figure A.1: First Version of the Frontend Design

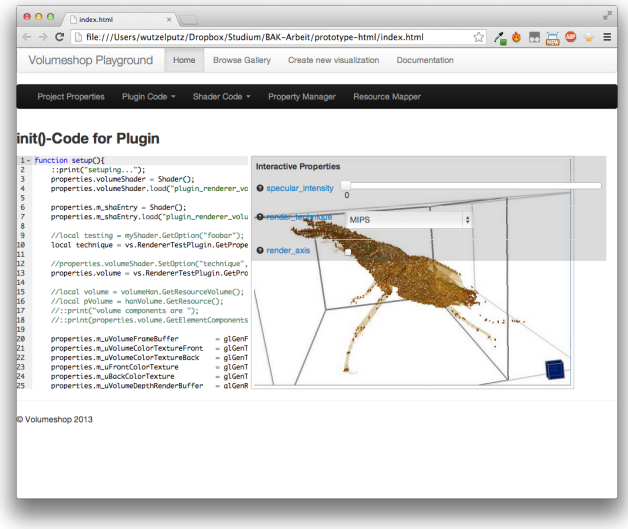


Figure A.2: 2nd Version of the Frontend Design

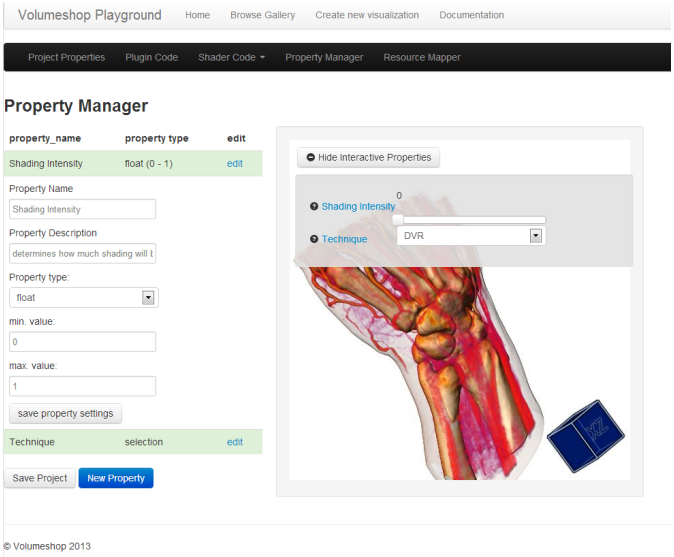


Figure A.3: Final Version of Frontend Design

Appendix

A.1 Prototype Frontend Designs

A.2 Squirrel Available OpenGL Constants

GLEW_ARB_fragment_shader	GLEW_ARB_vertex_shader
GLEW_EXT_framebuffer_object	GL_ARRAY_BUFFER
GL_ALL_ATTRIB_BITS	GL_ALWAYSG
GL_BACK	GL_BLEND
GL_CLAMP	GL_CLAMP_TO_EDGE
GL_COLOR_ATTACHMENT0_EXT	GL_COLOR_ATTACHMENT1_EXT
GL_COLOR_ATTACHMENT2_EXT	GL_COLOR_ATTACHMENT3_EXT
GL_COLOR_BUFFER_BIT	GL_CULL_FACE
GL_DEPTH_ATTACHMENT_EXT	GL_DEPTH_BUFFER_BIT
GL_DEPTH_STENCIL_EXT	GL_DEPTH_TEST
GL_ELEMENT_ARRAY_BUFFER	GL_EQUAL
GL_FALSE	GL_FLOAT
GL_FRAMEBUFFER_COMPLETE_EXT	GL_FRAMEBUFFER_EXT
GL_FRONT	GL_GREATER
GL_INDEX_ARRAY	GL_INVERT
GL_INT	GL_KEEP
GL_LESS	GL_LINEAR
GL_MODELVIEW	GL_NEAREST
GL_ONE	GL_ONE_MINUS_SRC_ALPHA
GL_ONE_MINUS_DST_ALPHA	GL_PROJECTION
GL_QUAD_STRIP	GL_QUADS
GL_RENDERBUFFER_EXT	GL_RGBA
GL_RGBA32F_ARB	GL_SRC_ALPHA
GL_STATIC_DRAW	GL_STENCIL_ATTACHMENT_EXT
GL_STENCIL_BUFFER_BIT	GL_STENCIL_TEST
GL_TEXTURE0	GL_TEXTURE1
GL_TEXTURE2	GL_TEXTURE3
GL_TEXTURE4	GL_TEXTURE5
GL_TEXTURE6	GL_TEXTURE7
GL_TEXTURE_2D	GL_TEXTURE_3D
GL_TEXTURE_MAG_FILTER	GL_TEXTURE_MIN_FILTER
GL_TEXTURE_WRAP_S	GL_TEXTURE_WRAP_T
GL_TRIANGLES	GL_TRUE
GL_UNIFORM_BUFFER	GL_UNSIGNED_INT
GL_UNSIGNED_SHORT	GL_VERTEX_ARRAY

A.3 Squirrel Available OpenGL Methods

glActiveTexture	glBegin
glBindFramebufferEXT	glBindBuffer
glBindRenderbufferEXT	glBindTexture
glBlendFuncSeparate	glBufferData
glCheckFramebufferStatusEXT	glClear
glClearColor	glClearDepth
glClearStencil	glColorMask
glCopyTexSubImage2D	glCullFace
glDepthFunc	glDepthMask
glDisable	glDisableClientState
glDrawBuffer	glDrawBuffers
glDrawElements	glEnable
glEnableClientState	glEnd
glFramebufferRenderbufferEXT	glFramebufferTexture2DEXT
glFinish	glGenBuffers
glGenTextures	glGenFramebuffersEXT
glGenRenderbuffersEXT	glIndexPointer
glLoadMatrixf	glLoadIdentity
glMatrixMode	glMultiTexCoord2f
glMultMatrixf	glPopAttrib
glPushAttrib	glRenderbufferStorageEXT
glStencilFunc	glStencilOp
glTexCoord2f	glTexParameteri
glTexImage2D	glUniform1i
glUniform1iARB	glUniform1f
glUniform2fvARB	glUniform3fvARB
glUniform3fv	glUniform4fv
glUniformMatrix4fv	glVertex2f
glVertex3fv	glVertexPointer
glViewport	

A.4 Squirrel Available Volumeshop Core Objects and methods

Class	methods	native binding
Box	GetCenter	yes
	GetTranslated	yes
	GetTransformed	yes
	GetExtent	yes
	GetMinimum	no
	GetMaximum	no
Canvas	bind	yes

	release	yes
Container		-
Image<uchar,4>	GetWidth	no
	GetHeight	no
	GetIntegral	no
ImageResource		-
ImageTexture2D	bind	yes
	release	yes
	GetOffset	yes
	GetScale	yes
	GetSize	yes
Matrix	GetInverse	yes
	GetRotation	yes
	GetRotationMatrix	yes
	GetScale	yes
	GetScaleMatrix	yes
	GetTranslation	yes
	scale	yes
	translate	yes
	transpose	yes
	$_m ul$	no
	Get	no
	print	no
Object	GetObjectTypeName	yes
	GetObjectName	yes
	SetObjectName	yes
	HasAttribute	yes
Observer		-
ModifiedObserver		-
Observable	addObserver	yes
Plane	GetTransformed	yes
	$_u nm$	yes
	GetClosestPoint	yes
	IsInside	yes
	GetCenter	yes
	GetNormal	no
	GetDistance	no
	constructor	no
	transform	no
	transform	no
PluginContainer		-
PropertyContainer	GetProperty	no
Resource		-

ResourceTexture		-
Shader	GetOption	yes
	GetUniformLocation	yes
	bind	yes
	release	yes
	load	no
	loadByName	no
	SetOption	no
StructuredGridResource	GetElementComponents	yes
	GetBounds	yes
Timer	getFloat	yes
	getDouble	yes
	start	yes
	isRunning	yes
Variant	require	yes
	TypeString	yes
	TypeArray	yes
	TypeMap	yes
	TypeOption	yes
	TypeBoolean	yes
	TypeInteger	yes
	TypeFloat	yes
	TypeVector	yes
	TypeQuaternion	yes
	TypeBox	yes
	TypeMatrix	yes
	TypeColor	yes
	TypeHandle	yes
	TypeFileName	yes
Vector	GetX	yes
	GetY	yes
	GetZ	yes
	GetX	yes
	GetY	yes
	GetZ	yes
	GetCross	yes
	GetDot	yes
	GetInverse	yes
	GetNormalized	yes
	GetMinimum	yes
	GetMaximum	yes
	$_{a}dd$	no
	$_{s}ub$	no

	$_{u}nm$	no
	$_{d}iv$	no
	$_{m}ul$	no
	constructor	no
	Get	no
Volume<[uchar|float], [1|2|3|4]>	GetWidth	yes
	GetHeight	yes
	GetDepth	yes
VolumeHierarchicalIterator<[uchar|float], [1|2|3|4]>	GetPositionX	yes
	GetPositionY	yes
	GetPositionZ	yes
	GetWidth	yes
	GetHeight	yes
	GetDepth	yes
	IsLeaf	yes
	GetMinimum	no
	GetMaximum	no
VolumeResource		-
VolumeTexture3D	GetSize	yes
	GetTransformation	yes
	bind	yes
	release	yes
Voxel<float, [1,2,3,4]>	Get	no

Native binding indicates that the method is bound directly to the original classes method without any modifications. Classes with a - sign in the 3rd column do not have any methods bound specifically. They might, however have inherited methods from bound parent classes.

There are also some helper methods available in squirrel:

method	description
GetResourceVolumeTexture3D	gets a 3D volume texture from a resource
GetResourceImageTexture2D	gets a 2D image texture from a resource
sqrtf	the standard square root function
rand	the standard random function
updatePlugin	calls the *updatePlugin* method on the scriptable renderer plugin
updatePlugin	calls the *renderBounds* method on the scriptable renderer plugin
RequirePropertyType	requires the variant type of a property
GetPropertyVariant	proxy to GetPlugin().GetProperty()
SetPropertyDescription	proxy to GetPlugin().SetPropertyDescription()
AddPropertyObserver	Proxy to GetPlugin().GetProperty().addObserver

Bibliography

[Bru13] Stefan Bruckner. Volumeshop: An interactive system for direct volume illustration. In *In Proceedings of the IEEE Conference on Visualization (Minneapolis, USA, Oct 2005), VIS '05, IEEE Computer Society*, pages 671–678, 11 2013.

[B.V] Ajax.org B.V. Ace editor - http://ace.c9.io/, last visited 11-2013.

[Dem] Alberto Demichelis. The squirrel programming language - http://www.squirrel-lang.org/, last visited 11-2013.

[Des] Thibaut Despoulain. Bkcore shader editor. http://shdr.bkcore.com/.

[DGE04] J. Diepstraten, M. Görke, and T. Ertl. Remote line rendering for mobile devices. In *Proceedings of Computer Graphics International (Crete, Greece)*, pages 454–461, June 2004.

[Gin] Stéphane Ginier. Sculptgl - http://stephaneginier.com/sculptgl/, last visited 02-2014.

[Gro] The Khronos Group. Opengl es - http://www.khronos.org/opengles/, last visited 11-2013.

[Haf] Brandon Haffen. The sqrat binding library - http://scrat.sourceforge.net/, last visited 11-2013.

[Inca] SketchFab Inc. Sketchfab - https://sketchfab.com/, last visited 02-07-2014.

[Incb] Verold Inc. http://verold.com, last visited 11-2013.

[jF] The jQuery Foundation. jquery stateful plugins. http://learn.jquery.com/plugins/stateful-plugins-with-widget-factory/, last visited 02-07-2014.

[mev] Mevislab - http://www.mevislab.de/, last visited 02-07-2014.

[MSRMH09] J Meyer-Spradow, T Ropinski, J Mensmann, and K Hinrichs. Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13, 2009.

[OT] Mark Otto and Jacob Thornton. Bootstrap frontend development framework - http://getbootstrap.com/2.3.2/index.html, last visited 11-2013.

[Pro14] The JOGL Project. Jogl opengl bindings for java. http://jogamp.org/jogl/www/, last visited 02-07-2014.

[RSFWH98] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[SBP+08] K. Sihan, Charl P. Botha, Frits H. Post, Sebastiaan de Winter, E Regar, R Hamers, and N Bruining. A novel approach to quantitative analysis of intravascular optical coherence tomography imaging. In *Proceedings of Computers in Cardiology*, http://www.cg.its.tudelft.nl/Projects/DeVIDE, 2008.

[SL07] Ol. Marquardt S. Lietsch. A cuda-supported approach to remote rendering. In *Proceedings of the 3rd International Conference on Advances in Visual Computing, ISVC '07*, 2007.

[SML07] W. Schroeder, K. Martin, and B. Lorensen. The visualization toolkit, third edition. *Kitware Inc.*, 2007.

[SNC12] S. Shi, K. Nahrstedt, and R. Campbell. A real-time remote rendering system for interactive mobile graphics. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 8(3s), 2012.

[SW05] H.-C. D. Stalling and M. Westerhoff. Amira - a highly interactive system for visual data analysis. In C. Johnson and C. Hansen, editors, *The Visualization Handbook*, pages 749–767. Elsevier, 2005.

[vir] Virtualgl - http://www.virtualgl.org/, last visited 02-07-2014.

[VWE05] Joachim E. Vollrath, Daniel Weiskopf, and Thomas Ertl. A generic software framework for the gpu volume rendering pipeline. In *In Proc. Vision, Modeling, and Visualization*, pages 391–398, 2005.

[web] Webgl playground - http://webglplayground.net/, last visited 02-07-2014.

List of Figures