UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

VJ-Framework Including a Module for Deskewing Planar Surfaces

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Anton Baumesberger

Betreuer: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Betreuer: Dr. Ing. Marcin Grzegorzek
(Institut für Informatik, AG Informationssysteme und Semantic Web)

Koblenz, im Mai 2009

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Seit 2005 beschäftige ich mich im Rahmen der Künstlergruppe “Farbraum” mit visuellen Installationen und live Video Performaces auf kulturellen Events. Dafür haben wir einzelne Video-Performance Applikationen entwickelt, die die Probleme einzelner Projekte lösen. Was uns bisher noch nicht gelang ist a) eine modulare Softwarearchitektur zu entwickeln und b) ein Werkzeug zur Entzerrung ebener Flächen, die nicht rechtwinklig projiziert werden, zu erstellen (unter der Annahme, dass Projektoren verwendet werden). Diese Arbeit beschreibt die Lösung des ersten Problems durch die Entwicklung eines modularen Frameworks und des zweiten Problems durch die Implementation eines benutzerfreundlichen Moduls zur Entzerrung von ebenen Flächen. Die Entzerrung findet komplett manuell statt, indem der Benutzer die Koordinaten der Flächeneckpunkte durch das Ziehen der Punkte mit der Maus verändert. Dabei werden die x- und y-Werte der Eckpunkte verändert, der z-Wert bleibt konstant. Während auf diese Weise die 3D-Interaktion mittels eines 2D-Eingabegeräts verhindert wird, führt die ausschließlich zweidimensionale Transofrmation der Flächen zu unerwünschten Textur-Mapping Artifakten, die durch das Triangulierungs-basierte Rendern von Grafikkarten entstehen. Um diese Artifakte zu vermeiden, wird ein Verfahren names “adaptive Subdivision” vorgestellt, das die entsandenen Rendering-Fehler korrigiert.

Abstract

As a member of the visual artist group “Farbraum” I have been creating visual installations and live video performances at cultural events since 2005. Until now, we have been developing single video-performance applications that mostly met the demands of a certain project. What we did not achieve so far is a) a modular software design and, b) a feature for deskewing planar surfaces that are projected at a non perpendicular angle (assuming the use of visual projectors). This paper deals with solving the first problem by desiging a modular framework and the second problem by implementing an user-friendly module for deskewing planar surfaces. The deskewing process is completely manual, letting the user edit the coordinates of the surfaces’ cornerpoints by dragging the points with the mouse. For this, the cornerpoints’ x- and y-values are manipulated and the z-value is left constant. While in this way, the 3D-interaction with a 2D-interface can be avoided, the exclusivley two-dimensional transformation of planar surfaces introduces undesired texture-mapping artifacts produced by the triangulation-based rendering of graphic-cards. In order to avoid these artifacts, a selected method called “adaptive subdivision” is presented that corrects the introduced rendering errors.

Contents

1 Introduction

1.1 Definition of “VJ”, “Framework” and “Module”

At first, it is important to clarify the expressions “VJ” , “Framework” and “Module”.

“A **VJ** is a performance artist who creates moving visual art (often video) on large displays or screens, often at events such as concerts, nightclubs and music festivals, and usually in conjunction with other performance art. This results in a live, multimedia performance that can include music, actors or dancers as well as live and pre-recorded video.” [Wik09c]

A “modular Framework” might seem to have a quite distinct meaning, at least from the software-develper’s point of view, but it is not the case. It has to be mentioned that at the one hand, “module” is a rather abstract term leading to controversal discussions in software engineering and on the other hand, the semantics of “framework” mostly depends on the point of view from which this expression is used (e.g. the framework’s developer or the developer that uses the framework or the end user who runs an application that uses the framework). We will not deal with that controversy at this point but define these two expressions from “Farbraum”’s point of view instead. Furthermore, when the term “framework” is being mentioned in the text, it will always be clear from which point of view the term is looked upon at that moment. Before taking the following definition of “framework” into consideration, we assume that acutally “software framework” is meant.

“A software **framework** is a re-usable design for a software system (or subsystem). A software framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project. Various parts of the framework may be exposed through an API.” [Wik09a]

The term “**Module**” shall be understood as “an approach that subdivides a system into smaller parts (modules) that can be independently created”. [Wik09b]

Summarizing the title and the mentioned definitions, the intention of this work is to develop a software system capable of gluing together smaller, independent parts (modules) that can be used to control visual art installations and to develop a module for deskewing planar surfaces which are projected at a non perpendicular angle.

1.2 Related Work (Deskewing)

Marcel Lancelle et. al. have adressed the deskewing problem [LOU^{+}06]. They present a semi-autmated procedure without additional hardware that can be integrated into OpenGL’s rendering pipeline. Their algorithm requires a user input of at least four reference points which are the target-plane’s cornerpoints. Then, a projection from the source-points to the target points is computed by the optimization of a correction matrix C that is the product of the translation-, scaling-, shearing-, rotation- and trapezoid deformation-matrices.

In Jens Barth’s diploma thesis [Bar09] the task of manual projector rectification is adressed among other things. He presents a constraint-based procedure to establish homographies between the distorted and the corrected image-planes by letting the user define the cornerpoints of the target plane. Moreover, he introduces a technique for subdividing the image-plane where in the first step, the geometry is divided and in the second step, the texture-coordinates are adjusted to compensate for undesired stretching and/or compressing effects.

Florian Limburg’s diploma thesis [Lim07] also adresses the problem of distorted projections amongst others. He suggests to solve the problem by modifying the Scheimpflug principle to meet his needs. Normally, the Schimpflug principle is used to correctly adjust a camera’s depth of focus when the plane of focus is not coplanar with the lens and image planes. Although the plane of focus is irrelevant, the Schimpflug principle provides the linear equations to rectify a distorted projection. In his approach, the author also lets the user drag the cornerpoints with the mouse.

1.3 Overview

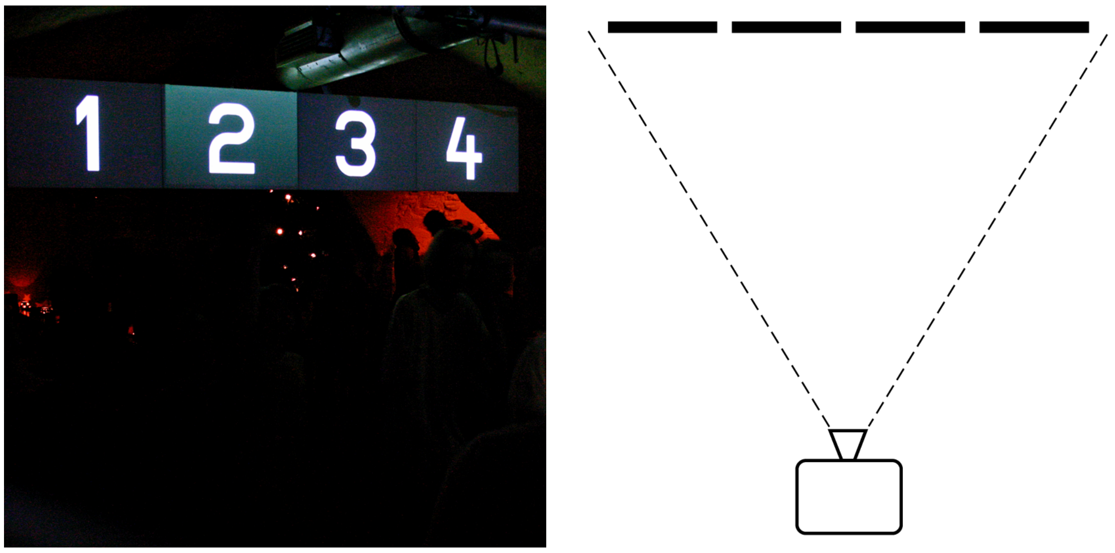
This paper is subdivided into the following topics: In chapter 2, the motivation for this work is explained outlining the target projection setup and a visual programming aplication for creating animations. Chapter 3 describes the rendering architecture of the target platform before it defines the requirements for the framework followed by the description of the framework's architecture. The deskewing module is outlined in chapter 4: first, the requirements for the deskewing module are deriviated from the desired deskewing result. After that, the used deskewing procedure is presented and the module's development is outlined from the developer's and the user's perspective. Chapter 5 gathers the results of the framework and of the deskewing process. Finally, section 6 draws a conclusion by comparing the achieved results with selected current state of the art software, provides a short summary and finally gives an outlook to possible enhancements and future work.

2 Motivation

This work and the intention behind it is based on the experiences made with “Farbraum”. Installing visual performances with projectors, we have always been using software to control the projections. For those projects we have developed different applications but it was often the case that a single application was the solution of a single problem for a certain project. Thus, several features have piled up distributed over different software-projects that are hard to maintain. In order to gain a more common ground for those features, developing a framework seems adequate providing a standartised and extensble API. Basing on that framework, new features would be developed as new modules that can be arranged independently of each other within one software-system. To understand the requirements for the software from “Farbraum”’s point of view, it is essential to inspect the target projection setup.

2.1 Projection Setup

An ideal setup is shown in 1 where figure 1(a) depitcs a foto of an exemplary setup and figure 1(b) its projection scheme (top view). One projector illuminates four different planes in this example. The projector is in rectangular position to the projection-plane, thus there is no problem from the designer’s point of view.



(a) (b)

Figure 1: Ideal Projection Setup

Asuming an oblique angle between the projector and the projection plane(s), distortion effects emerge as you can see in figure 2 where 2(a) shows the projection scheme from top view and 2(b) depicts the projection result from side view figuratively. The trapezoid in 2(b) outlines the projection image while the four rectangles represent the four target planes of the setup. Obviously, if the projector’s image is a trapezoid instead of a rectangle, the image on the planes is also going to be a trapezoid and not rectangular.

In addition to horizontal distortion, vertical distortion is another possible drawback depending on the vertical location of the beamer (figure 3(a)). Since it is unlikely to have the projector setup exactly at a point (horizontally and vertically) where no distortion effects arise, realistically the projection image is shaped like figure 3(b) shows. More generally spoken, the quadrilateral of the projection image has four different angles. The projection distortions have to be corrected

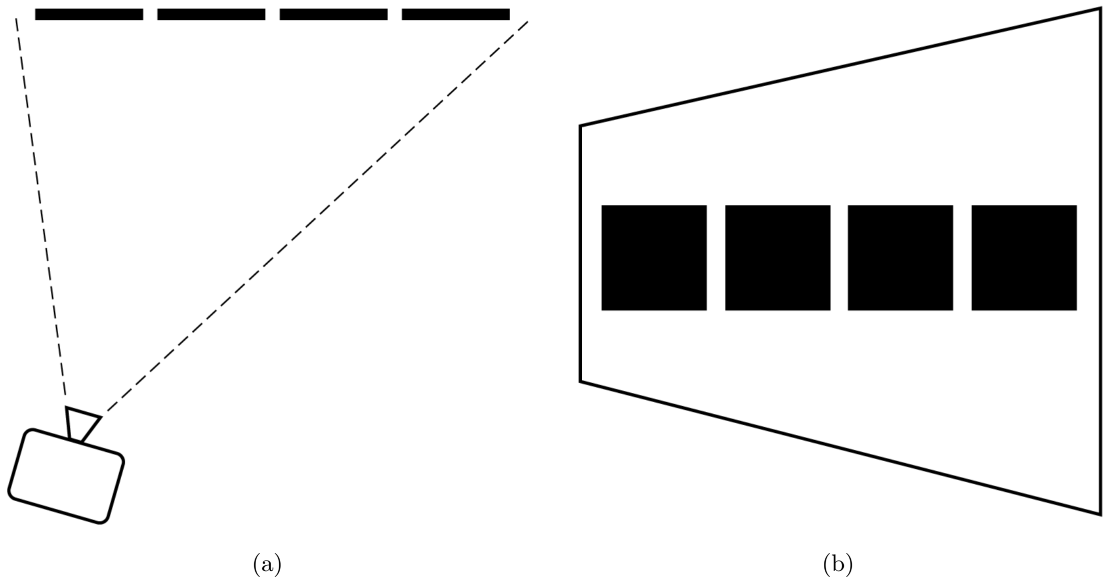


Figure 2: Projection Setup with Horizontal Distortion

in order to achieve the desired projection result: the four planes of the examplary setup have to stay rectangular independent of the projector's position.

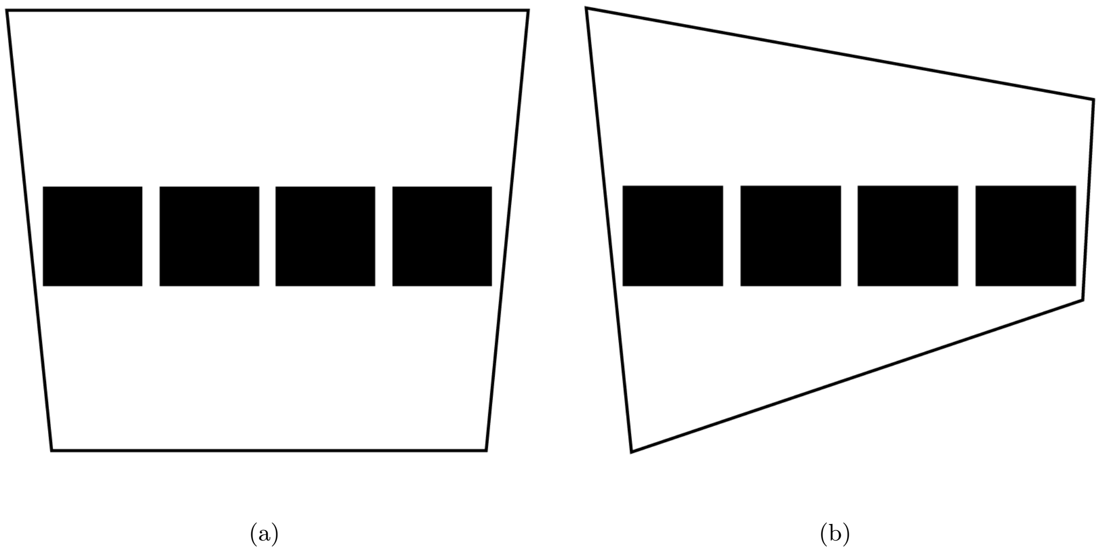


Figure 3: Projection Setup with Vertical and Horizontal Distortion

Most modern projectors offer a keystone correction along the vertical axis and more sophisticated (and more expensive) projectors additionally provide keystone-correction along the horizontal axis. These features are not sufficient though because the distortion is different for each of the four planes and cannot be corrected by a global keystone-setting. For that reason we need a feature for correcting the distortion of each plane locally instead of correcting the projector-image globally. This correction should be appliable for vertical distortion and for horizontal distortion as well as for the combination of both distortions allowing for variable setups.

Figure 4 illustrates such a possible setup, where the oblique angle does not emerge from the position of the projector (which is centered with respect to the planes) but from the rotation of the planes around their y- and x-axes respectively. However, the problem to be solved remains the same since the projection angle is not perpendicular.

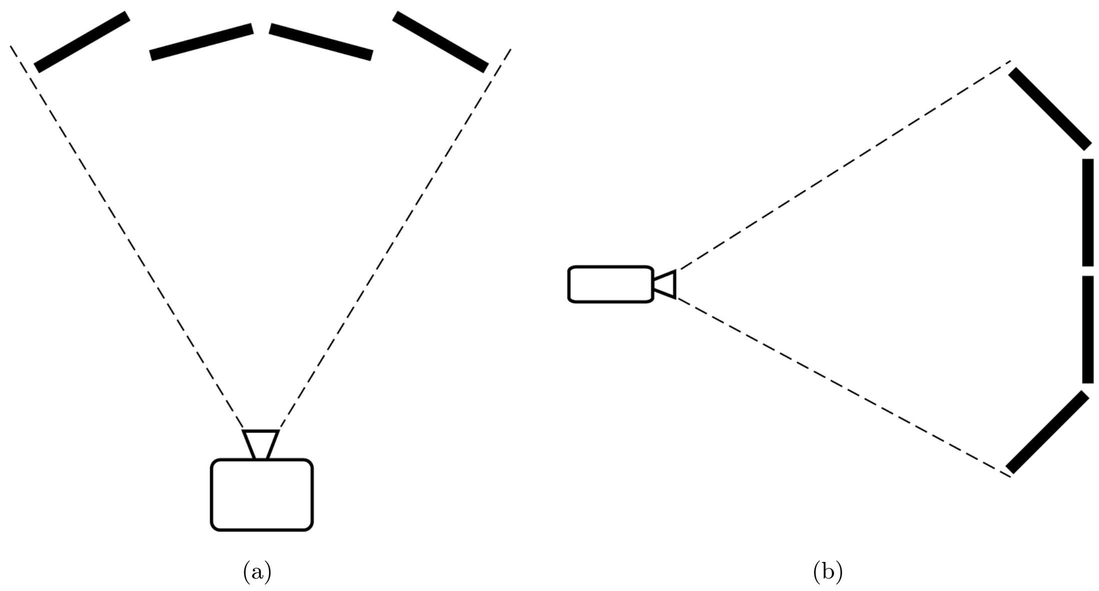


Figure 4: Possible Projection Setup with Vertical 4(a) and Horizontal 4(b) Distortion

While this section has described the installation conditions of the projector and the target plane(s), the next section deals with the content that is being projected on the planes (the animations).

2.2 Quartz Composer: Visual Programming

“Farbraum”’s preferred operating system is Apple’s OSX. Since version 10.4 of the platform, OSX offers an application that provides a visual programming paradigm. The application is called “Quartz Composer” and is presented in this section as a possibility to create animations (using Quartz Composer of the OSX version 10.5 “Leopard” [App09f]). “Farbraum” uses Quartz Composer for creating animations that are the content for the projection planes.

Apple’s webiste describes Quartz Composer as a *“real-time visual programming environment”* [App09f]. Looking at figure 5 you can see the visual editor on the left side. In the editor window, you can see several nodes and connections between them, alltogether this is called a composition. On the right side, a preview window shows the output of the composition. A composition is the visual programming source that is made up from nodes, called patches, and connections between patches. A connection is the link between the output port of the source patch and the input port of the target patch. Input ports are located on the left side of a patch and accordingly, output ports are at the right side of a patch, so that a composition can be read from left to right. Three types of patches are available: generators, filters and renderers and additionally, macro-patches provide nesting funcionality for hierarchical structures. Compositions are rendered at realtime without the need to compile or to preprocess the source. This capability allows for an effective workflow, creating patch combinations whose changes are instantly visible once a new connection between two patches has been established.

Figure 5 illustrates a simple example composition connecting a “Video Input” patch (a gen-

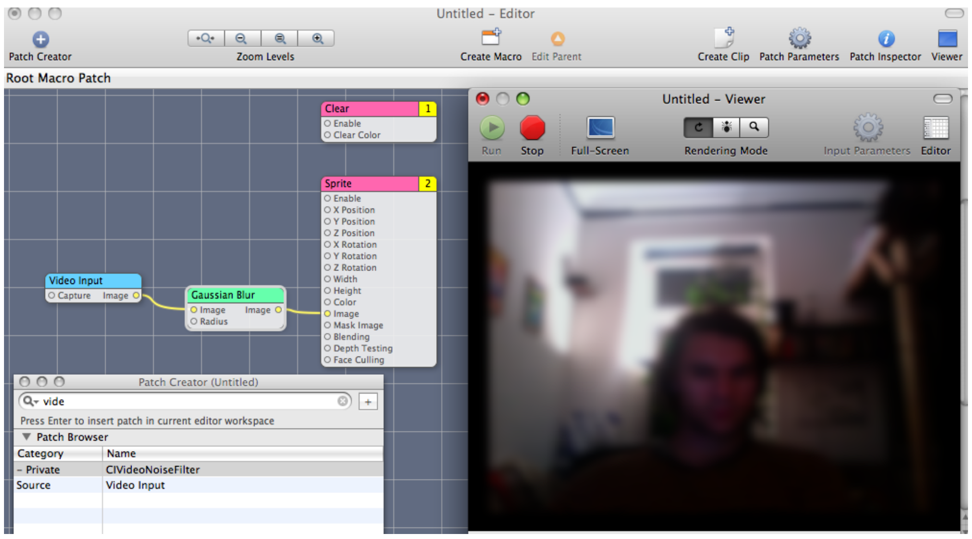


Figure 5: Screenshot of Quartz Composer

erator) to a "Gaussian Blur" filter patch that is wired to a "Sprite" renderer patch. The "Sprite" patch is at index 2 (the yellow box in the topright corner of the patch) while the "Clear" patch above the "Sprite" patch is at index 1. This means that the "Clear" patch is rendered first, before the "Sprite" patch is drawn in the rendering cycle. You can see the output of the composition in the viewer window on the right side. Besides, as figure 5 shows in the left bottom corner, the "Patch Creator" window provides access to the patch library hosting all available patches (e.g. math-patches, random-generators, image-filters, rendering-environments etc.).

Apart from the visual programming paradigm, Quartz Composer features publishing input ports of patches. When an input port is published, a user-interface element is created for that port automatically and the type of the user-interface control element depends on the type of the port variable. For example, boolean values are mapped to checkboxes while numbers can be manipulated by control dials and so forth. The purpose of publishing input ports is that selected parameters of a composition can be controlled without the need to edit the composition. The user interface for published ports is located in the viewer's "Input Parameters" panel. What's more important is that the Quartz Composer API offers a parameter-widget for published ports. This means that we can use the automatically generated user interface for composition parameters in the desired application. Furthermore, Quartz Composer provides an API for the rendering of compositions in an offscreen buffer, making them useable as textures. Although it does not concern this work, it is worth mentioning that an API for programming own patches is also available.

In this section the Quartz Composer application has been described shortly. What makes this application interesting is that it allows for creating animations very quickly proposing a visual programming paradigm. Moreover, the animations can be seperated into a core (the composition) that remains unchanged and a customizable set of parameters (the published input ports) that can be manipulated while performing. In combination with the possibility to render compositions as textures, those two features have convinced "Farbraum" to use Quartz Composer as the editing tool for the animations that are projected on the planes of their setups.

3 Modular VJ-Framework

This chapter deals with the architecture of the framework. However, before turning to the design of the framework, a short overview of the OSX system atchitecture gives an idea how the target platform is designed and which parts of the system the framwork will be connected with.

3.1 OSX System Architecture

Figure 6 draws the OSX system architecture with the "Darwin" component being the *"opensource UNIX-based foundation of Mac OSX"* [App09d] or to put it in a single word, the kernel. Reading the figure from bottom to top, each layer makes use of the layer beneath it. Following that principle, the "Graphics and Media" layer is built atop of the kernel layer and holds the first component of interest for the framework: the "OpenGL" component. It provides access to the API of the graphics hardware. The other component that is going to be used is nested inside the "Application Frameworks" layer: "Cocoa" is *"an object-oriented application environment designed specifically for developing Mac OSX native applications"* including *"a full-featured set of classes designed to create robust and poerful Mac OSX applications"*. [App09d]

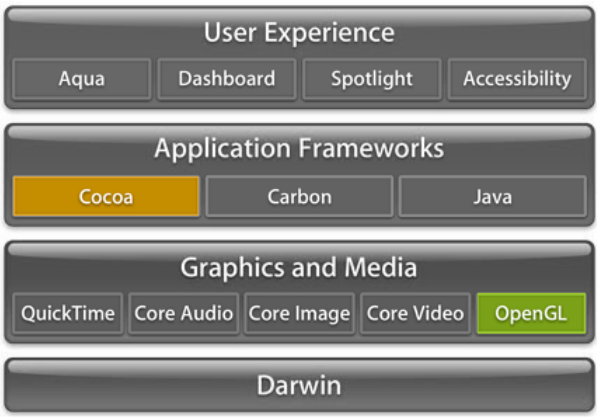


Figure 6: Mac OSX System Architecture [App09d]

The current version of OSX, version 10.5 (called "Leopard"), provides another interesting new feature for OpenGL-programming. As the webiste states, OpenGL performance is enhanced by *"offloading processing onto a seperate thread which can run on a different CPU core."* [App09c] This means that the application's run loop runs on a different CPU core, if a multicore system is available of course. The advantage is that the user-interface response is independent of the graphics-rendering effort. Additionaly, the background-thread that is responsible for the OpenGL processing *"can do everything possible to keep the GPU rendering pipeline efficiently filled"*. [App09c]

Obviously, one has to be aware of the fact that the used statements and descriptions are based on the information of the platform's manufactorer only since possible benchmarks or verification methods lie beyond the bounds of this work.

3.2 Requirements for the Framework

The requirements for the framework are based on the experiences gathered so far with the applications developed by "Farbraum". Apart from the demand of a mechanism to integrate modules, the common ground for all applications is an OpenGL-view that lies inside a frameless window. Moreover, registering modules for mouse-events is interesting. The exact requirements for the framework are listed in figure 7.

- a frameless window that can be positioned and resized arbitrarly including an access-mechanism for modules if they need to access the window
- an OpenGL view that is accessible for modules if they are interested in accessing it
- a machanism to register modules for mouseDown, mouseDragged and mouseUp events in the OpenGL-view if the modules want to register for those events
- a mechanism to plug-in and -out modules at runtime

Figure 7: Framework Requirements List

3.3 Framework Architecture

During the design process, the first idea was to develop a framework in the sense of an OSX-framework. An OSX framework is *"a hierarchical directory that encapsulates shared resources, such as a dynamic shared library, nib files, image files, localized strings, header files, and reference documentation in a single package. Multiple applications can use all of these resources simultaneously. The system loads them into memory as needed and shares the one copy of the resource among all applications whenever possible."* [App09g] However, my considerations led me to the decision that it would be inappropriate to create an OSX framework. The reason for this is that *"Frameworks are most effective when their code is shared by multiple applications"* [App09a]. Following that principle, frameworks are useful to *"separate out generic, reusable code from your application-specific code"*. [App09a] Since the idea was to create only one application featuring an arbitrary number of modules, what we rather want is a modularized application structure providing the possibility to plug-in and out modules at runtime. From this point of view, a plugin-architecture is more appropriate as opposed to an OSX framework.

With the decision for the architecture made, the framework can be laid out as the class diagram in figure 8 illustrates. From now on, the framework will be referred to as "Farbraum Studio" or "FRStudio".

The main class of Farbraum Studio is the **FRStudioController**. It composes the frameless window (FRWindow), the OpenGL-view (FRopenGLView) and the plugin-manager (FRPluginManager). Furthermore, it serves as the controller for the window-GUI (see figure 9(a)), receiving an event when the "Set Window" button is pressed. This causes the window to change its position and size to the values entered into the respective text-fileds (see also figure 9(a)). The window's position can also be changed by holding the Shift-Key and dragging the window with the mouse. The **FRWindow** class extends the standart Window class of Cocoa (NSWindow) with the frameless feature and moreover implements the mentioned dragging-mechanism. The **FROpenGLView**-class inhertis from Cocoa's NSOpenGLView-class configuring it to respond to the very first mouse-click inside the view. Additionally, the mouseUp, mouseDragged and mouseDown events are propagated using Cococa's notification-mechanism (NSNotificationCenter). Plugins can obtain those mouse-events in this way, how that works in detail is explained at a later point. The initialization of the OpenGL-view is up to the module so that it can configure OpenGL according to its needs.

Plugging in modules at runtime requires loading code into the application at runtime. Rainer Brockerhoff has described a possible plugin-architecture using Cocoa in his paper "Plugged-

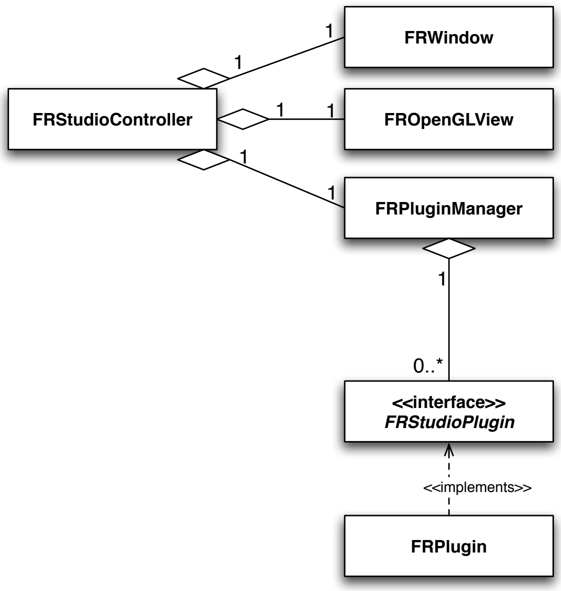
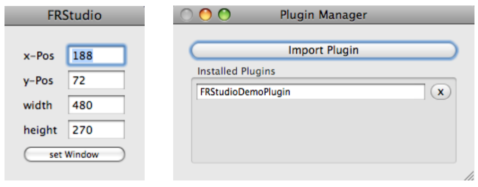


Figure 8: Class Diagram of the Framework Architecture (omitting attributes and operations)


FRStudio

x-Pos 188
y-Pos 72
width 480
height 270

Installed Plugins

FRStudioDemoPlugin

(a) (b)

Figure 9: User Interface Elements: Frameless Window Control 9(a) and Plugin Manager Window 9(b)

in Cocoa" [Bro02] for the "Machack 2002 papers competition". Moreover, Apple's developer documentation introduces the task of loading code at runtime and designing a plugin-architecture in the article "Introduction to Dynamically Loading Code" [App09b]. Taking into consideration the information of these two guides, the laid out plugin-architecture consists of two parts: the **FRPluginManager** class is responsible for handling modules in Farbraum Studio on the one hand and on the other hand the **FRStudioPlugin-interface** defines the operation signatures for plugins.

Figure 9(b) shows a screenshot of the FRPluginManager user-interface. The "Import Plugin" button triggers a Fileopen-panel in which the user can chose a FRStudio-plugin for installation. Following the Cocoa standart, FRStudio-plugins are files ending on ".bundle". If the chosen plugin has been successfully installed, a textfield with the name of the plugin appears in the "Installed Plugins" box as you can see in figure 9(b). Otherwise, if the installation has not succeeded, an error-dialog apperas and names the reason why it could not be installed (e.g. the chosen file is not a valid FRStudio-Plugin). Installed plugins can be deinstalled by pressing the "x"-button on the right side to the plugin's textfield-entry.

Now that the user-interface is described, the following section provides information about the most important methods of the FRPluginManager class to explain its functionality from the developer's point of view.

– (**void**)setTheOpenGLView:(NSOpenGLView∗)view
This method sets the reference to the OpenGLView. FRStudioController calls that method with a reference to the FROpenGLView as argument. Thus, the FRPluginManager has a reference to the FROpenGLView and can pass this reference to the plugins that are interested in accessing the OpenGLView.

– (**void**)setTheWindow:(NSWindow∗)window
This method sets the reference to the window. FRStudioController calls that method with a FRWindow-reference as argument. In this way, the FRPluginManager can pass the window-reference to those plugins that are interested in accessing the window.

– (**void**)updateOpenGLViewForPlugins
This method gets called whenever the window-size changes. FROpenGLView fills the window entirely so when the window-size has changed, the FROpenGLView changes accordingly. Plugins that have a reference to the FROpenGLView are notfied whenever the OpenGLView's size changes.

– (**void**)loadPlugin:(NSString∗)path
This method is called as soon as the user has chosen a file from the Fileopen-dialog on plugin-import. The parameter "path" holds the filepath to the chosen file which is parsed. If it is a ".bundle" file, it gets validated. The validation checks whether the principal class of the plugin exists and whether it is FRStudioPlugin-interface conform. Moreover, a duplicate-check is performed testing whether the plugin is already installed. If it is not installed yet, the last validation-check examines whether the plugin has already been installed in the same FRStudio session. If so, the user is shown a warning, nevertheless, the plugin will be installed. The reason for this is that once Cocoa has loaded a class, this class cannot be overloaded. Thus, if the plugin's principal class was modified between its deinstallation and the recent installation, the modified version of the plugin's principal class cannot be instantiated but the older version of the plugin's principal class is instantiated (the class that was loaded at the first installation). If the plugin was validated successfully, the installation-routine is called.

– (**void**) installPlugin :(Class<FRStudioPlugin>)pluginClass andPluginName:(NSString∗)pluginName
This is the installation routine. It is responsible for creating the plugin-instance. Moreover, the plugin is examined whether it is interested in mouse-events, access to the FRWindow or access to the FROpenGLView. After the instantiation has succeeded a new entry for the installed plugin is created in the FRPluginManager's user-interface (see figure 9(b)).

– (IBAction)removePlugin:(**id**)sender
This method gets called when the "x"-button in the FRPluginManager's user-interface is pressed (see figure 9(b)). It is responsible for removing it from the application and for calling the plugin's exiting-mechanism which is responsible for cleaninig up all allocated resources (this will be explaned in detail in the context of the FRStudioPlugin-interface).

Now that the FRPluginManager has been characterized, the focus shifts to the last piece of FRStudio to describe: the FRStudioPlugin-interface. Before that, it is important to know that interfaces are called "protocols" in Cocoa and in addition to that, two types of protocols exist : **formal protocols** and **informal protocols** (see [App09e]). Formal protocols guarantee that all defined methods must be implemented by the consumer-class. A compiler warning is given and a runtime error occurs if an unimplemented method is called. An informal protocol lists optional methods and no compiler-warning is generated if a method is missing in the consumer-class. Obviously, a runtime-error occurs if an unimplemented method is called. The keywords "required" and "optional" are the modifiers for formal and informal protocols respectively. If a protocol has required and optional method-definitions, it is categorized as an informal protocol. Since the requirements for the framework list optional features (e.g. access to the FRWindow is provided only if the plugin is interested in accessing it), it is appropriate to define optional method-signatures for those features. Thus, the FRStudioPlugin-protocol is informal, consisting of required and optional method-signatures. Listing 1 shows the FRStudioPlugin protocol.

Listing 1: The FRStudioPlugin Interface

```
#import <Cocoa/Cocoa.h>

@protocol FRStudioPlugin

// the required methods
@required

+ (BOOL) initializeClass:(NSBundle*)theBundle;

+ (void) terminateClass;

+ (NSObject<FRStudioPlugin>*) createInstance;

- (void) exit;

- (BOOL) wantsMouseEvents;

- (BOOL) wantsTheWindow;

- (BOOL) wantsTheOpenGLView;


// the optional methods
@optional

- (void) mouseDown:(NSNotification*)inNotification;

- (void) mouseUp:(NSNotification*)inNotification;

- (void) mouseDragged:(NSNotification*)inNotification;

- (void) setTheWindow:(NSWindow*)window;

- (void) setOpenGLView:(NSOpenGLView*)view;

- (void) updateOpenGLView;

@end
```

+ (BOOL)initializeClass:(NSBundle*)theBundle
is a class method that should do all global plugin-initialization (such as loading preferences) and only return TRUE if the initialization succeeds.

+ (**void**)terminateClass is called when the plugin has been deinstalled and should release the resources that were loaded by the initializeClass-method.

+ (NSObject<FRStudioPlugin>∗)createInstance
is the plugin's static factory responsible for allocating the instance of the plugin. The created instance has to be returned by this method. This is the place for alocating and initialising the subcomponents of the plugin. This method is called by the FRPluginManager to instantiate the plugin's principal class.

− (**void**) exit
is called when the plugin gets deinstalled. It should release all allocated subcomponents.

− (BOOL)wantsMouseEvents
has to return TRUE if the plugin wants to register for mouseDown, mouseDragged and mouseUp events, otherwise it has to return FALSE. If it returns TRUE, the plugin-class is examined whether it provides the mouseDown, mouseDragged and mouseUp methods and only if this is the case, the plugin gets registered for receiving those events.

− (BOOL)wantsTheWindow
has to return TRUE if the plugin needs a reference to the FRWindow, otherwise, the plugin must return FALSE.

− (BOOL)wantsTheOpenGLView
has to return TRUE if the plugin is interested in a reference to the OpenGLView. If so, the plugin has to implement the setOpenGLView and the updateOpenGLView methods and only if this is the case, a reference to the FROpenGLView is passed to the plugin.

− (**void**)mouseDown:(NSNotification∗)inNotification
is called on a mouseDown-event if the plugin was registered for mouse-events.

− (**void**)mouseDragged:(NSNotification∗)inNotification
is called on a mouseDreagged-event if the plugin was registered for mouse-events.

− (**void**)mouseUp:(NSNotification∗)inNotification
is called on a mouseUp-event if the plugin was registered for mouse-events.

− (**void**)setTheWindow:(NSWindow∗)window
is called if the plugin wants a reference to FRWindow and if so, sets the reference.

− (**void**)setOpenGLView:(NSOpenGLView∗)view
is called if the plugin wants a reference to the OpenGLView and if so, sets the reference.

− (**void**)updateOpenGLView
is called whenever the size of the FRWindow changes, affecting the change of the FROpenGLView's size. Through this method, plugins are notified of the resizing and can react to it.

This chapter has presented the requirements, the design and the implementation of FRStudio. The next chapter deals with the deskewing module which will be implemented as a FRStudio-Plugin.

4 Deskewing Module

The previous chapter defined the FRStudio framework and described its plugin-architecture which is why in the following, the deskewing module will be referred to as the "deskewing plugin". In order to describe the requirements for the deskewing plugin, figure 10 illustrates a distorted projector-image figuratively in the context of a realistic setup.

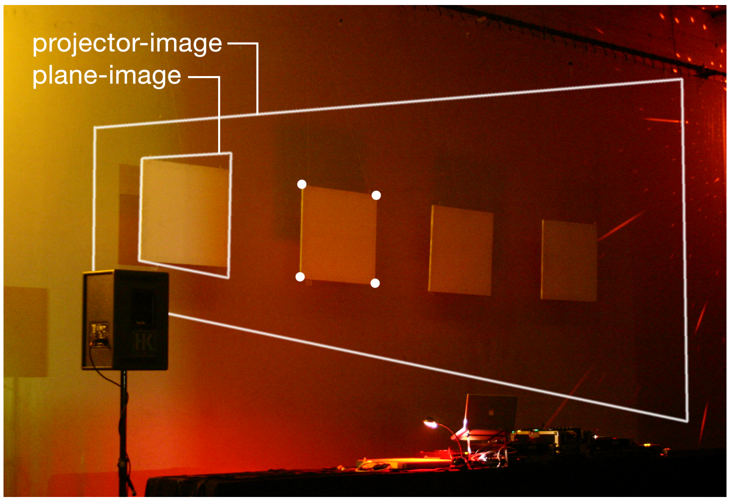


Figure 10: Distorted Projector- and Plane Image in Context of a Realistic Setup

The figure shows a foto of the four target planes (the canvas planes) and additionally, the outlined trapeziod "projector-image" represents a possible projector image. This projector image assumes that the projector is approximately at the same position as is the camera that shot the foto with respect to the target projection planes. The exact position is not important here, what the figure focuses on is the assumption that the angle between the projector and the target planes is not right. As you can see at the leftmost canvas-plane on the foto, the assumed projector-image would lead to the distorted "plane-image" if the plane-image generated by the controlling software was a rectangle. Now if you look at the second canvas-plane from left, the four white points represent the cornerpoints of the desired plane-image for that canvas-plane. So what we want is a mechanism to define different plane-images for the different canvas-planes. Ideally, the mechanism would allow for modification of the cornerpoints through a convenient mouse-dragging feature.

4.1 Requirements for the Deskewing Module

The mentioned feature for defining image-planes through cornerpoints and the dragging of the cornerpoints with the mouse are certainly the most important requirements for the deskewing plugin. In addition to that, several other features would be very interesting. All requirements for the deskewing plugin are listed in figure 11.

- defining image-planes through cornerpoints resulting in undistorted image-planes
- dragging the cornerpoints with the mouse
- moving image-planes
- removing image-planes
- dividing image-planes into subplanes
- generating plane-textures by rendering Quartz compositions offscreen
- a well designed user-interface for controlling the planes
- a user-interface element for controlling the published ports of the current Quartz composition (compare section 2.2)
- a feature for saving and loading image-plane configurations

Figure 11: Deskewing Plugin Requirements List

4.2 Deskewing Planar Surfaces

First, it is important to state the assumptions that we are using a) rectangular textures and b) OpenGL's orthographic projection model. Deskewing planar surfaces is a task that can be tackled by different approaches. A 3D-rotation could be applied to the projected image-planes assuming that the positioning of the projector at an oblique angle is a rotation around the y-axis (and probably the x-axis) with the center of the projector-image as the rotation center. However, this method would induce a transformation of the z-value(s) of the image plane(s). Since we are aiming at controlling the cornerpoints (the vertices) with the mouse, the immediate interaction is two-dimensional and takes place in the xy-layer only. For this reason, I assume that it would be appropriate tho manipulate only the x- and y-values of the image-planes. Under this assumption, the problem of sensibly handling the z-value-transformation can be ignored. Thus, we have to take care of a two-dimensional transformation. In order to achieve a rectangular plane-image on a trapezoid-like projector-image, a source square has to be projected as a trapezoid in order to appear rectangular on the distorted projector-image, as figure 12 illustrates. To achieve the

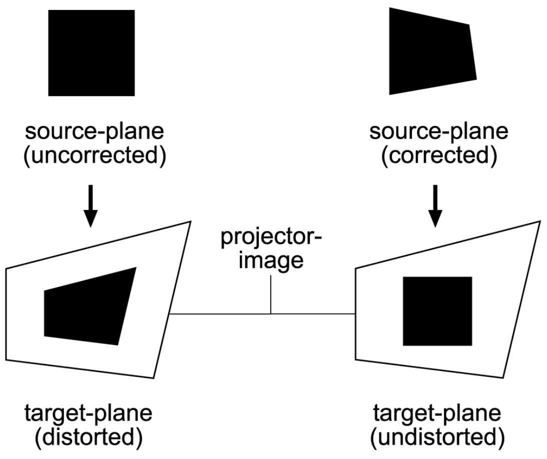


Figure 12: Distortion Correction

correction, we would then only need to drag the cornerpoints to their desired destination, that is, to adjust their x- and y-values. Unfortunately, the rendering process of graphic-cards introduces a problem by triangulating the geometry for the rasterization (the calculation of the pixel color values for the framebuffer). Manuel M. Olivera addressed this problem in his paper *"Correcting Texture Mapping Error Introduced by Graphics Hardware"* [Oli01] and moreover, suggested a solution for the problem. To understand the problem we must be aware of the fact that as long as the quadirlaterals meet the conditions of a parallelogram, the texture-mapping's rendering produces the expected results without artifacts. The reason for this is that when OpenGL triangulates the textured geometry for drawing, it is only parallelogram-like quadrilaterals that can be devided into two triangles of same area-size. As soon as the parallelogram-criteria are not met by the quadrilateral (i.e. the quadrilateral becomes a trapezoid), the areas of the two triangles are not equal in size anymore resulting in texturing artifacts as figure 13 depicts: on the left side you can see the texture image, in the middle the rendering result of OpenGL is shown and the desired rendering result is on the right side. What happens during the rendering-prcess and produces the middle image is that *"half of the texture is compressed in the smaller triangle, while the other half is stretched in the bigger one"* [Oli01].

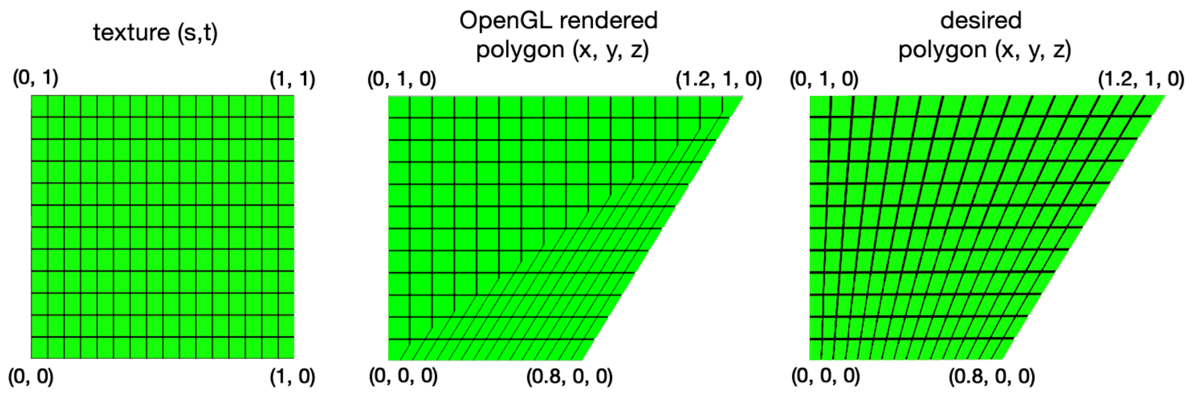


Figure 13: Triangulation Problem for non-parallelograms

Oliveira suggests to solve the problem by subdividing the quadrilateral and the texture-image into sub-quadrilaterals recursively, until both, the quadrilateral and the texture-image approximate a parallelogram closely enough. To measure the approximation intensity, Oliveira proposes to compare the opposite angles of the quadrilateral since in a parallelogram, the values of opposite angles are equal. For the comparison of the opposite angles the following assumption is made: let $V_1 = A - B, V_2 = C - B, V_3 = C - D$ and $V_4 = A - D$ be the four vectors that are depicted in figure 14 on the right side. Equal values of opposite angles in a parallelogram can be expressed by the dot product of the respective vectors: $V_1 \cdot V_2 = V_3 \cdot V_4$ and $V_1 \cdot V_4 = V_2 \cdot V_3$. If the difference between opposite angle-values is under a certain threshold, the quadrilateral approximates a parallelogram closely enough. These considerations sum up to the two formulas for the angle similarity calculation:

$$d_1 = abs(V_1 \cdot V_2 - V_3 \cdot V_4) \tag{1}$$

$$d_2 = abs(V_1 \cdot V_4 - V_2 \cdot V_3) \tag{2}$$

As Oliveria denotes, it is important to use different thresholds for the geometry and the texture because the texture space lies within the normalized coordinates $[0, 1]$ whereas *"the use*

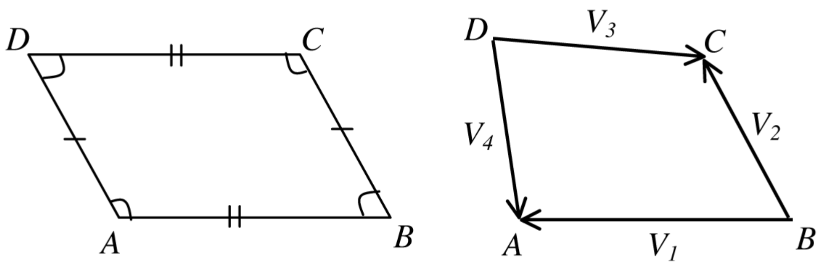


Figure 14: Calculating Parallelogram Approximation [Oli01]

of non-normalized vectors in the dot products favors the subdivision of larger polygons, which are most likely to exhibit noticeable artifacts” [Oli01]. Oliveira also provides empirical thresholds for the angle-differences of the geometry and of the texture respectively. If a quadrilateral needs to be subdivided, it is seperated into four sub-quadrilaterals by simply dividing the edges of the quadrilateral into two parts of equal length and then connecting the opposite points that emerge from this division. The intersection of these two connections generates the center point of the quadrilateral and finally leads to the four sub-quadrilaterals as you can see in figure 15.

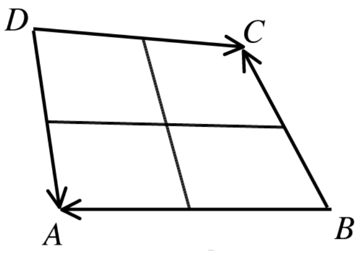


Figure 15: Subdividing a Quadrilateral

In addition to the two criteria for cancelling the recursion of the subdivision, Oliveira introduces a third condition: the size of the visible quadrilateral. In this way, unneccessary refinement can be avoided. Refinement becomes obsolete as soon as the (subdivided) quadrilateral covers only a few pixels on the output screen and no difference between the standart OpenGL-rendered and the subdivision-rendered result is visible to the human eye. For each frame the area-size that is covered by the quadrilateral is calculated in pixels. When calculating the area, the algorithm makes the simplifying assumption that the subdivision of a quadrilateral produces four regions of equal size. *“Although this assumption is often incorrect, it provides a good approximation and is used to avoid computing the projected area for sub-quadrilaterals.”* [Oli01] So, what Oliveira suggests is to correct the errors produced by the triangulation process of the graphics hardware by subdividing the quadrilaterals until they approximate a parallelogram closely enough or until the area-sze they cover is small enough. The algorithm is presented in pseudocode in listing 2. Additionally, the empiric threshold values for the cancellation-criteria of the recursion are introduced as suggested by Oliveira.

Listing 2: Adaptive Subdivision Algorithm

```
pixelThreshold          = 200;
polygonThreshold        = 0.001;
textureThreshold        = 0.0001;

function peformAdaptiveSubdivision(quadrilateral: quad)
      size= areaSize(quad);
      dotProductsPolygon = dotProducts(quad.polygon);
      dotProductsTexture = dotProducts(quad.texture);

      if ( (size < pixelThreshold) OR
            ( (dotProductsPolygon < polygonThreshold) AND
               (dotProductsTexture < textureThreshold) ) )
               draw(quad);
      else
         subquads = subdivide(quad);
         for each subq in subquads:
            peformAdaptiveSubdivision(subq);
         end for
      end if
end function
```

4.3 Module Development

4.3.1 Architecture

The adaptive subdivision method for deskewing planar surfaces is the algorithm that is part of the implementation of the “FRDeskewedMultiDisplayForCompositions”-plugin. Figure 16 outlines the class diagram of the plugin.

FRDeskewedMultiDisplayForCompositionsPlugin is the principal class of the plugin and implements the FRStudioPlugin-interface confirming to the protocol as required by FRStudio. Following the class diagram, the components are described in the following from top to bottom. The principal class composes the **FROpenGLScene** and the **FRDisplayManager**. The former receives a reference to the FROpenGLView and is responsible for setting up the OpenGL-context and initializing the FROpenGLView with that OpenGL-context. In addition to that, **FROpenGLScene** features a timer responsible for the rendering process. **FRDisplayManager** serves as the manager for the displays providing the features of adding a display, removing a display and also saving and loading display setups. The **FRDisplay** class stores all the information of a planar surface and serves as the controller for user-input. **FRDisplay** consists of at least one **FRTetragon** depending on the configuration the user has chosen (the possible configurations will be referred to in the next section). **FRTetragon** is a quadrilateral consiting of four **FRVertex**-objects in a certain order, starting with the left-bottom **FRVertex** A and iterating counter-clockwise over B and C to D. The **FRVertex** class stores a corner-point’s geometry- and texture-coordinates. The **FRVertex**, **FRTetragon** and **FRDisplay** classes are serializable allowing for the mentioned saving- and loading feature. In addition to that, **FRDisplay** composes one **FRQTZTextureRenderer** which is responsible for rendering a composition into an OpenGL-texture at a given framerate and providing access to that rendered texture. In contrast to the **FRQTZTextureRenderer**, the **FRAdaptiveSubdivisionRenderer** class implements the presented adaptive subdivision algorithm and renders the textured display it is associated with.

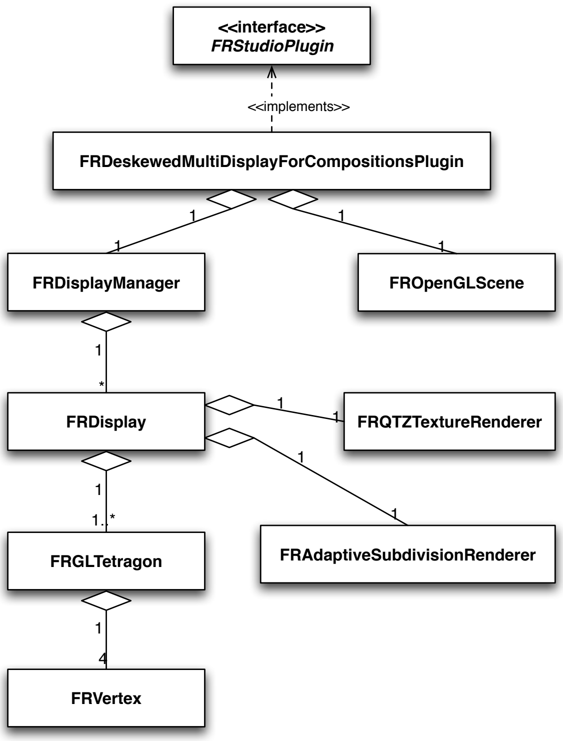


Figure 16: Deskewing Plugin Class Diagram (omitting attributes and operations)

So far, a conceptual overwiev of the deskewing plugin's subcomponents has been presented, while following section deals with the plugin's workhorse: the rendering process.

4.3.2 Rendering Process

The rendering process is structured hierarchically. **FROpenGLScene** is the central rendering component responsible for the rendering process of the complete OpenGL-view. The render timer of **FROpenGLScene** is created at initialisation and then triggers the rendering cycle at a given framerate. Figure 17 illustrates the rendering sequence.

FROpenGLScene first calls its rendering method for the scene ("renderGLScene") which triggers the call to "renderDisplayForTime" for each display. The displays are drawn one after the other beginning with the rendering of the display's texture ("updateTextureForTime") performed by the **FRQTZTextureRenderer**. After that, the texture-mapping takes place and the result (the textured display) is passed to **FRAdaptiveSubdivisionRenderer** ("computeAreaAndDrawQuad") to perform the adaptive subdivision.

Now that the architectural and the sequential points of view have been presented shortly, the next section deals with Farbraum Studio and the deskewing plugin installed from the user's perspective.

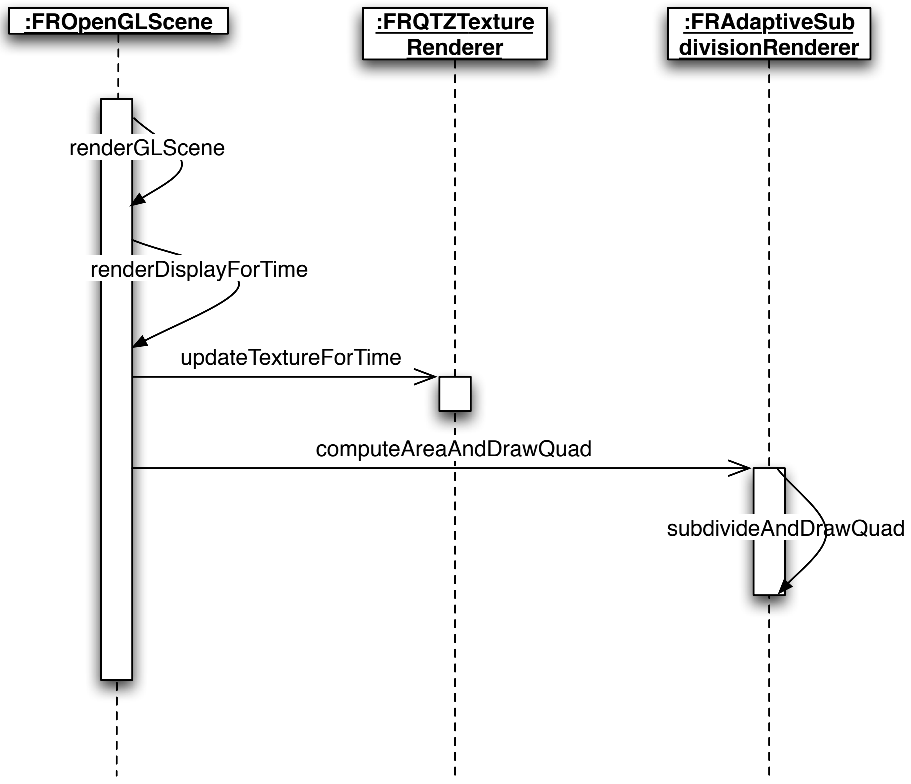
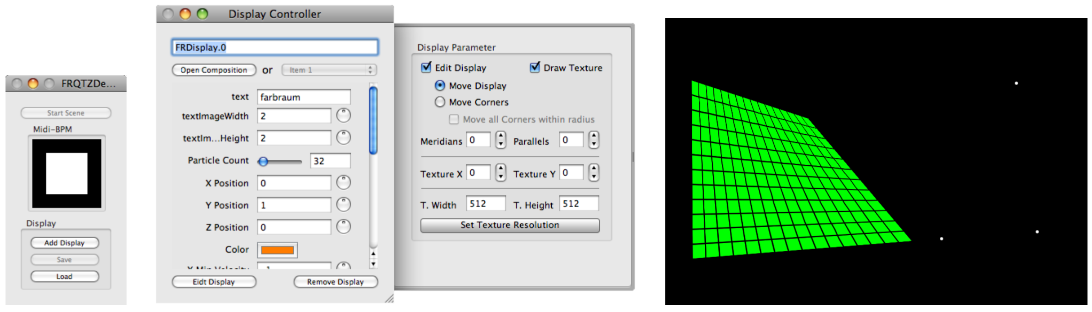


Figure 17: Deskewing Plugin Rendering Sequence

4.3.3 Workflow

When the plugin is installed in Farbraum Studio, at first, the user-interface of the controller appears that is depicted in figure 18(a). Pressing the "Start Scene" button initialises the frameless window and changes the window's background-color to black. Holding the Shift-key, the user can drag the window to the desired position, usually to the target screen of the output (the projector screen).


FRQTZDe...
Start Scene
Midi–BPM
Display
Add Display
Save
Load

Display Controller
FRDisplay.0
Open Composition or Item 1
text: farbraum
textImageWidth: 2
textIm...Height: 2
Particle Count: 32
X Position: 0
Y Position: 1
Z Position: 0
Color
Eidt Display
Remove Display

Display Parameter
[✓] Edit Display [✓] Draw Texture
(●) Move Display
( ) Move Corners
[ ] Move all Corners within radius
Meridians: 0 Parallels: 0
Texture X: 0 Texture Y: 0
T. Width: 512 T. Height: 512
Set Texture Resolution

(a) (b) (c)

Figure 18: Deskewing Plugin Screenshot with Plugin Controller 18(a), Display Controller 18(b) and Output Window 18(c)

Now, the user can add a display by pressing the "Add Display" button which is followed by a popup-dialog asking the user to define the four conrerpoints of the display by clicking on the corners' target destinations inside the window (as figure 18(c) illustrates). Alternatively, the user can load a display-setup with the "Load" button if an already saved setup ("Save") exists. Once a display is created or loaded, for each display a user-interface is created (see figure 18(b)). This display controller features loading a Quartz Composition by pressing the "Load Composition" button which triggers a fileopen-dialog for chosing the desired .qtz-file. Once the composition is loaded, its content starts being rendered into the display's texture. What's more is that if the composition has published ports (compare section 2.2), the corresponding control-elements are loaded into the user-interface and can be adjusted at runtime. Next, the "Remove Display" button does the expected job of removing the display and its user-interface controller. The "Edit Display" button triggers the so called "drawer" to appear (the panel on the right side of figure 18(b)). Here, the user can turn on and off the editing mode of the display's geometry by clicking on the "Edit Display" checkbox. For this, two options exist: either the whole display can be moved by chosing "Move Display" or the cornerpoint(s) can be dragged when "Move Corners" is selected. Moreover, a display's geometry can be splitted into several parts by abjusting the "Meridians" (vetical subdivision) and the "Parallels" (horizontal subdivision) values. In this way, a display's geometry can be seperated into subareas with independent cornerpoints. This feature allows for projecting on more than a flat-shaped plane, on the different sides of a cube for example. The "Texture X" and "Texture Y" values offer the possibility to repeat the texture over subareas if such are present. Thus, the user can chose between distibuting the texture among the subareas or to repeat the texture on each subarea along the horizontal ("Texture X") and the vertical ("Texture Y") axes. The "T. Width" and "T. Height" textfields' values are used to adjust the texture-resolution (in pixels) when the "Set Texture Resolution" button is pressed. The checkbox in the top-right corner of the panel, "Draw Texture" switches between the texture-rendering mode (with the checkbox checked) and the visualisation of the adaptive subdivision algorithm (when the checkbox is unchecked). While the former triggers the display to be rendered with its texture, the latter illustrates the adpative subdivision process by drawing the outlines of the quadrilaterals produced by the algorithm.

This chapter has presented the deskewing plugin's requirements, its design (from the developer's point of view) and its functionality (from the user's point of view). In the following chapter, the results achieved will be summarized.

5 Results

5.1 Functionality of the Framework and the Module

The requirements for the framework (compare Figure 7) have been met: the FRWindow-class provides a frameless window capable of being resized and positioned as desired. It contains the OpenGL-view and both, the view and the window are accessible to plugins. If the plugin registers for the OpenGL-view, it will be informed if the view-size changes. If a plugin is interested in receiving mouse-events, it can register for those and gets informed whenever a mouse-event occurs in the OpenGL-view. Finally, the FRPluginManager-class empowers the FRStudio-user to install and deinstall FRStudioPlugin-interface confrom plugins. The deskewing plugin serves as the proof of concept for the plugin-architecture and the possibility of installing and deinstalling plugins at runtime.

The features described in the worklow-section (see 4.3.3) show that the requirements defined for the deskewing plugin (compare Figure 11) have also been met.

5.2 Deskewing Process

The chosen deskewing process for planar surfaces produced acceptible results. However, using a manual rectification method, the accurateness of the deskewing is exclusively *"achieved by visual judgment of the user"* [Bar09] and thus, can only be as good as the vision of the user. For the purpose of the target setup scenario, this degree of accuracy is sufficient. As soon as the setup involves large projection distances or if the projection plane is not clearly visible to the user, the accuracy that can be achieved by manual deskewing would reach its limits, of course. Referring to the presented target setup (compare figure 10), the actual visual output achieved is depicted in figure 19 while the projection scheme is illustrated in figure 20(a) from top-view and in figure 20(b) from side-view.

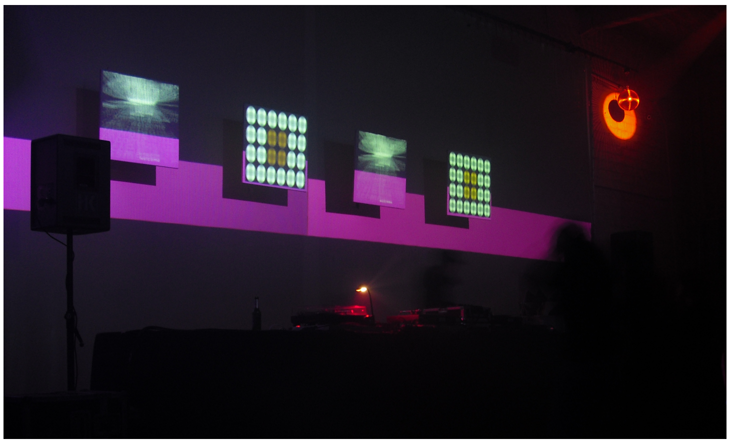


Figure 19: Projection Setup Result

Although the projectiors' positions are perpendicular to the target planes along the horizontal axis, vertical distortion emerges from the projectors' installation position. Additionally, as the top-view projection setup shows (figure 20(a)), we have a setup of five planes: apart from the four planes in the foreground, a fifth plane is defined by the large screen in the background. That one is stretched over the complete angle of beam of the two projectors continuously. What we can achieve as a combination of the manual deskewing feature on the one hand and the "Meridians" feature on the other hand is that we don't have to setup the projectors too accurately. As long as they overlap (a little), a conituous screen can be illuminated with a continuous image. The

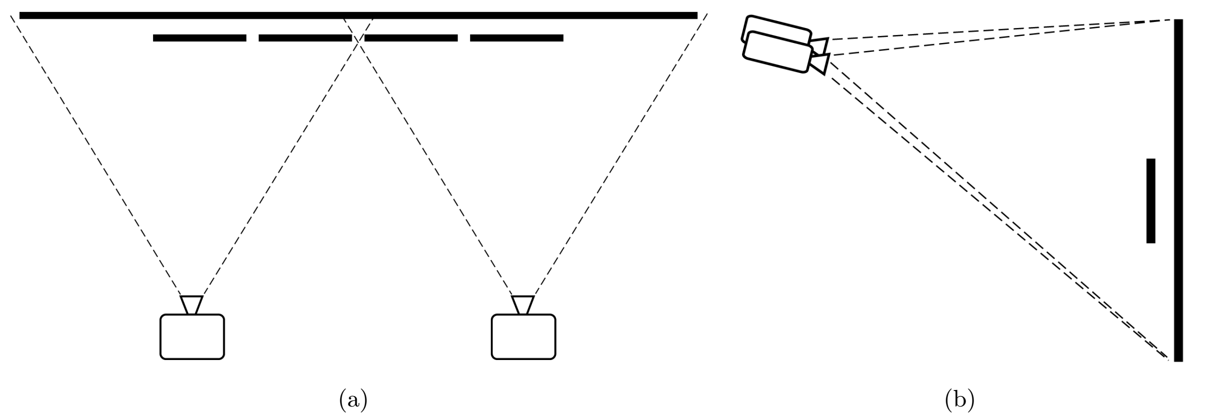


Figure 20: Projection Setup from top-view 20(a) and from side-view20(b)

overlapping region that can be seen in figure 20(a) results from the slightly overlapping projector images represented by the intersecting beams of the two projectors in figure 20(a). The user can divide the background plane into two halves by creating a "Meridian" and correct the overlapping artifact by dragging the cornerpoints of the halves to the distinct positions. In this way, the images of the two projectors which would produce undesired effects in the overlapping region, can be manually corrected. The visual result of the overlapping region that would be produced without correction is depicted figuratively in figure 21(a) and the corrected version is shown figuratively in figure 21(b) leading to the desired result presented in figure 22.

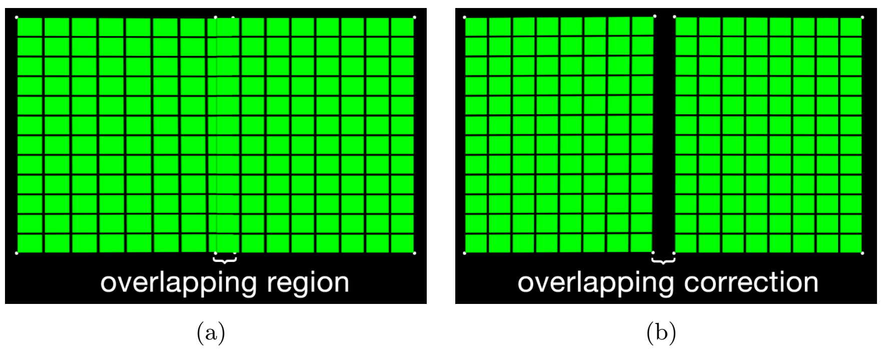


Figure 21: Overlapping Region Correction

Another important point is that the manipulation of the geometry-values limited to two dimensions (x- and y-values) unfortunately introduces projection errors due to the linear interpolation of the adaptive subdivision procedure. To illustrate that error, a circle is assumed as the target texture for a plane that is heavily deskewed as figure 23 depicts. Since the geometry is interpolated linearly, a correct circle-projection cannot be achieved. Instead of projecting an ellipse that would produce the correct result of a circle on the target plane, the used method produces an egg-like shape. If the geometry would be subdivided bilinearly, this error could be avoided.

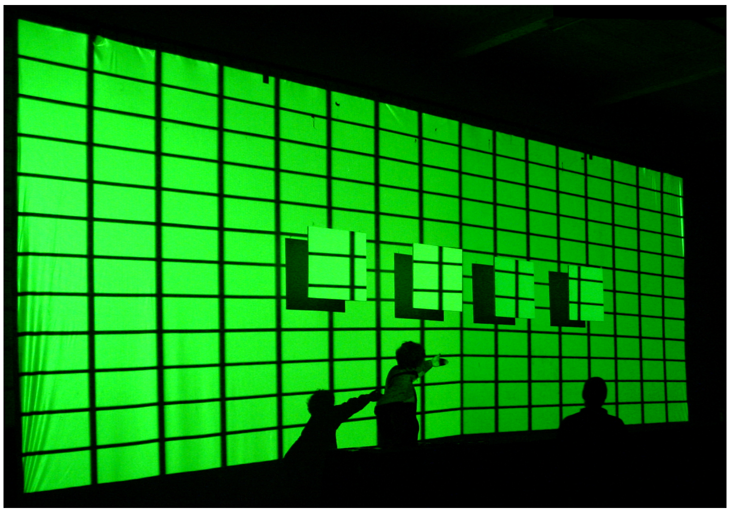


Figure 22: Overlapping Region Correction Result

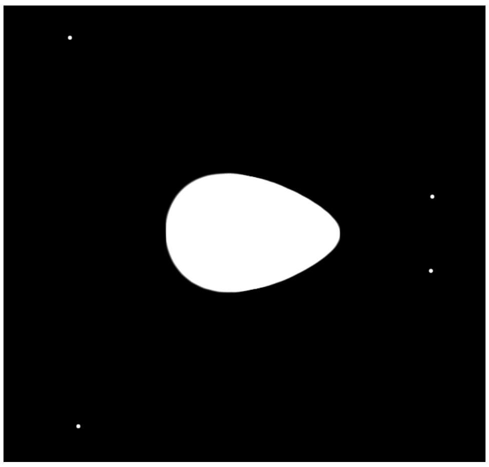


Figure 23: Erroneous Circle Projection Introduced by Linear Subdivision

6 Conclusion

Now that the results of the produced software have been presented, it seems quite interesting to compare them to other applications designed for VJ-performances. In addition to that, a short summary of the subject will be provided in the following and conclusively, the lessosns learned lead to the thoughts of improvement and possible future work.

6.1 Comparison with Selected State of the Art VJ-Software

There are numerous VJ-applications one can find browsing the internet. www.softwarevj.com [Sof09] lists over 80 applications and describes their features. I have chosen three applications to compare to the Farbraum Studio and the deskewing plugin: the already introduced Quartz Composer, Resolume and Modul8. The reason why those three were selected is rather personal on the one hand but on the other hand these applications are very widespread among VJ-artists. Especially Modul8 is granted a very high reputation among VJs. The comparison focuses only on the features that are the subject of this paper: the first criterion is whether the software features rendering Quartz Compositions including the dynamic generation of a user-interface for the published ports of a composition. The second criterion is whether the examined software offers a deskewing mechanism.

Obviously, **Quartz Composer** is capable of rendering Quartz Compositions and also provides a feature for controlling the published input ports at runtime. However, there is no patch for deskewing planar surfaces as far as i could find out.

Modul8 [Gar09] is a realtime video mixer available for OSX. It offers a very complex but powerful user-interface and thus many features. Based on a layer metaphor, the user can compose videos in realtime, taking advantage of graphics-hardware acceleration. Additionally, the user can configure multi-projection setups, apply numerous effects and filters on the videos and record the generated output. Moreover, the application provides a modular architecture allowing for the development of custom modules. Concerning the deskewing ability, a filter called "Perspective Transform" exists which can map a video-arranegment to the desired target points of the output-screen. However, the subdivision of a video-arrangement into different parts is not possible whereas the Farbraum Studio's deskewing plugin is capable of doing that (the "Meridians" and "Parallels" feature as described in section 4.3.3). Although Modul8 is capable of rendering Quartz Compositions as textures, currently (version 2.5) there is no feature to control the published input ports of a composition through a dynamically generated user interface as far as I know.

The current version (version 3) of **Resolume** [EdK09] offers the possibility to mix videos in realtime and to operate on multiscreen-projection setups. For this purpose, three layers can be composed and mixed with each other applying different blending options. What's new in version 3 is that the user can connect visual effects to audio effects if Resolume is used for audio-output. The software is available for OSX and for Windows. The feature of deskewing planes is embedded in the so called "keystone plugin" while there is no possibility to import Quartz Compositions in the current version.

Due to time limitations i was not able to examine further VJ-applications closely enough but nevertheless, **VDMX** [Vid09] and **vvvv** [vg09] have to be mentioned in this context and might be interesting for furrther investigation. While the former is a realtime video studio available for OSX and comparable to Modul8, the latter is a toolkit for realtime video synthesis available for Windows and comparable to Quartz Composer because it also uses a visual programming interface.

6.2 Improvements and Future Work

Of course, the framework and the deskewing plugin could be significantly improved. As described in secion 4.3.3, the texture resolution the Quartz compositions are rendered at have to be set manually by the user. This is a rather suboptimal circumstance especially from the user's point of view since it it not always clear what resolution a plane should be rendered at. Thus, the best would be if the plugin did the job taking into consideration that the area in pixels is already calculated by the FRAdaptiveSubdivisionRenderer-class.
Next, the useability of the plugin could be enhanced providing the user a better workflow. For this purpose, keyboard-control would be adequate. Keyboard-events could be embedded in the framework and plugins could register for them in the same way the mouse-events can be retrieved by plugins. The keyboard-control could be used to make a plane's cornerpoints adjustable on a per pixel basis by the arrow-buttons of the keybard for example, providing a nice fine-adjustment of the cornerpoints. What also would upgrade the useablity is the presence of a control window that is a copy of the output window. In this way, the user had a screen that would help him with the orientation of the mouse-pointer which is often difficult to control on a distorted projection image, especially when the view on the target screen is not the best.
In addition to the feature of dragging the vertices, that means the geometry-coordinates, dragging the texture-coordinates would be a nice-to-have feature, allowing for a more precise configuration. If we imagine a projection into a corner and the two plane-halves on the two different walls are different in size, the two equal-sized texture-halves would be distributed over the two walls the way the deskewing plugin is implemented. This would lead to the undesired effect that on the smaller half, the texture would be compressed while on the bigger half, the texture would be stretched. Dragging texture-coordinates would empower the user to correct that effect manually. Thinking about that feature and examining if it was provided by other applications (see section 6.1) led me to the following considerations: manipulating the geometry-coordinates is a nice feature, however, it has two essential disadvantages the way that i put it into practice: first, manipulating the x- and y-values of the geometry only led to the rendering artifacts of non-parallelograms. Although this error could be compensated for by applying the adaptive subdivision, this method introduced another drawback that was described in section 5.2: projected circles are deformed as a result of the the linear interpolation performed on the geometry by the subdivision of the adaptive subdivision algorithm. If the subdivision would be bilinear, this error would not be present. Alternatively, the geometry could be transformed three-dimensionally as is the case in the procedures presented in [LOU^{+}06] or [Lim07]. Another idea would be the following: instead of manipulating geometry-coordinates we could define the geometry to be a rectangular plane that ranges over the entire output-screen. Defining multiple planes could then take place in the texture-domain exclusively. In this way, the adaptive subdivision process would become obsolete since the geometry would be defined by a rectangular planar surface. Thus, the described artifacts introduced by the triangulation-based rendering of graphic-cards would not occur since the geometry would be a rectangular planar surface (leading to two equally-sized triangles after the triangulation). In addition to that, OpenGL offers the feature of bilinear texture sampling within the minification and magnification filters (compare [BSW^{+}04, p. 402] that could be used to achieve a bilienar interpolation of the output-image of planes. In this way, the mentioned problem of incorrect circle-projection could also be solved if the feature of subdividing planes with "Meridians" and "Parallels" (see section 4.3.3) could be completely transferred from the geometry-domain into the texture-domain. In particular, OpenGL's glTexSubImage2D([...])-operation [BSW^{+}04, p. 376] might be a possible method to achieve the transfer from the geometry-domain into the texture-domain. However, whether this is a theoretically plausible alternative and if it is practically possible has to be investigated!

Another issue is the setup of a target plane that spans over the image of several projectors (figure 19 for example): as the foto shows, the intersection of the two images of the different projectors is clearly visible. Of course, this should be avoided in order to achieve the impression of a continuous image. This issue is a matter of projector calibration (color-space, contrast, saturation etc.) concerning the hardware but from the software's point of view, an edge-blending mechanism could be provided.
Although i pointed put in section 3.3 that it would be inappropriate to develop an OSX-framework for the reason that *"Frameworks are most effective when their code is shared by multiple applications"* [App09a], now i realise that it could be the different plugins that share their code. For example, the FRVertex-class, the FRGLTetragon-class or the FRQTZCompositionRenderer-class would be candidates that, apart from the FRWindow- or FROpenGLView-classes, could be embedded into the framework and plugins could then simply instantiate them instead of implementing.

6.3 Summary

Now that the first version of FR Studio is implemented, the common ground for the future work of VJ-applications is laid for "Farbraum". It allows for developing new modules under the presented plugin-architecture, empowering us to configure a desired set of plugins at runtime. The framework provides the basic features that have been common for the applications created so far, and now those features are available for plugins trough an API. In addition to that, the deskewing plugin solves the problem of projecting at an oblique angle using a manual correction method: the user has the possibility to drag the cornerpoints of projection planes to their target destinations with the mouse. Moreover, Quartz compositions are rendered as textures of the projection planes and the compoisitions can be controlled at runtime by their published ports' user interface that is dynamically generated.

7 Acknowledgements

I would like to thank Prof. Dr. Stefan Müller for the supervision of this work. Not only did he support my idea for the subject and motivated me to follow my personal interest leading to this work, but what's even more important, helped me with precise and inspiring critique during the time of development.

Next, i want to thank Dr. Marcin Grzegorzek for mentoring. He made me profit by his experience as a scientist, giving me advice at exactly the right moments and contents. It was also thanks to him that i even had the possibility to write my Studienarbeit upon that subject.

Without a doubt, this work had not been possible without "Farbraum", that's why i am very grateful to Jens Barth and Christof Pohl. It was them who introduced me to the world of VJing from both, the designer's and the develper's point of view. Moreover, their experiences in developing VJ-applications laid the basis for the achievements of Farbraum Studio and the deskewing plugin. I hope that this work contributes to the evolution of Farbraum and that Jens and Christof can profit from it as i could profit from their efforts.

As a matter of course, my beloved family is the basis of all and it was my parents who gave me the possibility for studying at the university. Apart from the materialistic grounds they laid for me, i want to thank them for their emotional support and for their pride and confidence in me.

It is very important to thank my girlfriend Melina for everything she gave me. Not only do i treasure the love that inspires me during all of my work, but i am so grateful for her interest in the subject and for the inspiring comments she made, always believing in me and my ideas.

This work would not have been possible without the support of Holger Kesselheim who provided us access to the projector equipment of the university. That's why a a special thanks goes out to him.

Last but not least, i send props to my friends for their support. They made my considerations flourish by profitable discussions on the topic and moreover, gave me useful technical advice. Especially it is Christof Pohl, Daniel Brehme, Sebastian Vetter, Christian Latsch and Frederik Jochum who have to be mentioned for their contributing efforts.

References

[App09a] Apple. Guidelines for creating frameworks [online]. 2009. Available from: http://developer.apple.com/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/CreationGuidelines.html [cited 03 May 2009]. 9, 27

[App09b] Apple. Introduction to dynamically loading code [online]. 2009. Available from: http://developer.apple.com/documentation/Cocoa/Conceptual/LoadingCode/LoadingCode.html [cited 15 March 2009]. 10

[App09c] Apple. Osx graphics and media [online]. 2009. Available from: http://developer.apple.com/leopard/overview/graphicsandmedia.html [cited 03 May 2009]. 8

[App09d] Apple. Osx system architecture [online]. 2009. Available from: http://developer.apple.com/macosx/architecture/index.html [cited 03 May 2009]. 8

[App09e] Apple. Plugin architectures [online]. 2009. Available from: http://developer.apple.com/documentation/Cocoa/Conceptual/LoadingCode/Concepts/Plugins.html [cited 21 March 2009]. 11

[App09f] Apple. Quartz composer [online]. 2009. Available from: http://developer.apple.com/mac/articles/graphicsmedia/quartzcomposerinleopard.html [cited 02 May 2009]. 6

[App09g] Apple. What are frameworks [online]. 2009. Available from: http://developer.apple.com/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html [cited 03 May 2009]. 9

[Bar09] Jens Barth. A proxy user interface for multi-display environments. Master's thesis, Universität Koblenz-Landau, Campus Koblenz, Fachbereich Informatik, Computervisualistik, February 2009. 2, 22

[Bro02] Rainer Brockerhoff. Plugged-in cocoa [online]. 2002. Machack 2002 papers competition. Available from: http://www.brockerhoff.net/pap.html [cited 03 May 2009]. 10

[BSW$^+$04] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide Fourth Edition*. Addison-Wesley, 2004. 26

[EdK09] Bart van der Ploeg Edwind de Koenig. Resolume avenue [online]. 2009. Available from: http://www.resolume.com/ [cited 29 April 2009]. 25

[Gar09] GarageCube. Modul8 [online]. 2009. Available from: http://www.modul8.ch/ [cited 29 April 2009]. 25

[Lim07] Florian Limburg. Geometriekorrektur von stereo-projektionen. Master's thesis, Universität Koblenz-Landau, Campus Koblenz, Fachbereich Informatik, Computervisualistik, March 2007. 2, 26

[LOU$^+$06] Marcel Lancelle, Lars Offen, Torsten Ullrich, Torsten Teichmann, and Dieter W. Fellner. Minimally invasive projector calibration for 3d applications. *Virtuele und Erweiterte Realität, 3. Workshop der GI-Fachgruppe VR/AR*, 0:195–201, 2006. 2, 26

[Oli01] Manuel M. Oliveira. Correcting texture mapping errors introduced by graphics hardware. *Computer Graphics and Applications, Pacific Conference on*, 0:31–38, 2001. 16, 17

[Sof09] SoftwareVJ. Softwarevj [online]. 2009. Available from: `http://www.softwarevj.com/` [cited 02 May 2009]. 25

[vg09] vvvv group. vvvv - a multipurpose toolkit [online]. 2009. Available from: `http://vvvv.org` [cited 15 May 2009]. 25

[Vid09] Vidvox. Vdmx 5 - realtime video studio for professional vjs [online]. 2009. Available from: `http://www.vidvox.com` [cited 15 May 2009]. 25

[Wik09a] Wikipedia. Framework [online]. 2009. Available from: `http://en.wikipedia.org/wiki/Framework` [cited 29 April 2009]. 2

[Wik09b] Wikipedia. Modular design [online]. 2009. Available from: `http://en.wikipedia.org/wiki/Modular_design` [cited 29 April 2009]. 2

[Wik09c] Wikipedia. Vj [online]. 2009. Available from: `http://en.wikipedia.org/wiki/VJ_(video_performance_artist)` [cited 28 April 2009]. 2