Porting Visualization Toolkit to OpenGL ES 2.0 and iPad

SASON OHANIAN SAKI

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2012
Porting Visualization Toolkit to OpenGL ES 2.0 and iPad

SASON OHANIAN SAKI

DD221X, Master’s Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 270 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Lars Kjelldahl
Examiner was Olle Bälter

TRITA-CSC-E 2012:034
ISRN-KTH/CSC/E--12/034--SE
ISSN-1653-5715
Abstract

Visualization Toolkit (VTK) is an open source, cross-platform library for visualization, 3D graphics and image processing, written in C++ and OpenGL. In this report we investigate how VTK can be extended and modified to add support for the iPad. In particular the rendering library of VTK was investigated and its library dependencies. We found that the rendering implementation of VTK makes wide use of the fixed function pipeline of OpenGL, which is unsupported in OpenGL ES 2.0 (GLES) used on iPad. Consideration was taken into the platform specific assumptions made by VTK and the use of unsupported code and external libraries. The conclusion is that the build system of VTK (CMake) needs revision and extension to allow the generation of a compiling XCode iOS project. Furthermore we investigate the window handling and interaction frameworks of VTK to plugin and use iPad specific windowing, views and multitouch events.
Portning av Visualization Toolkit till OpenGL ES 2.0 och iPad

Sammanfattning

# Table of Contents

1 Introduction ................................................................................. 1

2 Problem Statement ................................................................... 1

3 Delimitations ............................................................................ 1

4 Background ............................................................................. 2
  4.1 VTK .................................................................................. 2
  4.2 VTK Graphics Model .......................................................... 2
    vtkProp/vtkProp3D (super-class of vtkActor, vtkActor2D, vtkVolume) .... 2
    vtkLight ............................................................................. 2
    vtkCamera ........................................................................... 3
    vtkProperty, vtkProperty2D, vtkVolumeProperty ...................... 3
    vtkMapper, vtkMapper2D, vtkVolumeMapper .. ...................... 3
    vtkRenderer, vtkRenderWindow ....................................... 3
  4.3 OpenGL ............................................................................. 4
  4.4 OpenGL ES ....................................................................... 5
  4.5 CMake ............................................................................. 6
    Conditional statements ......................................................... 7
    Looping constructs ............................................................. 7
    Procedure definitions ........................................................ 7
    Hello World Example ......................................................... 8

5 Building VTK ........................................................................... 9
  5.1 CMake ............................................................................. 9
  5.2 XCode ............................................................................ 12
  5.3 iOS - Simulator and Hardware ........................................ 13

6 Compiling and Running VTK on iOS ...................................... 15
  6.1 System overview ................................................................. 15
  6.2 vtkRendering Target Dependency Hierarchy ..................... 16
  6.3 Modifying Targets ............................................................... 17
    Vtkftgl and vtkFreetype ...................................................... 17
    VtkGraphics, vtkCommon, vtkFiltering and vtkVerdict ............. 17
    VtkImaging ........................................................................ 17
    VtkIO .............................................................................. 17
    VtkParseOGLExt .................................................................. 18
    VtkEncodeString ............................................................... 18
    VtkRendering .................................................................... 18

7 VTK Frameworks .................................................................... 19
  7.1 Rendering .......................................................................... 19
    VtkRenderWindow ................................................................ 20
    vtkRenderer ..................................................................... 21
    Extending VTK factories ..................................................... 22
  7.2 Window and View handling ............................................... 23
    UIWindow .......................................................................... 23
    UIView ............................................................................ 24
  7.3 Porting OpenGL to OpenGL ES 2.0 .................................. 24
    Shaders ............................................................................ 24
  7.4 Interaction .......................................................................... 26
    VTK interaction framework ................................................ 26
8 Conclusions........................................................................................................... 28
 8.1 Why not OpenGL ES 1.1?.................................................................................. 29
 8.2 Future work........................................................................................................ 29
References ................................................................................................................. 30
1 Introduction

VTK (Visualization Toolkit) is an open source, cross platform library for visualization, 3D graphics and image processing. VTK is supported by a very large community and is used in many fields, including, medical applications and physics simulation.

Given the wealth of tablet computers emerging on the market, there has recently been a large interest in the community for iOS support in VTK, or more generally, embedded systems support. For the moment VTK cannot be run on an iPad, because of a number of concerns, one of which is the lack of OpenGL support.

It is far from trivial to make VTK compatible with the iPad/iOS. There are many more obstacles beyond graphics rendering that have to be taken into consideration in order to port VTK.

In this report we investigate how VTK works at a technical level in order to get an understating of how VTK can be modified to support OpenGL ES rendering and iOS.

2 Problem Statement

The work described in this report aims primarily at answering the following question:

*How may the Visualization Toolkit, which currently supports the operating systems Windows, Macintosh and Linux, and renders its graphics in OpenGL, be adapted to run on iPad/iOS?*

This general problem statement can be broken down into sub-problems.

1. What platform specific assumptions does VTK make? What has to be modified, added or removed in order for VTK to compile and run on iOS?
2. How can VTKs graphics rendering be ported or re-implemented to support OpenGL ES 2 and iOS.
3. How is user interaction handled in VTK (e.g. mouse and keyboard input), and what needs to be modified in VTK to support multi touch interaction in iOS?

3 Delimitations

The scope of the question asked is potentially very large. In order to narrow down the area of investigation the work will primarily concentrate on answering the question at a general level, which means that the aim is not to make a runnable prototype of VTK on iOS, neither to give a detailed implementation proposal on how to make all parts of VTK iOS compatible.

The main aim is to answer sub-question one and two, which are assumed to be essential for making a proof of concept possible.
The work will only concentrate on investigating the rendering capabilities of VTK, and its dependencies. Higher-level VTK libraries will be left out.

4 Background

4.1 VTK

VTK (Visualization Toolkit) is an open-source software system for 3D graphics, image processing, and visualization [1]. VTK consists of a C++ class library with several interpreted interface layers including Tcl/Tk, Java and Python. VTK is cross-platform and runs currently on Linux, Windows, Macintosh and Unix.

VTK had a number of design goals in its early stages [2]. One is the toolkit philosophy that focuses on having well defined pieces of software with simple interfaces, which could be assembled into larger systems.

The system should have an interpreted layer with a crisp boundary from the compiled core.

The system needed to be portable. This suggests a high-level abstraction of 3D graphics that would be independent of graphics libraries such as OpenGL. There are also issues with different windowing systems on different platforms. The core system needed to be independent of windowing systems.

Other important design goals were, to use standard components and languages, make the software freely available as open-source and to make the system as simple as possible for the end user who might not have any knowledge of computer graphics and visualization.

4.2 VTK Graphics Model

The graphics model of VTK is defined to be easy to understand and use. It consists of a couple of basic core objects [3] that combined together creates a scene.

\textbf{vtkProp/vtkProp3D (super-class of vtkActor, vtkActor2D, vtkVolume)}

Props represent the things that we “see” in a scene. Props can be positioned and manipulated. Props do not directly represent their geometry; instead they refer to mapper objects, which are responsible for representing data. A prop also refers to a property object, which controls the appearance of the prop such as color, lighting effects, etc.

\textbf{vtkLight}

Lights are used to represent and manipulate lighting in a scene. Lights are only used in 3D.
**vtkCamera**

The camera object controls how 3D geometry is projected onto the 2D image plane during the rendering process. It defines amongst other characteristics, the view, position and focal point.

**vtkProperty, vtkProperty2D, vtkVolumeProperty**

These objects represent the property objects that instances of vtkProp refer to.

**vtkMapper, vtkMapper2D, vtkVolumeMapper**

Mappers are used to transform and render geometry.

**vtkRenderer, vtkRenderWindow**

These objects are used to manage the interfaces between the graphics engine and the computers’ windowing system.

Figure 1 shows an example of the inheritance of rendering classes in VTK in order the meet the above-mentioned requirements.

![Figure 1: Inheritance diagram for vtkRenderWindow and subclasses](image)

**VtkObject** is the base class for most objects in VTK, it provides methods for tracking modification time, debugging, printing, and event callbacks.

**VtkWindow** is an abstract object for specifying the behavior of a rendering window.

**VtkRenderWindow** is an abstract object for specifying the behavior of a rendering window. It adds methods for synchronizing the rendering process, setting windows size, control double buffering, support for stereo rendering and more.
VtkOpenGLRenderWindow and vtkMesaRenderWindow are concrete implementations of vtkRenderWindow. VtkOpenGLRenderWindow interfaces with the OpenGL graphics library. Application programmers use vtkRenderWindow instead of the OpenGL specific version. A factory class determines which renderer is used.

vtkCocaRenderWindow and the other objects that inherit from vtkOpenGLRenderWindow are platform specific implementations of vtkOpenGLRenderWindow that handles OS specific windowing.

4.3 OpenGL

OpenGL is a standard specification defining a cross-language and cross-platform API for 2D and 3D graphics. OpenGL was developed by Silicone Graphics Inc. in 1992 and is used in many applications, amongst others, CAD, visualization and video games. Khronos Group [4] now manages OpenGL.

The purpose of OpenGL is to hide the complexities of interfacing with different graphics hardware by presenting a single, uniform software interface.

OpenGL accept geometric primitives as input and converts them into pixels. This is done in the graphics pipeline known as the OpenGL state machine [5]. The OpenGL commands issues primitives and configuration to be processed by the pipeline. Prior to OpenGL 2.0, each stage of the pipeline performed a fixed function and the user could make configurations to a small extent. In version 2.0 and later the user is able to fully program parts of the pipeline using vertex- and fragment (also know as pixel) shaders as seen in Figure 2 and Figure 3.

Figure 2: Simplified OpenGL fixed function pipeline prior to OpenGL 2.0
Figure 3: Simplified OpenGL programmable pipeline, OpenGL 2.0 and later

4.4 OpenGL ES

The “ES” in OpenGL ES stands for Embedded System. There are three versions of OpenGL ES released so far: OpenGL ES 1.0, OpenGL ES 1.1 and OpenGL ES 2.0. This work will concentrate on OpenGL ES 2.0, which is defined relative to the OpenGL 2.0 specification.

This section briefly outlines the OpenGL ES 2.0 specification relative to the OpenGL 2.0 specification. Henceforth OpenGL ES will be referred to as GLES.

The most significant changes in GLES 2.0 relative to GLES 1.1 are that the fixed function transformation and the fragment pipeline are not supported. Commands in GLES 2.0 cannot be placed in a display list for later processing, there is no polynomial function evaluation stage and blocks of fragments cannot be sent directly to individual fragment operations. GLES 2.0 is designed for a shader based pipeline and has no support for the fixed function pipeline. This means that a developer is forced to use and need to have knowledge of shaders.

GLES 2.0 does away with the glBegin/glEnd paradigm that was used for specifying geometry. GLES 2.0 draws geometric objects exclusively using vertex arrays. Support for vertex position, normal, colors and texture coordinates are removed since they can be represented using vertex arrays.

The fixed function transformation pipeline can be omitted by calculating the necessary matrices in the application and load them as uniform variables in the vertex shader. The code that transforms vertices will now be executed in the vertex shader.
Clipping against the view frustum is supported, while user-specified clip planes are not. User-specified clip planes can be emulated in the vertex shader.

The fixed function lighting model is no longer supported. Lighting can be implemented by writing appropriate vertex and/or pixel shaders.

The texture environments present in OpenGL 2.0 are no longer supported. Fragment shaders can replace the fixed functions texture functionality.

Stencil tests and blending are supported with minor changes while alpha test is removed since this can be done inside a fragment shader.

4.5 CMake

CMake is used by VTK to manage the build process. It is a cross-platform, open-source build system designed to build, test and package software. CMake is compiler independent and is designed to support directory hierarchies and applications that depend on multiple libraries. CMake generates native makefiles and workspaces that can be used in the compiler environment of your own choice. The only requirement CMake has is a C++ compiler. VTK uses CMake to support a platform independent build process.

CMake uses CMakeList.txt files, which contain project parameters and describe the flow control of the build process [7]. These files are placed in every directory that wishes to control the build process. The CMakeList.txt file include one or more commands using the syntax "COMMAND(args...)", where "COMMAND" represent the name of the command and "args" a list of arguments. The basic data type in CMake is a string, but it also supports lists of strings.

The following statement groups together multiple arguments into a list using the SET command.

\[ \text{set} (\text{Foo} \ a \ b \ c) \]

This results in setting the variable Foo to a b c, and Foo can be passed into another command using

\[ \text{command} (\${\text{Foo}}) \]

Which is equivalent to

\[ \text{command} (a \ b \ c) \]

Furthermore CMake provides flow control structures much like a regular programming language.
Conditional statements

\[ \text{if}(\text{var}) \]
\[ \quad \text{some\_command}(\ldots) \]
\[ \text{endif}(\text{var}) \]

Looping constructs

\[ \text{set}(\text{Foo a b c}) \]
\[ \text{foreach}(f \ {\$}\{\text{Foo}\}) \]
\[ \quad \text{some\_command}(f) \]
\[ \text{endforeach}(f) \]

Procedure definitions

Functions create a local scope for a variable, and macros use a global scope. The following is an example of a macro and its corresponding function:

\[ \text{macro}(\text{loop\_foo Foo}) \]
\[ \quad \text{foreach}(f \ {\$}\{\text{Foo}\}) \]
\[ \quad \quad \text{some\_command}(f) \]
\[ \quad \text{endforeach}(f) \]
\[ \text{endmacro}(\text{loop\_foo}) \]

\[ \text{function}(\text{loop\_foo Foo}) \]
\[ \quad \text{foreach}(f \ {\$}\{\text{Foo}\}) \]
\[ \quad \quad \text{some\_command}(f) \]
\[ \quad \text{endforeach}(f) \]
\[ \text{endfunction}(\text{loop\_foo}) \]

The macro or function can then be called by writing:

\[ \text{set}(\text{Foo a b c}) \]
\[ \text{loop\_foo}(\{\text{Foo}\}) \]
**Hello World Example**

The figures below are examples of how a simple “Hello World” project [8] can be structured and configured using CMake.

---

**Figure 4:** Example of CMake directory structure for "Hello World" application

---

**Figure 5:** CmakeList.txt files in "Hello World" application
5  Building VTK

5.1  CMake

Learning how to build VTK for iOS using CMake is not the main focus of this work, however it became impossible to ignore. Some minor changes and settings was made to the CMake configuration of the project in order to get a better starting point for the XCode project that is generated by CMake.

Since VTK does not support iOS for the moment, the CMake project is not customized to support building of an iOS XCode project. To customize CMake files in VTK for complete iOS support would require a great deal of work. This is not the aim of the thesis and was therefore mostly omitted.

In order to gain iOS support in the CMake build process one needs to customize CMake to cross-compile for iOS by adding iOS specific logic. This means that new files have to be included in the project and any required platform libraries such as the iOS SDK must be linked into the build. Since iPad does not support OpenGL, which VTKs rendering is implemented in, all OpenGL implementations must be switched out to OpenGL ES implementations. To do this, it is necessary to know which implementing files these changes concern. These files must be added to the CMake project hierarchy, and relevant CMake files have to be modified to add the correct files to the build depending on which platform we are building for.

An example of where cross-compiling support needs to be added is in the directory rendering, which is located directly under the projects root directory. There exists a “CMakeLists.txt” in the rendering directory, which describes, amongst other things, which files that are included in a specific target, in this case the rendering target. There are also references to eventual dependencies to other targets, platform concerns and more. The figure below is an extract of the mentioned CMake file.
The Kit_SRCS set contains files in the rendering target, which are classes general to the VTK rendering framework, independent of the concrete rendering implementation. The framework classes are inherited by concrete rendering implementations like the ones in KitOpenGL_SRCS. These files are concrete OpenGL implementations and need therefore to be omitted when building the project for iOS. We instead need to include new OpenGL ES specific class files. Figure 7 is a simplified example of how one could modify the CMakeLists.txt file to support building for iOS. Changes are marked with bold text.

```cpp
/* vtk_root/rendering/CMakeLists.txt */

SET(KIT Rendering)  
SET(UKIT_RENDERING)  
SET(KIT_TCL_LIBS vtkGraphicsTCL vtkImagingTCL ${VTK_TK_LIBRARIES})  
SET(KIT_PYTHON_LIBS vtkGraphicsPythonD vtkImagingPythonD)  
SET(KIT_JAVA_LIBS vtkGraphicsJava vtkImagingJava)

IF (JAVA_AWT_LIBRARY)  
  SET(KIT_JAVA_LIBS $(KIT_JAVA_LIBS) $(JAVA_AWT_LIBRARY))  
ENDIF (JAVA_AWT_LIBRARY)

SET(KIT_INTERFACE_LIBRARIES vtkGraphics vtkImaging)  
SET(KIT_LIBS vtkIO vtkftgl  
  ${VTK_FREETYPE_LIBRARIES})

#INCLUDE(${VTK_CMAKE_DIR}/vtkTestGL.cmake)  
#INCLUDE(${VTK_CMAKE_DIR}/vtkTestGLX.cmake)

IF(APPLE AND VTK_WRAP_JAVA)  
  ADD_DEFINITIONS("-ObjC++")  
ENDIF(APPLE AND VTK_WRAP_JAVA)

SET( Kit_SRCS  
  vtkActor.cxx  
  vtkCamera.cxx  
  /* Additional files included in Kit_SRCS set */  
)  
...  
SET( KitOpenGL_SRCS  
  vtkOpenGLActor.cxx  
  vtkOpenGLCamera.cxx  
  /* Additional files included in Kit_OpenGL set */  
)
```

Figure 6: Extract from CMakeLists.txt under VTK rendering directory
In order for the example to work, the IOS_IPAD and USE_GLES variables must be defined and set somewhere. This is done in a toolchain [9] configuration file, which in short is a file that defines the settings for the build environment and
target platform. Among these are: system properties, compiler and compiler options, processor architecture, linked libraries and build properties. The toolchain file is passed as an argument to CMake when building.

5.2 XCode

After configuring CMake, an XCode project can be built by using a CMake command line tool or the CMake GUI.

Since the CMake files were not adapted to cross-compile and build for iOS, the delivered project will have build settings matching the environment it was built on, in this case OS X. The settings involve architecture, build location, build options, code signing, linking, packaging, search paths and more. Some of these settings have to be changed in order to compile the code for the iOS platform.

![Figure 8: Xcode build settings](image)

The following changes where made to the build settings for each target in the VTK project.

**Architecture:**

The hardware architecture of a Mac differs from that of an iOS device. The target architecture must be changed. Changed from “32/64-bit intel” to “Optimized armv7”
Base SDK:

The iPad does not support the Mac OS X SDK. The Base SDK flag has to be changed to the iOS SDK that is being used. Changed from “Mac OS X 10.6 SDK” to “iOS 4.3”

Supported platform:

Changed from “macosx” to “iphonesimulator iphoneos”

Valid architectures:

Changed from “i386 X86_64” to “armv6 armv7”

iOS deployment target and Targeted Device family:

These flags determine which iOS SDK and device the project uses in deployment. In this case iOS 4.3 and iPad.

Code signing:

To be able submit an application to the AppStore the code must be signed [10]. To do this, a developer profile has to be set up in XCode. When a profile is configured correctly the code signing is set up using the code signing identity flag, where a profile is selected for signing. Code signing will not be necessary in this case since there will be no attempt at submitting a working application to the AppStore, however an Apple approved developer profile is still needed to be set up in order to test the software on an actual iOS device.

5.3 iOS - Simulator and Hardware

To compile, build and run a target, a platform on which to compile the code on must be selected for the requested target. This is easily done in XCode via the GUI. Available platform are dependent on the build settings for each target. In this case, since the build settings where changed to be iOS compatible, the selectable platforms are “iPad 4.3 Simulator” and “iOS device”. If the application is built and executed using the iPad simulator, the application is launched in a simulated iPad environment on the local development machine. If we instead choose to build and run on an iOS device, an iPad has to be connected to the computer via usb, and a developer profile must be set up as described earlier.
Figure 9: Running an OpenGL ES 2 application on an iPad Simulator in XCode
6 Compiling and Running VTK on iOS

6.1 System overview

VTK consists of several targets with some interdependency. The targets are considered to be logical delimited parts of the system. Below are some examples of the more central targets in play.

vtkCommon
Contains the most general type of classes. Most other targets depend on vtkCommon. The target provides some basic patterns used by the system, and an extensive amount of data structures such as vtkObjectBase, vtkObject, vtkObjectFactory, vtkAbstractArray, vtkParametricTorus, vtkMath and more.

vtkIO
As the name suggest, this target is responsible for data input and output. vtkIO is dependent on many other targets, amongst those, vtkjpeg and vtkpng which are needed to handle reading and writing of jpg and png files.

vtkFiltering/
  vtkGraphics
These targets contain classes representing graphical data and transformation/filtering of graphical data. vtkGraphics is dependent on vtkFiltering which holds basic graphical objects such as vtkVertex, vtkQuad, vtkPolyData, vtkSpline, and classes for graphics transformation and filtering. vtkGraphics include more complex graphical objects and classes responsible for transforming and filtering, e.g. vtkQuadraticClustering and vtkRibbonFilter.

vtkRendering
This is the target containing classes responsible for rendering graphics to a device. It’s an integral part of the thesis, and will be studied thoroughly in coming chapters.
6.2 vtkRendering Target Dependency Hierarchy

Figure 10 illustrates the target dependency hierarchy of the vtkRendering target. This implies that all dependent targets must compile and run for vtkRendering to compile. Some targets are able to compile and run out of the box, while some others need modification and/or major changes. This is described thoroughly in the coming chapter.

---

Figure 10: Target dependency hierarchy for vtkRendering
6.3 Modifying Targets
This chapter describes how VTK was modified to provide code that compiles and runs on iOS without the actual implementation of the rendering. Only vtkRendering and its library dependencies are taken into consideration. This leaves some higher level parts of VTK out, such as vtkVolumeRendering, vtkHybrid and vtkWidgets which all depend on vtkRendering.

The main approach to compiling and running vtkRendering was done through trial and error. Since little was known about VTK and what is potentially compatible on iOS and not, no effort was placed modifying the build process in CMake. Instead, a Mac OS build was produced with CMake as a starting point and the project was modified thereafter. This of course breaks the cross platform support that CMake provides, though our primary focus is to investigate how to modify VTK for iOS support, which is a prerequisite for extending the build process.

Vtkftgl and vtkFreetype
Vtkftgl has some minor OpenGL implementations of font rendering. To maintain font rendering support this target needs reimplementation in GLES 2.0. Vtkftgl is dependent on vtkFreetype [11], a wrapper around freetype, which is a free software font engine written in C. There is no iOS binary available of freetype, but with the source being written in C and free to download it is possible to build an iOS compatible binary [12]. The vtkFreetype target needs only to be linked with an iOS compiled freetype binary.

VtkGraphics, vtkCommon, vtkFiltering and vtkVerdict
The vtkGraphics target, along with its dependencies (vtkCommon, vtkFiltering, vtkVerdict) was successfully compiled and run on an iPad only by configuring the build settings for iOS.

VtkImaging
VtkImaging is a library in VTK providing some basic image processing functionality such as high/low-pass filtering, calculating image gradient, laplacian, and many other standard image processing tasks. This target can also produce a working iOS binary only by changing the build settings.

VtkIIO
VtkIIO is at its current state incompatible with iOS. It depends on a command line tool, which is executed during compilation to generate files needed by vtkIIO. Command line tools are not supported in iOS, hence compile errors are produced while trying to build the target. The command line tool that vtkIIO depends on is responsible for creating GLSL [13] shader files that are written to a material library folder. This dependency was removed since support of built in shaders is not a priority. In order to support the creation of shaders during compilation, either the code that creates the shaders needs be changed to not be a command
line tool; alternatively shaders for iOS devices can be generated and prepackaged with VTK.

VtkIO also depends on the target vtktiff, which in turn depends on an additional command line tool target. This target is responsible for initializing and writing fax decoder tables [14] and Huffman code tables [15] to files, which are included in the build and used by vtktiff. Just as the shader files, either the code needs to be changed to not be a command line tool, or the files can be created once for iOS and be prepackaged with VTK. The vtktiff target was removed as a dependency to vtkIO. This essentially removes support for file compression and decompression.

Additionally, some other classes had to be removed from vtkIO for compilation to succeed. These are classes for decoding and encoding video, such as video for windows [16], MPEG2 [17] and oggtheora [18]. Furthermore classes for interfacing with MySQL and PostgreSQL databases do not compile due to necessary header files missing. iOS support for the mentioned classes could possibly be added but is considered as out of scope for this work.

With these modifications done the vtkIO target compiles and runs on iOS.

VtkParseOGLExt

The target is a command line tool executed by the vtkIO target before compilation, which reads OpenGL extension header files and outputs VTK code that handles OpenGL extensions in a more platform-independent manner. It takes as input an OpenGL extensions header file (glext.h), a GLX extensions header file (wglext.h), a WGL extensions header file (wglext.h) [19] and writes a new header file for handling OpenGL extension (vtkgl.h). The tool is based on OpenGL and writes OpenGL specific code to the generated header. The generated file is not compatible with iOS since it, amongst other things, includes OpenGL specific header files. The tool needs to be re-written, or preferably a new tool needs to be made, that handles extension for OpenGL ES.

VtkEncodeString

VtkEncodeString, also a command line tool, is used to encode a string in a C++ file from a text file. It is used by vtkRendering to encode GLSL [13] source files. The program itself is compatible with iOS, but not its command line execution, which needs to be modified.

VtkRendering

The rendering implementations of VTK require major changes in form of new GLES 2.0 implementations to run on an iPad or any other GLES 2.0 compatible device. This showed to be a comprehensive task since the OpenGL rendering implementation in VTK makes wide use of the fixed function pipeline, which is not supported in GLES2. A new vtkRendering target was created specifically for iOS where VTK rendering classes were placed to which new GLES2 and iOS specific implementations could be added.
7 VTK Frameworks

This chapter describes relevant frameworks and design patterns used by VTK, which are necessary to have knowledge of in order to extend and/or customize the source for new platform support.

7.1 Rendering

The core of the VTK rendering framework is located within the vtkRendering target. VtkRendering consists of a set of abstract classes that make up an API with the necessary rendering elements to describe a scene. An example of this are the classes vtkRenderer, vtkRenderWindow, vtkLight, vtkCamera, vtkActor and vtkProp, that combined together in code makes up a scene which is rendered onto a device. The classes mentioned do not perform any actual rendering, since rendering depends on the underlying hardware and software. The VTK rendering classes are abstract and need to be in parts implemented or extended to perform the actual rendering. The rendering framework is independent of the underlying implementations, which is one of the design goals of VTK [2].

This section describes how a minimal rendering context can be created and how VTK objects interact to choose the rendering implementations for the desired platform. A minimal rendering context in this case is defined as the instances of necessary VTK classes needed to display a window in iOS using the VTK framework.

Creating and running a minimal rendering context in VTK is demonstrated by the following code.

```cpp
#include "vtkRenderWindow.h"
#include "vtkRenderer.h"
#include "vtkSmartPointer.h"

int main(int argc, char *argv[]) {
  vtkSmartPointer<vtkRenderWindow> window =
    vtkSmartPointer<vtkRenderWindow>::New();
  vtkSmartPointer<vtkRenderer> renderer =
    vtkSmartPointer<vtkRenderer>::New();
  window->addRenderer(renderer);
  window->render();
  return 0;
}
```

As the code suggest, first a window is creating by calling the function “New” on vtkRenderWindow, or in this case vtkSmartPointer, that delegates to vtkRenderWindow, and returns the platform specific implementation of a vtkRenderingWindow.

The next step is to create a renderer object. The renderer is added to the render window. Calling the function render on the window renders the current state of
the scene onto the window. In this case the result is a black window since no geometry, materials, lights or cameras are added to the scene.

**VtkRenderWindow**

VtkRenderWindow is an Abstract object to specify the behavior of a rendering window. A rendering window is a window in a graphical user interface where images are drawn. Figure 11 shows the inheritance diagram for vtkRenderingWindow. Polymorphism [20] is widely used in VTK to create layers of abstraction. VtkRenderWindow defines several virtual functions that are inherited and partly implemented by vtkOpenGLRenderWindow. Since OpenGL does not handle platform specific windowing systems some functions are inherited further down to the concrete platform specific implementations of window handling.

![Inheritance diagram of vtkRenderWindow](image)

**Figure 11: Inheritance and subclass diagram of vtkRenderWindow**

Most VTK objects inherit directly or indirectly from the class vtkObject. VtkObject provides some basic functions, one of which is the function “New” that takes no parameters and returns a vtkObject pointer. Inheriting classes override this function to instantiate an object of the calling class. Often the function “New” delegates the instantiation to an object factory that determines which implementation is going to be used.

![Class and collaboration diagram for instansiating a vtkRenderWindow](image)

**Figure 12: Class and collaboration diagram for instansiating a vtkRenderWindow**
Figure 12 shows how calling the “New” function creates a vtkRenderWindow object. VtkRenderWindow does not create an object but instead delegates to vtkGraphicsFactory that determines which implementation will be used given the class name as in parameter. VtkGraphicsFactory in turn delegates to vtkObjectFactory that tries to create the object. If vtkObjectFactory fails, control falls to vtkGraphicsFactory. In the case of vtkRenderWindow, vtkGraphicsFactory is responsible for instantiation. VTK extensively uses macro definitions [21] to define what libraries the build supports. To support iOS, new macro definitions have to be added to follow the design patterns. The macros are defined in the CMake configuration and are set during the CMake build process. For this project, the macros are hard coded for the sake of simplicity. The graphics factory creates an instance of the concrete implementing class vtkIOSRenderWindow, which inherits vtkGLESRenderWindow and returns an object of the type vtkRenderWindow.

From a VTK users point of view, concrete implementation classes are never handled, these are, as described, handled by the framework and the user needs only to worry about how the abstract VTK classes work to implement the rendering functionality.

**vtkRenderer**

VtkRenderer provides an abstract specification for renderers. A renderer controls the rendering process for objects. Rendering is the process of converting geometry, specification of lights and a camera view to pixels.

The render function of vtkRenderWindow loops through each renderer that exist in the window, and draws the image produced by each renderer.

The vtkRenderer instance is created analogous to vtkRenderWindow as demonstrated in the previous section. In fact, the vtkGraphicsFactory class creates all graphics objects. As said, the creation of objects must me modified to create new GLES2 and iOS specific implementations.

![Inheritance and subclass diagram of vtkRenderer](image.png)
Extending VTK factories

To make the iOS adaption of VTK usable it is necessary not to break the design pattern for creating VTK objects. In an ideal situation, a visualization application that uses VTK should work on an iPad without modification to the application itself by building against an iOS binary of VTK.

Calling the static function named “New” on a VTK class creates an instance of the desired class with the underlying implementation hidden. Of course the programmer could create an instance of the desired concrete implementation of the class, which is not the recommended usage of VTK, but that would make the application platform dependent. The call to “New” delegates to a factory class with the class name as in parameter and receives a concrete implementation of that class type casted to the callers’ type, if one can be found.

VtkObjectFactory is the base factory class that other factories sub class. In the case of graphics objects, which are our main concern, objection creation is delegated to a graphics factory. The function of the graphics factory is straightforward. Given a class name and a set of pre defined macros the factory simply uses the macros to determine what platform is being used and the class name to determine which object to create. Below is an extract from the class vtkObjectFactory to demonstrate this.

```c
#ifndef VTK_USE_COCOA
if( !vtkGraphicsFactory::GetOffScreenOnlyMode() ) {
    if( strcmp( vtkclassname, "vtkRenderWindowInteractor" ) == 0 ) {
        return vtkCocoaRenderWindowInteractor::New();
    }
}
if( strcmp( vtkclassname, "vtkRenderWindow" ) == 0 ) {
    return vtkCocoaRenderWindow::New();
}
#endif
```

At this point in the code, if we are building for Mac OS, are using Cocoa and vtkclassname is “vtkRenderWindow”, an instance of vtkCocaRenderWinow will be created and returned to vtkRenderWindow which will cast the instance to a vtkRenderWindow.

One could extend this approach, add macro definitions for iOS and GLES2 and return instances of new implementations of the necessary classes. An alternative method would be to create a new graphics factory specifically for GLES2 implementations that is delegated to from the main graphics factory, or added to the vtkRendering target in favor of the current graphics factory during the build process.
7.2 Window and View handling

A window in iOS differs from windows in supported operating system of VTK in that an iOS window has a more limited role.

Window objects in iOS are called UIWindow, compared to NSWindow on Mac OS. An iOS application can technically have more that one window, with windows layered on top of each other, but an application is by convention limited to a single window. A UIWindow always occupies the entire screen and has no title or controls and cannot be manipulated by the user [22].

This of course adds the limitations of not being able to manipulate windows (resizing) and only showing one window at a time while using VTK on iOS. Although the goal is to make the usage of VTK on iOS transparent for the user these limitations must be taken into consideration when creating applications.

The role of a UIWindow is to display the application’s visible content, it delivers touch events to the views and other object and handles orientation changes. A UIWindow can be created using xib files (A file format defining user interfaces for OS X and iOS) or programmatically during applications startup.

A UIWindow can hold one or many views. A view is responsible for drawing content into its rectangular area and handling touch events. Views in iOS are of the type UIView and can be subclassed to create custom views. For VTK a GLES 2.0 view is needed for rendering.

UIWindow

A UIWindow naturally maps to vtkRenderWindow. The vtkRenderWindow is responsible for setting up and manipulating windows, so is UIWindow.

To implement our own iOS window for VTK we need to subclass vtkRenderWindow. VtkRenderWindow has a number of virtual functions that are left to the implementing classes to handle, such as the virtual functions “Start()”, “Render()” and “Finalize()”. “Start()” initializes the rendering process, “Render()” asks each renderer owned by the render window to render its image, and “Finalize()” ends the rendering process.

An iOS application is launched in the main function using the following function call [23]:

```c
int UIApplicationMain(  
    int argc,  
    char *argv[],  
    NSString *principalClassName,  
    NSString *delegateClassName  
):
```

The call creates the application object and the application delegate and set up the event cycle. The arguments argc and argv are usually the same parameters as passed to the main function. PrincipalClassName is the name of the UIApplication class or subclass. This is usually set to nil, which assumes
hints about what each implemented function to sub framework is supposed to work and a great deal of trial and error. One approach is rendering, but to investigate how it could be done. This work does not concentrate on creating the actual implementations of GLES2 and most time-consuming concerns in making VTK iOS compatible is changing the rendering implementation from OpenGL to GLES 2.0.

Since VTK has as a design goal to be platform independent and rendering library independent the framework is created independent of OpenGL, although VTK only supports OpenGL rendering. This practically means that we can create new classes that implement the abstract VTK rendering classes as we see fit and create new factories that override the old ones. We could completely disregard the current rendering implementations and start implementing GLES2 code from scratch.

This work does not concentrate on creating the actual implementations of GLES2 rendering, but to investigate how it could be done.

One approach is, as mentioned, to throw away the OpenGL implementations and start from scratch. This will require extensive knowledge how the rendering framework is supposed to work and a great deal of trial and error. We only have to subclass the VTK rendering classes and rely on VTK documentation to give us hints about what each implemented function should do.
Another approach that might be faster and require less VTK framework knowledge is to create the implementing classes and try to copy as much of the current OpenGL code as possible. This is possible since GLES 2.0 is defined as a subset of OpenGL 2. The first step would be to identify the differences between OpenGL 2 and GLES 2.0. The API documentation for each library gives us a list of functions that are available.

The design difference between OpenGL 2 and GLES2 is in principle that, whatever is not supported in GLES2 can be done with shaders. Upon identifying code that uses the unsupported fixed function pipeline, that code can be removed and replaced by the loading the corresponding fragment and vertex shaders.

As an example the class vtkOpenGLLight is responsible for specifying light sources in a scene. It implements a single function Render() that specifies each light source by calling the OpenGL function glLightfv. The function call is not supported by GLES 2.0 and therefore needs to be replaced with shaders. Instead of making the function call we can store the lights specifications as vertex arrays and load the vertex and fragment shaders for the desired lighting effect.

The same approach can be applied to texturing, geometry transformation and other unsupported OpenGL functions. Instead of calling the function, we store the necessary shading information in vertex arrays and load the desired shader programs before rendering our geometry.

**Shaders**

Although VTK mostly uses the fixed function pipeline to perform its rendering, shaders using GLSL are supported. VTK provides a framework for handling shaders, such as compiling, loading and binding shader programs and applying different shaders to different vtkActors.

Shaders in VTK are encapsulated in a material [24]. A material is described in an xml file. The material description includes:

- Which vertex and fragment shaders to use
- Shader language
- Entry routine of the shader
- Uniform variables passed to the shaders
- Surface properties such as ambient and specular color

A material is applied to an actor by calling vtkProperty::LoadMaterial(const char*) on the actors vtkProperty object instance. When a material is loaded, the vtkProperty is updated with the data in the material file and the appropriate subclass of vtkShaderProgram is instantiated. VtkShaderProgram compiles and binds a shader every time an actor is rendered [25].

To go from the fixed function pipeline to a completely shader based rendering in VTK, we first need to create a library of GLSL shaders and define xml material files.
In VTK materials are optionally added to vtkProperty. If GLES2 is used as the concrete rendering implementation, this can no longer be optional. Materials must always be added to vtkProperty objects in order to use shaders. Once the fixed function rendering is removed in favor of shaders, a standard set of materials applied as default can replace the functionality of the current rendering implementation, practically making it optional to use materials.

7.4 Interaction

Another aspect that differentiates the iPad from current VTK supported platforms is the interaction capabilities. Instead of using a traditional mouse to transform camera and geometry one uses the multi touch support of the iPad to perform gestures. This section gives a brief introduction on the interaction capabilities of the VTK framework and some insights on multi touch gestures on VTK and iPad.

VTK interaction framework

VTK has a couple of abstract classes that handle interaction and events, mainly vtkRenderWindowInteractor, vtkInteractorStyle and vtkInteractorObserver.

VtkRenderWindowInteractor provides a platform-independent interaction mechanism for mouse, key and timer events. It is a base class for platform dependent implementations. A subclass of vtkRenderWindowInteractor listens for platform events, translates and routes event messages to vtkInteractorStyle.

VtkInteractorObserver is an abstract superclass for subclasses that observe events invoked by vtkRenderWindowInteractor.

VtkInteractorStyle is a subclass of vtkInteractorObserver. VtkRenderWindowInteractor forwards events to vtkInteractorStyle that provides an event driven approach to the rendering window.

Interaction in iOS

Event handling in iOS [26] is available in the UIKit library. To handle multitouch events, a subclass of a responder class must be created, which can be a subclass of a UIView. To receive and handle multitouch events one or several of the following methods must be implemented in the views controller.

```
(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
(void)touchesMoved:(NSSet *) touches withEvent:(UIEvent *)event;
(void)touchesEnded:(NSSet *) touches withEvent:(UIEvent *)event;
(void)touchesCancelled:(NSSet *) touches withEvent:(UIEvent *)event;
```

The events are handled by evaluating the properties of the set of touches and proceeding accordingly. There is also the possibility to use gesture recognizers, which simplifies the handling of gestures. In that case, one must subclass the UIGestureRecognizer class. Instead of receiving raw touch events one receives gesture events that are commonly used in iOS. Gestures that can be recognized
are, tapping, pinching in and out (for zooming), panning or dragging, swiping, rotating and long press. Gesture recognizers are attached to a view and deliver touch objects to the view after a gesture has been recognized. When a gesture or touch event has been received it is up to the programmer to transform the graphics accordingly.

iOS supports additional events such as, shaking the device, and GPS events. These events are delivered similarly to touch events.

**Short on iOS multitouch in VTK**

Adding iOS multitouch support to VTK requires new implementations of vtkRenderWindowInteractor and vtkInteractorStyle. The role vtkRenderWindowInteractor subclass would be to intercept touch events or gesture events from iOS and also handle timing events and the event thread in a platform dependent manner. The events sent to vtkInteractorStyle must be extended to support platform independent descriptions of touch events.
8 Conclusions

From the investigation and work done we can conclude that VTK for the most part can be modified and extended to run on iPad thanks to the design of VTK.

To add GLES 2.0 and iOS support in a way that makes sense, the CMake build process needs major refactoring, so that the user can download the VTK source and produce an XCode project ready for iOS integration. Ideally the GLES2 and iOS part of VTK should be integrated into the VTK project be and developed alongside “standard” VTK. As discussed in chapter 5.1, the CMake build files require modifications to handle new platforms. The CMakeList.txt file for the each target needs new macros defining GLES and iOS specific building options to add the correct files to the target during the build step. We also need to configure CMake to link the targets to required iOS libraries and set iOS compatible build options.

The issues encountered while trying to compile and run VTK, described in chapter 6 “Compiling and Running VTK on iOS”, must be addressed. One issue was generation of required OpenGL extension files, which simply is incompatible with GLES 2.0. A proposed solution is to generate VTK extension files for each supported hardware platform, package the file with the build and use CMake to link the correct file to the build depending on configuration. Another issue is the generation of material files. This can also be resolved by removing the command line tool that is executed before compilation and create a library of material files that are included in the build.

Parts of the vtkIO target and dependent targets could not be compiled due to missing libraries for iOS. Although this is a small delimited part with no affect on this work, further investigation is needed to determine what can be supported and not. This concerns support for video encoding and decoding, compression, and database interfacing.

The largest part of making VTK iOS compatible is of course the OpenGL rendering implementation, which requires a lot of additional work. While spending a lot of time looking at the rendering implementation of VTK and studying the VTK frameworks, it was concluded that the fixed function pipeline of OpenGL is widely used, with no trivial way of porting the OpenGL code to GLES 2.0. However, GLES 2.0 is defined as a subset of OpenGL 2, which entails that a lot of compatible code can be reused while use of the unsupported fix function pipeline has to be replaced by shaders using the material framework provided by VTK, described in chapter 7.3 “Porting OpenGL to OpenGL ES 2.0”.

To provide iOS specific window handling is relatively problem free since the VTK framework is created for cross-platform support. New iOS implementations of vtkRenderWindow and vtkRenderer are needed to interface with the UIWindow and UIView APIs of iOS.

Interaction in iOS differs a great deal from mouse interaction. VTK provides us with an interaction framework where a vtkRenderWindowInteractor is attached to a vtkRenderWindow to enable manipulation of the scene. The framework
offers an abstract way of handling interactions and makes almost no assumptions regarding the underlying physical form of interaction. New interaction implementations can be added by inheriting the classes described in 7.4 “Interaction”. The iOS SDK supplies comprehensive support for event handling and gesture recognition, which helps in implementing multitouch support for iOS. Though a desirable approach might be to create an abstract multitouch interaction layer on top of VTK's current interaction framework to facilitate a common way for different devices to implement their own multitouch logic.

8.1 Why not OpenGL ES 1.1?
One could ask why GLES 1.1 wasn't investigated instead of GLES ES 2.0 since GLES 1.1 supports the fixed function pipeline.

The main issue with GLES 1.1 is that it does not support a programmable pipeline. Since rendering in VTK is based on OpenGL 2 and shaders are supported in VTK, using GLES 1.1 would remove support of shaders and materials in VTK.

Another concern is the fact that OpenGL moves in the direction of removing the fixed function pipeline in favor of shaders. In GLES 2.0 the fixed function pipeline is not supported, which is also the case for OpenGL 3.1 and newer versions.

8.2 Future work
There remains a major amount of work to be done to get a working prototype of VTK running on iOS. Below is a summary of the work left to be done for VTK to be fully iOS compatible, some of which are investigated in this work, and others not.

- The CMake build process needs revision to be able to produce a working iOS project
- The OpenGL rendering classes have to be re-implemented in GLES 2.0 and added to the build process
- Any required files such as xml materials, decoder tables etc must be created and come packaged with the build, alternative generated in a way that is compatible with iOS execution.
- Consideration must be taken into how user interaction using touch and multitouch devices should be handled and new implementing classes are needed.
- Scripting languages that wraps VTK, such as Tcl/Tk and python should be investigated to determine the possibility of iOS support.
- Higher level libraries which where not taken into consideration must be further investigated
  - Volume rendering in VTK will require additional GLES 2 re-implementations.
  - The vtkWidget library needs revision since all widgets might not be suitable for the platform.
References


