DEVELOPMENT OF THE OPERATING SYSTEM USER INTERFACE OF A SMARTPHONE USING THE OpenGL ES API

ΑΝΑΠΤΥΞΗ ΛΕΙΤΟΥΡΓΙΚΟΥ ΣΥΣΤΗΜΑΤΟΣ ΕΝΟΣ SMARTPHONE ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ OpelGL ES API

ΝΤΟΥΖΟΣ ΓΕΩΡΓΙΟΣ ***Α.Μ:2005030019***

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: ΠΝΕΥΜΑΤΙΚΑΤΟΣ ΔΙΟΝΥΣΙΟΣ

Prologue

The compilation of this diploma thesis started at the beginning of the current year and finished a few weeks before the presentation date. The purpose of this thesis is the implementation of a user interface of a smartphone using the OpenGL ES 2.0 API. Prior to this project I had no experience or knowledge in the field of graphics and OpenGL, thus this diploma thesis was a stepping stone for me in the world of graphics. The selected theme is a thesis proposal from Imagination Technologies Ltd (IMG), with the collaboration of which I carried it out. After contacting IMG in order to ask advice over my diploma thesis subject, I was pleasantly surprised as I received an e-mail with proposals and an encouragement to download the company's SDK which provided full documentation and guidance for the proposed subjects. After agreeing with my supervisor over the matter, I decided to undertake the "DEVELOPMENT OF THE OPERATING SYSTEM USER INTERFACE OF A SMARTPHONE USING THE OpenGL ES API". The thesis proposal was addressed to a group of students, thus it had to be limited, but at the same time extra care should be given so that it would not lose its goals.

The target of this project is to present a novice's point of view while trying to learn all concepts in 3D graphics, alone with implementing a functional interface for a smartphone. That does not mean that the target was the implementing of all features of a fully functional device, but more the introduction from the very basics of OpenGL ES until addressing issues a developer can come across during a similar project and presenting several techniques and thoughts that the author used to face the problems that came up. Besides the graphics part, one could find highly interesting the industry connected part of the thesis, as the SDK and the technologies described are widely used currently in the market, in many high sale, successful devices. This of course may not be widely known as we usually tend to focus on the brand names which are on the cover of the products we use instead of knowing what is hidden under the hood.

Thesis Outline

The project commences with an introduction that deals with OpenGL ES. Though it does not go though many details, this part provides adequate information so that the rest of the report can be easily read through, even from an amateur in the field of graphics reader. The introduction deals with fundamental terms of the OpenGL ES 2.0 world like the programmable pipeline, buffers, EGL, shaders, shader language and textures.

In chapter two follows a presentation of IMG's innovation technology which includes a GPU architecture as well as the SDK that I used and proved to be very helpful. In this part one can find the SDK's framework, on which this diploma thesis was build, an introduction to the hardware design that IMG has launched in the market, and the popular texture compression of the company that is widely used from top world-wide companies in the field of portable devices.

Further analysis of the SDK and the tools included in, follows in the presentation of the project. Before this part deepens into the code of the project, it begins with an overview of the project and a description of the GUI of the interface. Moreover, one can find details about the code of each section of the interface as well as the showcase of different techniques that were used in the implementation, along with issues that came up and the suggested solutions. Special attention is given at performance issues throughout the report, both in the code analysis as well as with a special section providing performance tips.

The report concludes with a sum up of the project, highlighting what it could be done differently, acknowledging that the handling of some issues may not be ideal due to the inexperience of the developer, and by referring to future work that can be done in order to further expand the existing one. At the end of the project lies an appendix with few details that could possible raise questions after a closer look at the following code.

Acknowledgments

During the months I worked in this project, or even before to start it, I was lucky enough to meet people that supported and helped me through this effort. I would like to thank my supervisor Professor Dionisis Pnevmatikatos for the guidance he provided and the excellent cooperation we had during this project. Also I would like to thank him for taking me into account and giving me the opportunity to get this thesis at a time when my obligations towards the school were such to create doubts whether or not I would be able to carry this task out. I would like as well to thank the other members of the committee, Professor Aikaterini Mania and Professor Ioannis Papaefstathiou, who approved this diploma thesis and contributed with useful remarks during my presentation.

I would specially like to thank the persons that made this collaboration with IMG become possible, my childhood friend Thanos Grassos who led and urged me towards this opportunity and also his manager Jamie Broome, whose help and interest was more than anyone would expect. I would also like to mention Giorgos Koulieris, from Technical University of Crete, who provided me with useful tips throughout my search. Finally I want to thank my friend Marina who put up with me and helped me out in coming through this period, when multiple and difficult tasks were assigned to me.

Chapter 1

1.1: An introduction to OpenGL ES

OpenGL (Open Graphics Library) is an application programming interface (API) for rendering 2D and 3D graphics. OpenGL ES is a subset of OpenGL, designed for embedded systems such as cell phones, PDAs, consoles and vehicles' systems. While OpenGL was originally created by Silicon Graphics in 1992, OpenGL ES was created and managed by Khronos Group, a consortium focused on developing free APIs.

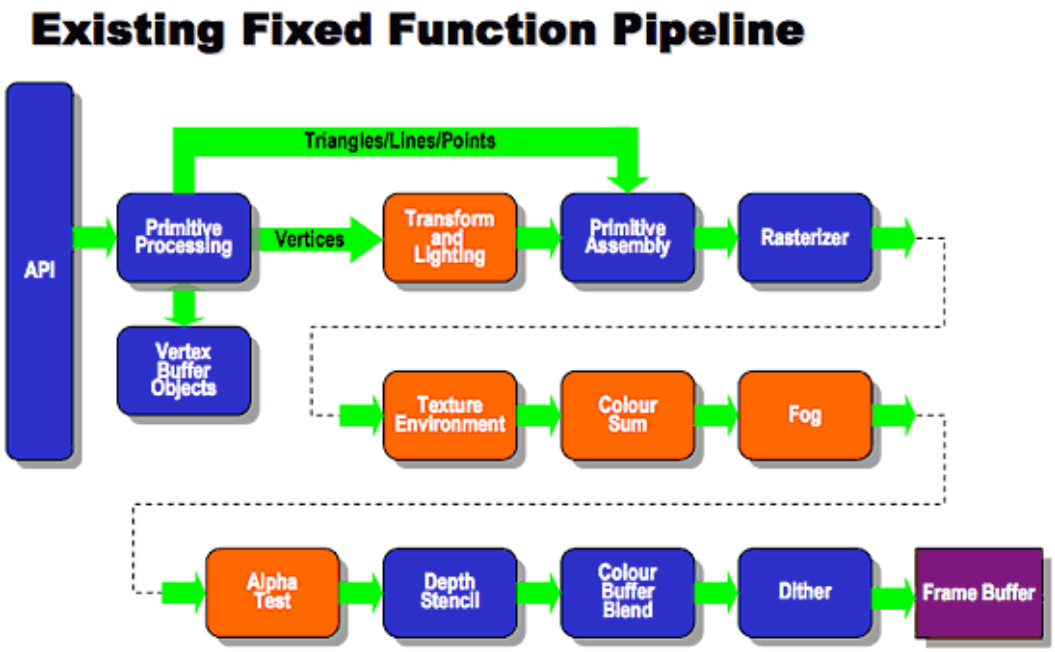
OpenGL is the closest point of interaction between the CPU and the GPU. What makes OpenGL truly admirable is that, not only is a cross-platform 3D API working on Linux, Unix, Mac OS and Microsoft Windows, but it is also "Language Free". This means that we can use C, C++, Objective C, Java, Perl or almost any language we want while the API will present the same behavior, functions and commands.

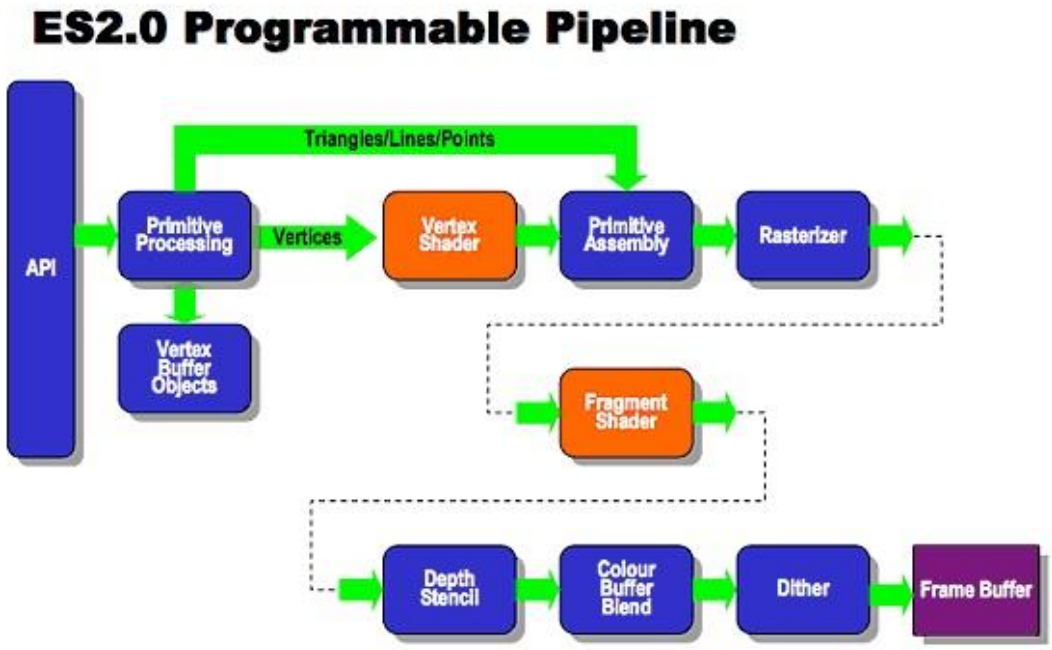
OpenGL ES was designed under certain standards and specifications. Because the role of OpenGL ES was to be suitable for constraint devices, any redundancy from OpenGL was removed and only the most useful methods were used while compatibility with OpenGL was kept. Furthermore, the OpenGL ES designers aimed to reduce power consumption needed to use the API and they kept certain agreed-on standards for image quality, appropriate for handheld devices.

In 2007 OpenGL ES 2.0 was launched with a great change, implementing a programmable graphics pipeline succeeding the older fixed pipeline.

To make things clear, the "fixed pipeline" refers to the older generation pipeline used in GPUs that was not really controllable; the exact method in which the geometry was transformed, and how fragments (pixels) acquired depth and color values were built-in to the hardware and couldn't change.

Modern GPUs have a programmable pipeline; those previously rigid, possibly burned into the chip stages of the pipeline (transformation, shading) have been replaced with stages that can be controlled by bits of user-supplied code called *"shaders"*. This is a far more flexible approach that has enabled a wide variety of graphic effects that were not previously possible without, for example, hardware support and extensions specifically dedicated to enable them. The older, fixed-function pipeline has no real advantages over this one, and so it has phased out.





1

[1] http://www.khronos.org/opengles/2_X/

Before trying to explain further each step of the pipeline, it would be preferable to introduce the three basic concepts around of which OpenGL works. These would be primitives, buffers and rasterize. OpenGL s primitives are three kinds of objects: 3D points, 3D lines (composed by two points) and 3D triangles (composed by three points). Buffer is a temporary optimized storage and rasterize is the process by which OpenGL takes all information about 3D objects to create a 2D image which will be presented on the device's screen.

Vertex Shader:

The vertex shader is a little programmable method for operating on vertices. Vertex shader defines the final position of a vertex. It can be used for operations such as transforming the position by a matrix, computing the lighting equation to generate a per-vertex color, and generating or transforming texture coordinates.

The inputs to a vertex shader are the attributes, which are used to construct the vertices of your object, and the uniforms, which are constant data used by the shader. The outputs are varying variables which are passed to the fragment shader.

Primitive assembly:

Each vertex transformed by the vertex shader includes its position and other information such as its color or texture coordinates. In the primitive assembly stage, OpenGL determines the setup of the primitives. That means that vertices will be clipped inside the viewing frustum and those which are outside or they are at the backside - not visible part - of our frustum will be culled.

Rasterization:

As mentioned before, rasterization is the task of taking primitives and converting them from a 3D scene it into a raster image (pixels), a set of two-dimensional fragments, which are processed by the fragment shader for output on the screen.

Fragment shader:

Fragment shader is a programmable method operating on each visible fragment of the final image. In the fragment shader we can work everything related to the mesh' surface like colors, materials, texture, shadows and light effects.

The inputs to a fragment shader are the varying variables, outputs of the vertex shader that are generated by the rasterization unit for each fragment, uniforms, constant data used by the fragment shader and samplers, a specific type of uniforms that represent textures.

Per fragment operations:

A fragment is a candidate to become a pixel in a buffer, called frame buffer. For every fragment, OpenGL applies a series of tests to check if the fragment meets certain specifications in order to eliminate the fragment early and as a result to avoid updating the frame buffer (an energy devouring procedure). If these specifications are met, OpenGL processes the fragment's final calculations. These are the pixel-ownership test, scissor test, multi-sample operations, stencil test, depth test, blending, logic operations and dithering.

1.2: Buffers

In OpenGL there are three kinds of buffers. Frame buffers, render buffers and buffer objects.

Frame buffers are used to store rendering results. They look like a collection of images (like 3D objects, depth of objects and intersection of objects). By capturing images that would normally be drawn to the screen, we can use them to implement a large variety of image filters, and post-processing effects. They are used for their efficiency and ease of use.

Render buffer objects are a temporary storage of a single image. So we can understand that a frame buffer is a collection of render buffers. The render buffer can be used to allocate and store color, depth, or stencil values and can be used as a color, depth, or stencil attachment in a frame buffer object.

Buffer objects are used to hold information about our 3D objects in an optimized format. This information can be about structures or indices. Vertex Buffer Objects (VBOs) hold structures that can be an array of vertices that describe a 3D object, an array of coordinates or normals. Index Buffer Objects (IBOs) hold an array of indices which is used to indicate how the faces of our mesh will be constructed based on an array or structure. The advantage of a buffer object is that they work directly at the GPU processing and we don't need to hold the arrays after we create a buffer object.

1.3: EGL

OpenGL ES is not responsible for managing the windowing system of each device that supports it. Khronos group, in order to create a bridge between the rendered output of OpenGL and the device's screen, created the EGL API.

EGL is responsible for managing the windowing system of our device, creating drawing surfaces, synchronizing OpenGL ES 2.0 and other graphics rendering APIs (like the drawing commands of our windowing system) and managing rendering resources.

Before starting to use OpenGL ES we have to setup the EGL in order to know about our windowing system, to initialize a context about our OpenGL application and then present our render's output on the screen.

EGL works with 2 internal frame buffers. One presents the desired image on our screen while the other waits for a new render output. At the next EGL call, the buffers swap their positions. This technique provides great performance for our

system because the final surface is notified after we have finished our rendering. In addition to that, the buffer on the back can receive our commands faster that the device's screen.

It is also worth mentioning that we can display the frame buffer's result straight on our screen without the use of a windowing system, a technique known as null-windowing system. This of course removes the abstraction layer that a windowing system offers and thus we can't project more than one window on the display.

1.4: GLSL ES

Since we talked about shaders, we should also talk about the OpenGL ES shader language (GLSL). The shader language is quite similar to C language. It has the same syntax, variable declarations, control flow statements, loops even macros. GLSL is designed to be as fast as possible working directly in the GPU, thus we should be careful with the use of loops and conditions since they limit the performance of our program.

A major difference between the two languages is the native data types which are supported by GLSL ES. In computer graphics, there are two fundamental data types that form the basis of transformations: vectors (vec i where i = 2, 3, 4 and defines the number of components) and matrices (mat i). These two data types are central to the OpenGL ES Shading Language as well. Furthermore, in GLSL we have a variety of built-in functions which can be proved to be very useful. We will see the use of some of them later. It is also important to know the precisions qualifiers which GLSL have. Precision Qualifiers set the data range of any variable. (lowp, mediump, highp)

The great advantage of GLSL working directly to the GPU is the floating point operations. We can do multiplications between vectors and matrices (respecting their dimensions) in just a single line! Finally it is important to remember that GLSL is an inline language. This means, for example, that if we call a function before writing it, the call will fail.

1.5: Shaders and programs

There are two fundamental object types you need to create to render with shaders: *shader objects* and *program objects*.

The shader object is an object that contains a single shader. The source code is given to the shader object and then the shader object is compiled into object form (like an .obj file). The shaders will be processed in the GPU and so, in order to optimize the process, OpenGL ES compiles the source code in a binary format. After compilation, the shader object can then be attached to a program object.

A program object gets two shader objects attached to it. In OpenGL ES, each program object will need to have one vertex shader object and one fragment shader object attached to it (no more, and no less). The program object is then linked into a final "executable." The final program object can then be used to render.

The general process for getting to a linked shader object is to first create a vertex shader object and a fragment shader object, attach source code to each of them, and compile them. Then, you create a program object, attach the compiled shader

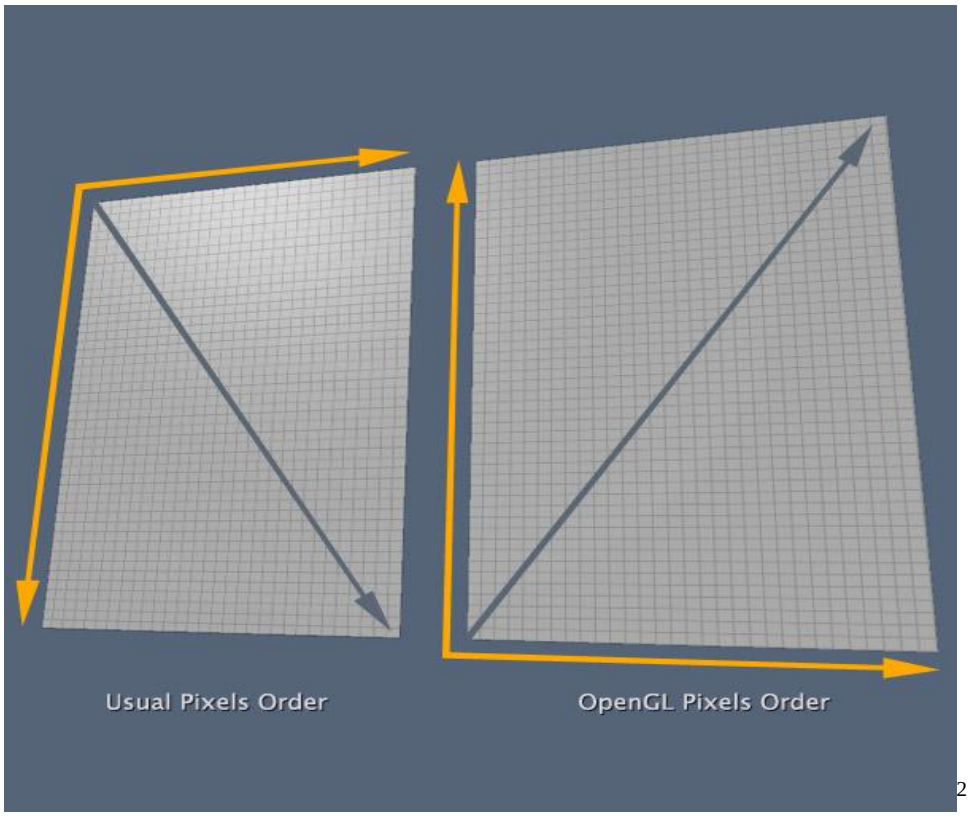
objects to it, and link it. If there are no errors, you can then tell the GL to use the program for drawing.

1.6: TEXTURES

Textures is a very large topic in OpenGL ES. In this chapter we will just make a little introduction in textures, mainly in 2D textures, that will help us understand better the concepts of the project.

Texture is one of the most fundamental operations used in graphics. Textures allow us to display more details on a mesh that is not possible to display just with geometry. There are 2D textures and cube map (3D) textures. 2D textures are actually a two-dimensional array of image data. A texture pixel is also known as a texel.

When rendering with a 2D texture, a texture coordinate is used as an index into the texture image. Each vertex has a texture coordinate. Texture coordinates for 2D textures are given by a 2D pair of coordinates (*s*, *t*), also called (*u*, *v*) coordinates. These coordinates represent normalized coordinates. Unlike the usual image file formats (jpg, png, bmp, gif) which store the pixel information starting at the upper right corner and moves through line by line to the lower left corner, textures in OpenGL reads the pixels starting from the lower left corner moving to the upper right corner. The lower left corner of the texture image is specified by the *u*, *v* coordinates (0.0, 0.0) and the upper right corner u, v coordinates (1.0, 1.0). In order to solve this issue, we can make a vertical flip to our image before loading it to the OpenGL core.



[2]

[2] http://db-in.com/blog/2011/02/all-about-opengl-es-2-x-part-23/

Another important thing about textures in OpenGL ES is that all textures is strongly advised to have power of two dimension values. That means that the size of a texture can be 32x32 or 256x256 but not 300x300.

Texture mipmap:

When trying to magnify a texture, it is quite possible that aliasing artifacts will be produced in our image. While an object becomes smaller in our screen, pixels quickly switch from one color to another and as a result we get a shimmering image. The solution to resolve this problem is called mipmapping.

The idea behind mipmapping is to build a chain of images known as a mipmap chain. The mipmap chain begins with the original image and then continues with each subsequent image being half as large in each dimension as the one before it. This chain continues until we reach a single 1x1 texture at the bottom of the chain. The mip levels can be generated programmatically, typically by computing each pixel in a mip level as an average of the four pixels at the same location in the mip level above it (box filtering).

Chapter 2

2.1: IMAGINATION TECHNOLOGIES Ltd

As mentioned in the beginning of this project, OpenGL ES 2.0 is a royalty free, cross platform API. As a result many world renowned companies implement projects using OpenGL ES such as NVIDIA, Intel, Apple, Nokia, Panasonic, Google, NexusChipsCo, AMD and Imagination Technologies Ltd.

Imagination Technologies (IMG) is an international company, playing a leading role in multimedia, communication and of course graphics/technologies markets. The company is listed on the London Stock Exchange and is a constituent of the FTSE 250 Index. Among other important releases, IMG has developed PowerVR, a 3D engine used widely in mobile devices, while recently acquired the processing technology firm MIPS.

2.2: POWER VR SERIES 5 ARCHITECTURE

PowerVR is a family of graphics IP cores from Imagination Technologies that uses the “Tile Based Rendering Technique” (TBDR) implemented solely by IMG in order to reduce the system memory bandwidth required by the GPU. The reduction of transfer data between the system memory and the GPU combined with reductions in system memory bandwidth and hardware optimizations can boost performance while allowing the GPU working at lower power.

The most notable difference between the traditional Immediate Mode Renderer (IMR) and TBDR is that the first one renders all objects within the screen’s

boundaries whereas the second determines up in front what is visible or not, allowing the hardware to render only what is necessary.

2.2.1: The Tile Based Deferred Rendering

What is tiling:

Tiling is a technique that allows rendering subsections of an image at each time instant instead of rendering the whole image. The main benefit is the use of the fast on-chip memory for rendering color, depth and stencil buffers and as a result the reduction of system memory bandwidth. The rendering comes in two phases. The first one takes over the geometry processing, determining which geometry has fallen in the bounds of a tile and the second one takes over the rasterization.

What is deferred rendering:

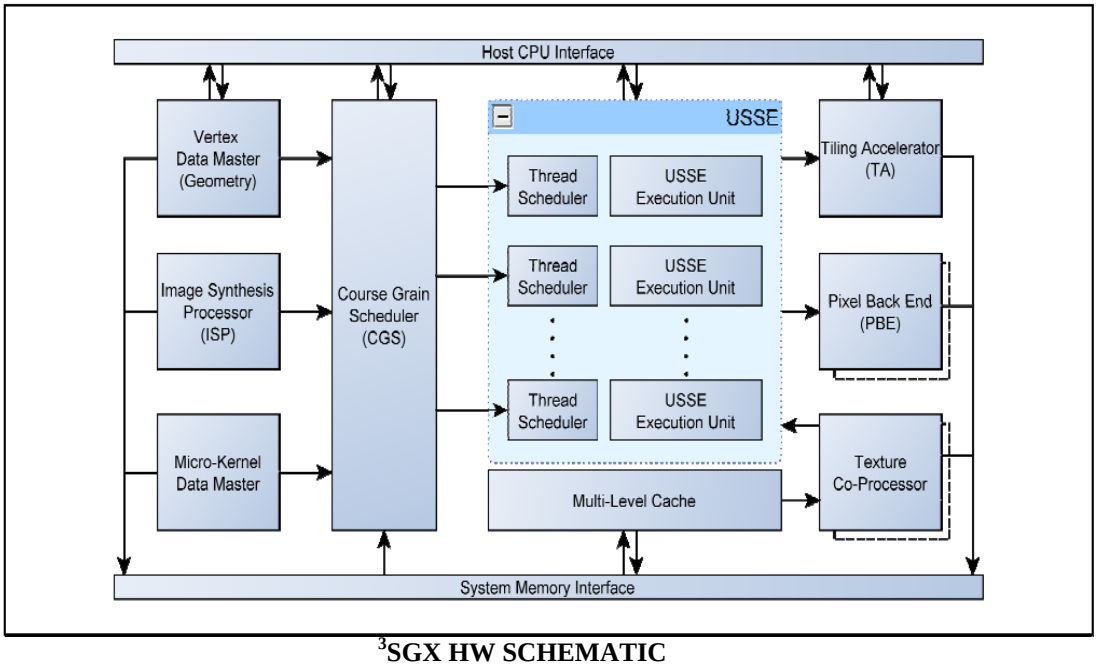
Deferred rendering splits the per-tile rendering into two phases, hidden surface removal (HSR) and shading. HSR will be performed in every tile, completely removing all fragments that will not contribute to the final image while the shading follows determining the meshes surface.

TBDR VS IMR:

IMR relies on Z-buffer to sort the end results after all objects have been rendered on screen. Although some progress have been made in order to increase the performance of IMR (such as early Z-buffer operations), the brute force approach of the technique taking the object through the entire pipeline does not leave any real margins in performance boost comparing with TBDR.

2.2.2: SGX overview

Taking a quick look over the PowerVR architecture is enough to understand its unique structure comparing with other market GPU architectures. Before continuing in describing the steps in which the data go through, we should make a special reference to the component that makes the real difference in the architecture. That would be the USSE (Universal Scalable Shader Engine). To cut a long story short, the USSE is a processor standing on top of our unified GPU architecture. The USSE is a multi-threading processor capable of processing vertex, fragment and GPU data. As vertex and fragment processing is completely decoupled – all geometry processing is done, then rasterization begins – the USSE thread scheduler can actually load balance in a queue of tasks and as a result, any idle time between processes or latency, bares to a minimum.



[3]**SGX HW SCHEMATIC**

Initially, the geometry data are submitted from the system's memory to the Vertex Master Data. This component takes the stream of data and sends them to CGS (Course Grain Scheduler) which in turn breaks down the job into smaller tasks. Thereafter, the CGS sends the data to the scheduler of any available USSE which processes them and carries out the per vertex operations before to pass it on to TA (Tiling Accelerator). The TA goes through the clipping, culling and updating of the display lists, which contain all information about the objects that will finally appear on screen. Data coming out of the TA are stored in memory of the PB (Parameter Buffer).

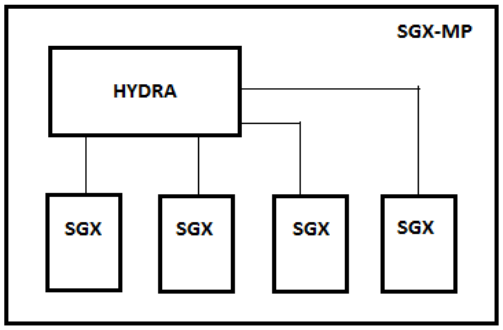
After geometry is processed, it is time about per-fragment operations. Rasterization, texturing and shading begins at each tile separately. The tile data pass to ISP (Image Synthesis Processor) where Hidden Surface Removal takes place. The resultant values are taken to the Texture co-Processor for pre-fetching and they are also distributed to the thread schedulers of the available USSEs. The USSEs process per-tile texture in cooperation with the Texture co-Processor and the final stream of data are send to PBE (Pixel Back End) for final operations before the rendered image is finally flushed to the frame buffer. In order to relieve the CPU from interrupts, the architecture also contains a micro kernel to handle these events.

2.2.3: SGX-MP

SGX-MP architecture is designed to enable many SGX cores to operate in parallel, scaling performance almost linearly. In real world, each additional core runs at 95% of the efficiency of a single one. That is possible thanks to a piece of logic called MP-WRAPPER (or Hydra) that allows the SGX cores to communicate and work together. Hydra splits objects into chunks which are evenly distributed between

[3] IMG SDK, PowerVR Series 5, Architecture Guide for Developers.

the cores. As a result the idle time in each core is kept to a minimum and hotspots that would be created, if one core for example had a lot more work than the others, are diminished.



What it is worth mentioning, is that for every core we add to the architecture memory bandwidth is just slightly increased. Another important feature is that all devices that operate under the SGX core can also operate under the SGX-MP core without any further modification.

2.3: SDK PRESENTATION

PVR Shell

PVRShell is a light-weight framework used for the setup of applications, shutdown and event handling. It is designed to stream line the process of writing cross-platform applications, making programming for PowerVR platforms easier and more portable, and it is used in many rendering engines that use PowerVR architecture GPUs.

It consists of a C++ class which takes care of all API and OS initialisation for the user and handles adapters, devices, screen/windows modes, resolution, buffering, depth-buffer, viewport creation & clearing, etc.

PVRShellOS.cpp and PVRShellAPI.cpp contain all the code to initialize the specific OS and API (Windows and OpenGL ES in our case). So we can easily understand that this code varies depending on our implementation preferences. The code that is always the same is the *PVRShell.cpp* and it interacts with the user though an abstraction layer.

A new application must link to these three files and must create a class that will inherit the *PVRShell* class. This class will provide five virtual functions to interface with the user:

InitApplication(),
InitView(),
RenderScene(),
ReleaseView() and

QuitApplication().

The user also needs to register his application class through the *NewDemo()* function.

The first two functions, as their name implies, are used for the initialization of the program. *InitApplication*() is called only once to initialize variables that don't depend on the rendering context that follows, such as object and texture handles. *InitView*() is called for the initialization or change of the rendering context like the creation, compilation and linking of the shaders, the vertices' creation, the textures' creation or loading and all similar variable initialization.

RenderScene() is the main rendering loop function. The function is called in every frame and manages all relevant OS events.

ReleaseView() function is called before the *QuitApplication*() function or before the change of the rendering context. It frees all handles from the existing vertices and textures, programs and shaders. *QuitApplication*() is called before finalizing the application.

PVRShell also offers a wide variety of helper functions like functions for passing and getting data from the *PVRShell* and other helper function which we will see while reviewing the program.

2.4: Texture Compression

OpenGL ES 2.0 supports both compressed and uncompressed texture image data. It goes without saying that compressed textures should always be used against the uncompressed ones, whenever that is possible.

Using compressed textures can prove to be very effective on the performance of our device. Modern applications need more and more textures in order to represent a satisfying result. The first and obvious reason to compress textures is to reduce the memory footprint of the textures. This allows the user to fit more textures on a given amount of memory which will probably increase quality. Furthermore, any memory savings are very useful for mobile devices where memory is shared across the entire SoC (System on Chip)

In addition, texture compression can save or utilize better the available bandwidth of memory resulting to better performance, sacrificing unnoticeable image quality. Finally, memory access is expensive in terms of energy consumption something very important for mobile devices where battery life is finite. Bandwidth savings contribute to decrease the quantity and magnitude of memory access.

The core OpenGL ES 2.0 specification does not define any compressed texture image formats. It is up to the vendor who implements OpenGL ES to provide compressed image data types. Most of us are familiar with the usual image file formats such as jpg, bmp or png. However we have to state here that there is a distinction between these kinds of formats and texture compression. Typical storage compression although occupies a small amount of storage on hard disc, it immediately decompresses when used or loaded in an application. So, it finally occupies a large amount of memory bandwidth. Texture compression schemes, designed for direct use in the GPU, always stay in the original compressed format and they are designed for random access so, consequently, they have usually fixed bitrate and high data locality. As a result texture compressions cannot achieve as high compression rates and image

quality, as the standard image compressions but they are more useful in graphics applications.

2.4.1: PVRTC

One of the most popular texture compressions in the graphics market (used in all generations of the iPhone, iPod Touch, iPad.) is PVRTC designed by Imagination Technologies. This differs from block-based texture formats such as S3TC and Ericsson Texture Compression in the representation of the compressed image. The image is represented by two lower resolution images which are bilinearly upscaled and then blended according to low precision, per-pixel weights, which results in considerable visual enhancement. This allows the use of 4-bits or 2-bits per pixel compression ratio which in turn leads to memory footprint savings compared with 32-bits per pixel textures. To conclude the use of PVRTC leads to reduced power consumption, increased performance, more textures in the same budget or instead noticeable high quality.

Chapter 3

3.1: Project Overview

At this project we are implementing an interface for a smartphone, using OpenGL ES 2.0 alongside IMG's SDK, which supports all basic menu functionality using 2D textures. This means, for example, that we can type a phone number but not make the actual phone call. We will present all code, talking about all implementation details, difficulties faced, tools used and general approach. At this point I would like to mention that before to undertake this project I had no previous experience in the field of graphics thus I will present it from the point of view of a newbie, which I think makes this project suitable for anyone who wants to use it as a stepping stone to enter the OpenGL ES world.

While trying to plan my next step in order to find a suitable approach on this thesis, I tried to build a plan of how I was going to start. As expected, it makes more sense to start from a general design of the interface and then gradually move on details. The basic parts of any mobile phone should be included as well as a structure for each one of the sections of the menu.

Starting from the main menu structure, one of the main features that any smartphone interface includes is the viewport sliding. Usually this happens after a prolonged gesture along the screen. This gesture is usually captured by certain tools, often provided by the operating system used by the smartphone. Since the SDK of IMG does not provide such tools, as that is not its purpose, the sliding will be done with a touch of the user at a certain part of the screen. Furthermore, the placement of the objects in the main menu was designed so that it will be stylish and user friendly at the same time.

Following the same philosophy, the other parts of the interface are meant to be functional and pleasantly designed. The alarm clock provides a clear structure of setting it up, with notifications both at its section and at the main menu which clarify if it is activated or not. The part of the messages is furnished with a sliding interface which gives information about the number of sent messages, and it leads as well to the creation menu where the user can write a message with a virtual keyboard. A similar approach can be also found in the calling menu where another virtual telephone

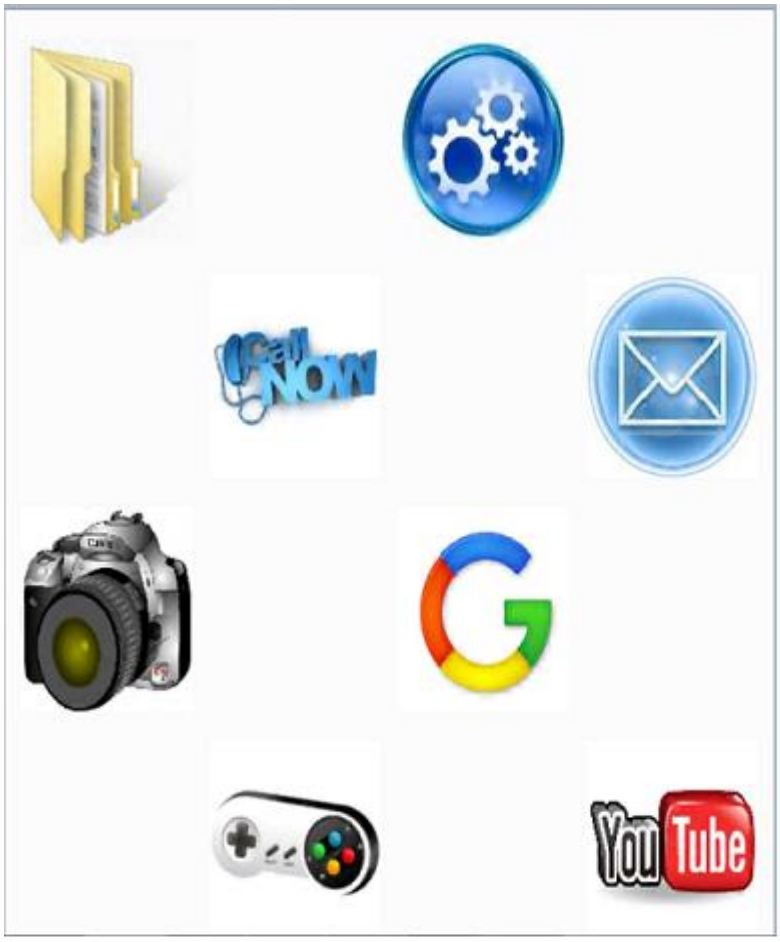
keypad is located along with other necessary buttons. Finally, in case the screen remains inactive for some time, the screensaver of the device interferes to protect the screen from burn-in pixels – although nowadays thanks to the new LCD screens, screensavers are mostly used for entertainment.

3.2: INTERFACE STRUCTURE

Before getting deep in to the code of this program it would help a lot to take a look at the structure of the interface and see what features were implemented.

The main menu of the interface is divided in four columns and six lines forming 24 objects of the same size with equal spaces between them. Only half of them are used in the menu mostly for aesthetic purposes. When running the application what you actually see is eight out of the twelve objects you would expect. In order to see the other options on which you can click on, you have to slide down your screen by clicking at the bottom left corner of your window. Clicking at the top right corner of the window slides the screen up, bringing you back where you were.

Click to slide down

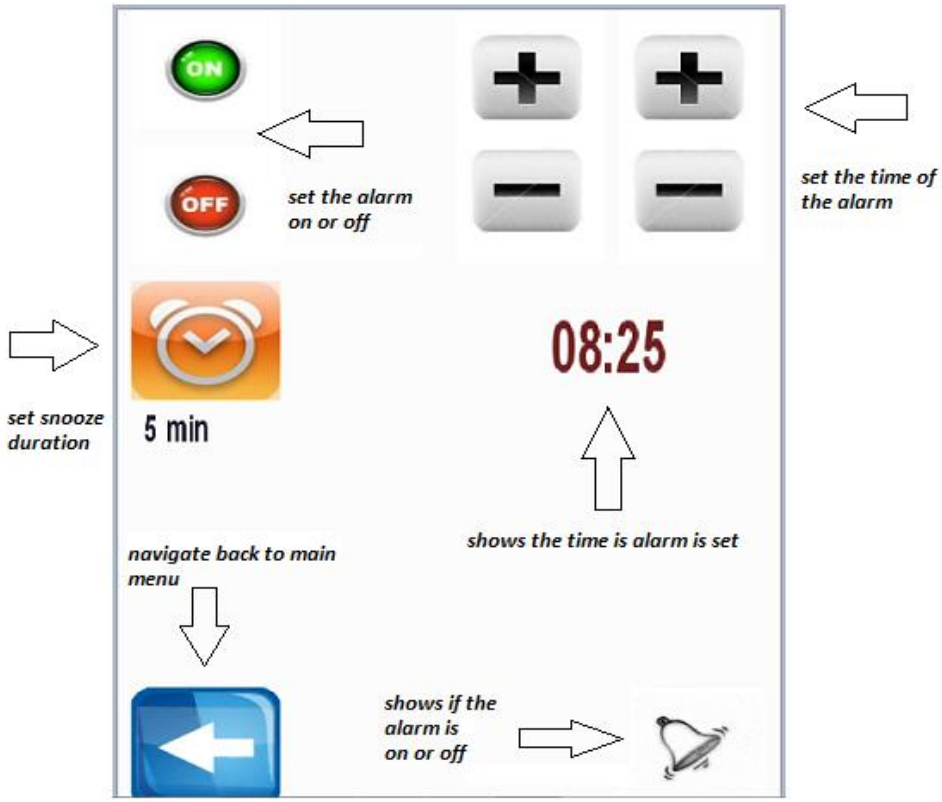


All objects in the main menu have taken their names (number) in the VBO depending on their position, according to the following diagram. The objects that were not used in the main menu are used in other parts of the interface, sometimes with changes to size or even position.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

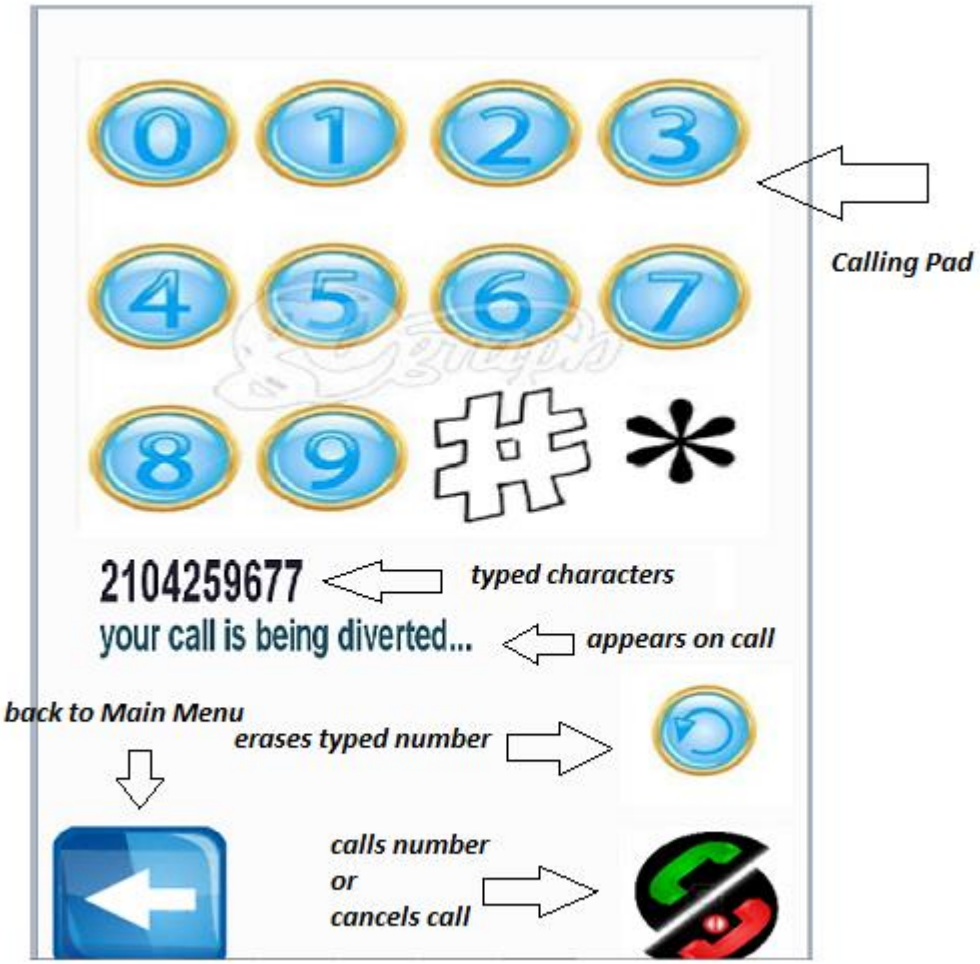
All icons in the menu interact with the user after a left mouse-click but because not all of them lead to another screen in the interface, I have chosen to project a green tick to show the interaction. The objects that do lead to another part of the interface are the messages, the alarm clock and the phone calls.

Now let us take a further look into the other parts of the interface. If you decide to navigate through the alarm clock, you will find a variety of settings that you can use. There are two “plus-minus” buttons that allow you to set the hour indication and minute indication. Under these buttons you will find the time indication and next to them you will find the buttons that set the alarm on or off. The orange button gives you the possibility of activating the snooze setting in order to set the alarm to ring again after a few minutes. The black and white bell at the right bottom of your screen is only visible if the alarm is activated. The blue button at the left bottom of the screen leads you back to the main menu.



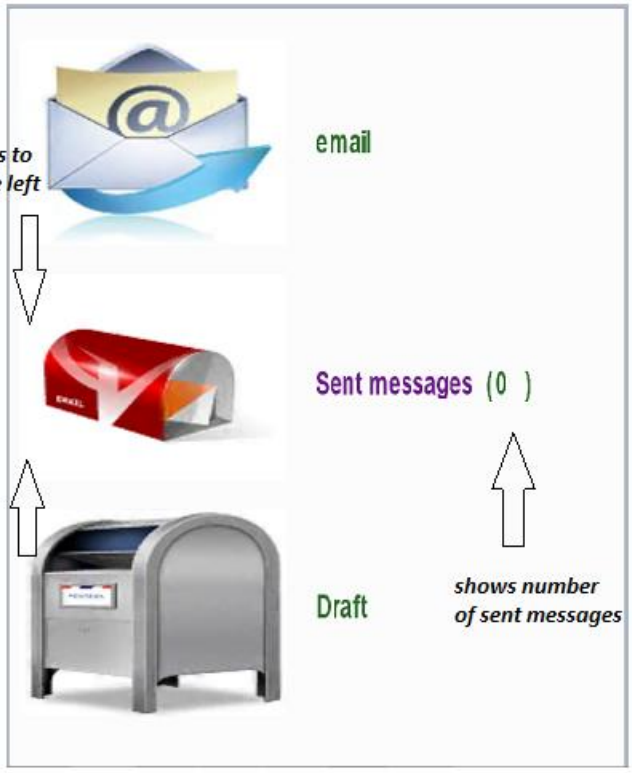
Finally, if the alarm is activated you can also see it at the upper right in the main menu, where a relevant little box appears.

If the user wants to take a phone call, he can tap on the “call now” button and immediately the phonecall pad will appear. The upper part of the screen is covered from a pad that contains all necessary characters for a call. At the bottom right part of the screen we can find two buttons: a green one for trying a phone call and a red one that cancels our current phone call. Above these two buttons, there is a circled light blue button which is used to erase any number that we might have typed and at the left bottom side of our screen we can see the arrow button that takes us back to the main menu. The numbers which the user types appear at the left centre part of the screen and under them occasionally appears a message depending on the user trying to make a phone call.

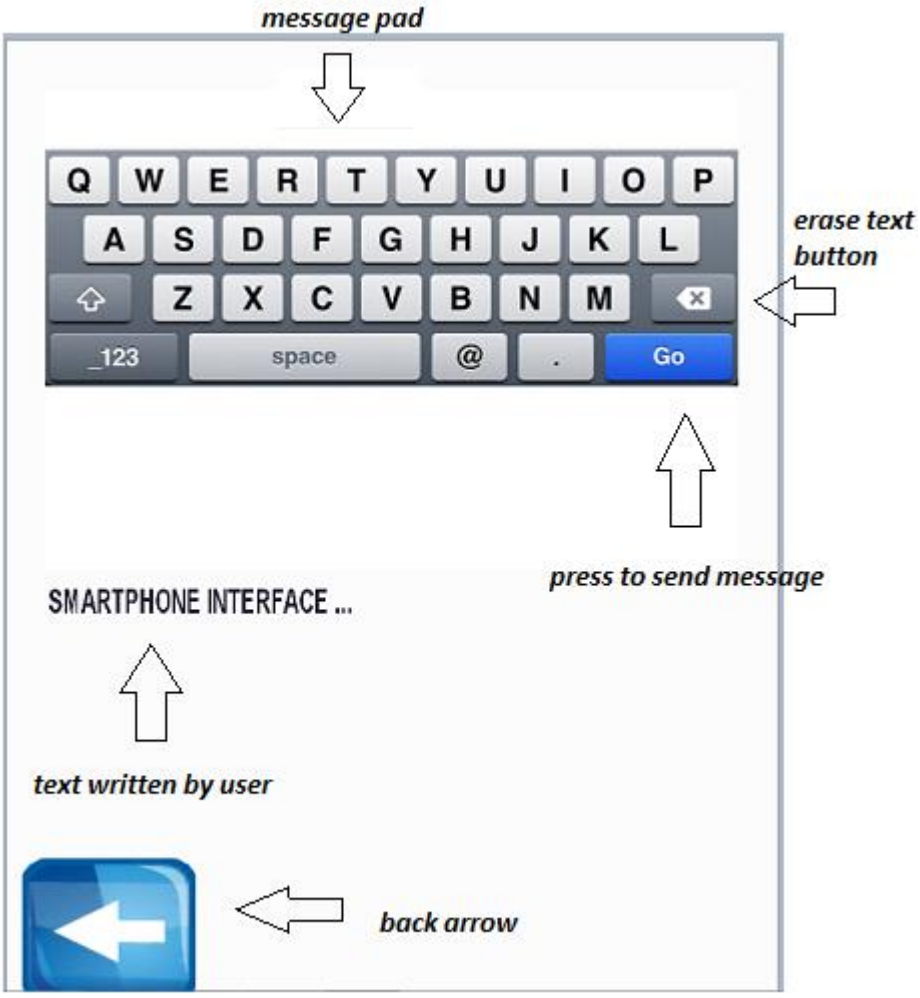


The messages main screen is divided in two parts. The first part that the user comes across entering from the main menu is the left part where you can see the incoming messages, the button that leads to the screen where you can create a message, and the back arrow. Pressing at the right part of the screen, slides you to the right where the other part of the messages menu is.

The thing to note here is the counter that keeps track of the amount of messages sent and similarly if we press at the left centre part of the screen we slide left to where we previously were.



If you try to create a message, you will find a virtual keypad with all letters and a few punctuation marks, a button to erase a part of the text in case you want to correct it, and a button that sends the message. The body of the text appears under the pad and it can take up to three lines. At the bottom left part of the screen you can see the back arrow as usual. After every sent message you will note that you are transferred to the message menu and the counter next to "sent messages" increments by one.



Finally, another feature that is added to the interface is the screen saver. After not clicking for some time on the screen, the screen saver comes along. That is in fact an IMG logo spinning around that disappears the moment we click again on the screen and we return to where we last were. Here follows an instant of the screensaver:

3.3: Shaders

Though we have not yet talked about the project itself, I think it would be more appropriate to start reviewing it starting from the shaders rather than taking it from top to bottom.

```
const char* pszFragShader = "\
              uniform sampler2D    sampler2d;\
              varying mediump float       varDot;\
              varying mediump vec2        varCoord;\
              void main (void)\
              {\
                  gl_FragColor.rgb = texture2D(sampler2d,varCoord).rgb * varDot;\
                  gl_FragColor.a = 1.0; \
              }";

       const char* pszVertShader = "\
              attribute highp vec4        myVertex;\
              attribute mediump vec3      myNormal;\
              attribute mediump vec4      myUV;\
              uniform mediump mat4        myPMVMatrix;\
              uniform mediump mat3        myModelViewIT;\
              uniform mediump vec3        myLightDirection;\
              varying mediump float       varDot;\
              varying mediump vec2        varCoord;\
              void main(void)\
              {\
                     gl_Position = myPMVMatrix * myVertex;\
                     varCoord = myUV.st;\
                     mediump vec3 transNormal = myModelViewIT * myNormal;\
                     varDot = max( dot(transNormal, myLightDirection), 0.0 );\
              }";
```

First we should take a look at the variable type modifiers used in our shaders. Uniform variables are variables that store read only values. They are useful for storing all kind of data like transformation matrices, light parameters or colours. There is also a special type of uniform variable, called sampler. Sampler is used to fetch from a texture map. The sampler uniform will be loaded with a value specifying the texture unit to which the texture is bound. Attribute variables are used only in the vertex shaders and are used to store the per-vertex inputs, such as positions, normal or texture coordinates. The final variable type modifier used in shader is varying variables. These are used to store the outputs of the vertex shaders which will be passed in the fragment shaders. So as we understand the varying declarations will be declared in both shaders identically.

OpenGL ES 2.0 also provides build-in special variables which can be output from the vertex shader and input to the fragment shader or output from the fragment shader. The first one we use to our shaders is gl_Position. gl_Position is used to output the vertex position in clip coordinates. The gl_Position values are used for converting vertex position from clip coordinates to screen coordinates. Another build-in function used in our shaders is “dot” function, which is used to compute the dot product (inner product) between two vectors. Finally, gl_FragColor contains the final output colour of a fragment.

Now lets take all declarations in our shaders to explain them one by one. Starting from the vertex shader, we have declared three attributes for the position, texture coordinates and normals of a vertex. An object such as a line or vector is called a *normal* to another object if they are vertical to each other. For example, in the two-dimensional case, the normal line to a curve at a given point is the line vertical to the tangent line to the curve at the point.

Then we have declared our projection matrix, our modelview matrix and a vector for the light direction. In short, the modelview matrix is used to define our camera and the projection matrix defines the frustum[4] of our scene. We pass the transformed vertex position as output by writing it to gl_Position and we also pass the x, y texture coordinates (since our interface will be 2 dimensional) into varCoord varying variable. Then we transform the normal vector from object coordinates to eye coordinates and we store the result in TransNormal vector. Since the normals are used to specify how much light falls on a surface or a vertex, by computing the inner product of the normal vector and the light vector we can find the diffuse lighting term and we store that term into varDot varying variable.

In the fragment shader, we declare the two varying variables that come as inputs from the vertex shader and a sampler. With gl_Fragcolor we pass the final color of the fragment and we define the RGB and the alpha value. We set the default alpha value to be 1.0 and we set the RGB values to be taken from the loaded 2D texture whose value is multiplied by the light diffusion in order to get the appropriate result.

3.4: Shader - Program Creation

The first step, after we write the code of our shaders, is to create a shader object. This is done using *glCreateShader*. This function has one enum type argument which can take either *GL_VERTEX_SHADER* value, if we want to create a vertex shader object, or *GL_FRAGMENT_SHADER*, if we want to create a fragment shader object. Once the object is created, the next thing that should be done is to provide the shader source code using *glShaderSource*. Then it would be time to compile our shader with *glCompileShader*. The first thing you want to know after compiling is whether there were any errors. This, along with other information about the shader object, can be queried for using *glGetShaderiv*. In case the shader fails to compile, the error will be written into the info log. This is a log written by the compiler for errors and messages. What we need to do next is to find the length of the log, allocate enough memory and then retrieve it. Finally we display the message in a dialog box before the application to quit. Note that this procedure should be done for both shaders. Here follows the code:

[4] a **frustum** is the portion of a solid (normally a cone or pyramid) that lies between two parallel planes cutting it.

```
	// Create the fragment shader object
m_uiFragShader = glCreateShader(GL_FRAGMENT_SHADER);
	// Load the source code into it
glShaderSource(m_uiFragShader, 1, (const char**)&pszFragShader, NULL);
	// Compile the source code
glCompileShader(m_uiFragShader);

	// Check if compilation succeeded
GLint bShaderCompiled;
glGetShaderiv(m_uiFragShader, GL_COMPILE_STATUS, &bShaderCompiled);
if (!bShaderCompiled)
{
	// An error happened, first retrieve the length of the log message
	int i32InfoLogLength, i32CharsWritten;
	glGetShaderiv(m_uiFragShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);

	// Allocate enough space for the message and retrieve it
	char* pszInfoLog = new char[i32InfoLogLength];
	glGetShaderInfoLog(m_uiFragShader, i32InfoLogLength, &i32CharsWritten,
	pszInfoLog);

	/*
	Displays the message in a dialog box when the application quits
	using the shell PVRShellSet function with first parameter prefExitMessage.
	*/
	char* pszMsg = new char[i32InfoLogLength+256];
	strcpy(pszMsg, "Failed to compile fragment shader: ");
	strcat(pszMsg, pszInfoLog);
	PVRShellSet(prefExitMessage, pszMsg);

	delete [] pszMsg;
	delete [] pszInfoLog;
	return false;
}


		// Loads the vertex shader in the same way
m_uiVertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(m_uiVertexShader, 1, (const char**)&pszVertShader, NULL);
glCompileShader(m_uiVertexShader);
glGetShaderiv(m_uiVertexShader, GL_COMPILE_STATUS, &bShaderCompiled);
if (!bShaderCompiled)
{
	int i32InfoLogLength, i32CharsWritten;
	glGetShaderiv(m_uiVertexShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);
	char* pszInfoLog = new char[i32InfoLogLength];
	glGetShaderInfoLog(m_uiVertexShader, i32InfoLogLength, &i32CharsWritten,
pszInfoLog);
	char* pszMsg = new char[i32InfoLogLength+256];
	strcpy(pszMsg, "Failed to compile vertex shader: ");
	strcat(pszMsg, pszInfoLog);
	PVRShellSet(prefExitMessage, pszMsg);

	delete [] pszMsg;
	delete [] pszInfoLog;
	return false;
}
```

The next step is to create a program and attach the two compiled shaders onto it. We have to say here that a shader can be attached to any point. It does not need to be compiled or even have a loaded source code. The only restriction is that one program object should have exactly one vertex and one fragment shader attached onto it. I just followed this order because it was suggested from all documentation I had studied and it actually seems like the most logical order to follow.

```
	// Create the shader program
m_uiProgramObject = glCreateProgram();

	// Attach the fragment and vertex shaders to it
glAttachShader(m_uiProgramObject, m_uiFragShader);
glAttachShader(m_uiProgramObject, m_uiVertexShader);
```

Next, we have to bind a generic vertex attribute index to an attribute variable in the vertex shader. This will happen with the use of *glBindAttribLocation* command. Generic attribute indices are used to enable a generic vertex attribute and specify a constant or per-vertex value. This binding will take effect when the program will be linked. The linking is responsible for generating the final executable program. We actually have to check if linking was successful in the same way that we checked if the compilation of the shaders succeded.

```
	// Bind the custom vertex attribute "myVertex" to location VERTEX_ARRAY
glBindAttribLocation(m_uiProgramObject, VERTEX_ARRAY, "myVertex");
	// Bind the custom vertex attribute "myUV" to location TEXCOORD_ARRAY
glBindAttribLocation(m_uiProgramObject, TEXCOORD_ARRAY, "myUV");

	// Link the program
glLinkProgram(m_uiProgramObject);

	// Check if linking succeeded in the same way we checked for compilation
success
GLint bLinked;
glGetProgramiv(m_uiProgramObject, GL_LINK_STATUS, &bLinked);

if (!bLinked)
{
	int i32InfoLogLength, i32CharsWritten;
	glGetProgramiv(m_uiProgramObject, GL_INFO_LOG_LENGTH, &i32InfoLogLength);
	char* pszInfoLog = new char[i32InfoLogLength];
	glGetProgramInfoLog(m_uiProgramObject, i32InfoLogLength, &i32CharsWritten,
	pszInfoLog);

	char* pszMsg = new char[i32InfoLogLength+256];
	strcpy(pszMsg, "Failed to link program: ");
	strcat(pszMsg, pszInfoLog);
	PVRShellSet(prefExitMessage, pszMsg);
	delete [] pszMsg;
	delete [] pszInfoLog;
	return false;
}
```

Next we have we have to get the uniform location and set the sampler 2d variable to the first texture unit and finally decide the background colour of the screen. The first three values of *glClearColor* determine the RGB value of the colour and the final argument describes the alpha value.

```
	// Sets the sampler2D variable to the first texture unit
glUniform1i(glGetUniformLocation(m_uiProgramObject, "sampler2d"), 0);

	// Sets the clear color
glClearColor(0.99f, 0.99f, 0.99f, 1.0f);
```

The RBG colour model is based in the addition of three basic colours (red, green and blue) in order to reproduce all other colours. The main purpose of the model is the display of images in electronic devices, like for example a portable device on which this program can be used to. The alpha value determines the transparency of an object; in this case the transparency of the background and that is why it is set to 1.0 as we want the background to be opaque.

3.5: Object generation

Thereafter, we have to generate an array buffer object in which we will store all the vertex attribute data for our vertices. We proceed to this generation after calling the *glGenBuffers* command in which we pass as arguments the number of objects we want to create and a pointer to the array of the objects.

```
glGenBuffers(26, &m_ui32Vbo[0]);
```

Each object should be bound at one position in the array after we have initialized its primitives, texture coordinates and normal. Each primitive has a triad of x, y, z coordinates, a pair which shows the texture coordinate and a triad with the normals' coordinates sequentially. We always have to keep in mind that in OpenGL ES 2.0 we can only create points, lines or triangles. So if we want to create a square or a rectangular we have to use two triangles. The z coordinate is always zero in our case since the interface will be 2 dimensional.

```
                            // Position          UVs          Normals
GLfloat zerVertices[] = { -0.51f,0.70f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                          -0.96f,0.70f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                          -0.96f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                          -0.51f,0.70f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                          -0.96f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                          -0.51f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
```

The vertex array data storage is created and initialized using the *glBufferData* command. In this command we have to pass the size of the buffer data in bytes, a pointer which shows to the buffer data and a hint of how the application will use the

buffer object. In this case we have used the *GL_STATIC_DRAW* argument which declares that the buffer object will be specified once and it will be used many times in order to draw the primitives. After the initialization of the buffer object data, we can unbind current selection and move on.

```
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, zerVertices,
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Taking a look at the matrices that store the vertices information about each object, we will notice that are grouped together in a certain structure. This is not actually essential for the application rather than it makes it easier for the developer to read out the information. Each line in the matrix contains information about a single point on the screen.

The first three numbers are the x, y and z coordinates of the point. These coordinates can take values from -1 to 1, with the (-1,-1) coordinate being the bottom left of the screen and the (1, 1) coordinates being the top right of the screen. The next two values are the texture coordinates. What we actually have to do here is match the bottom left side of our texture with the bottom left side of our object, the top left side of our texture with the top left side of our object and so on. The values that the textures coordinates can take, has range from 0.0 to 1.0.

Finally, the last three values in the row are the normals. As we mentioned before, normals are vectors which are vertical to the tangent plane of the surface and actually determine the surface's orientation towards the light source.

3.6: Texture loading

PVRTexTool:

PVRTexTool is a suite of utilities for compressing textures, an important technique that ensures the lowest possible texture memory overhead at application run-time. PVRTexTool's components include a library, a command line with GUI tools and a set of plug-ins, all available for Autodesk 3DSMax, Maya, and Adobe Photoshop. The tool offers the user a variety of compressed texture formats such as PVRTC and ETC. It also includes many pre-process image features like, resizing, border generation and image bleeding.

My experience, though I did not go deep into the features offered, was quite pleasant. The tool was easy to use, the GUI had adequate clarity and the documentation offered had clear and up to the point instructions. The only feature I found a bit weak was the resizing of objects. Final texture was often blurry especially when taking very large pictures and creating smaller – which was my case mostly. I overcame this problem using Microsoft Windows Paint. I resized all textures I used in Paint, in which the result was more than satisfying, and then I performed vertical flipping and encoding using the PVRTexTool. Another point that the user should keep in mind, and this one has nothing to do with the tool but mostly with PVRTC, is the quality which will choose for the encoding of his texture. Surely that won't be the

case in a portable device where the screen is small and at the same time the processing speed limited, but at the screen of my personal computer choosing a medium quality encoding had unpleasant effects at the final result, thus I preferred to use higher quality – since the processing power was more than plenty. Of course, installing the interface in a portable device will maybe demand to lower graphics quality in favour of rendering speed and at the same time it is quite probable that this decision won't have any real cost at the result illustrated.

Filewrap:

"Filewrap is a utility that 'wraps' external data required for an application so that it can be included within an executable. Data is statically linked into the application and may be accessed at run time using the *PVRTResourceFile* tools code or from your application. This is useful for platforms without a file system and for application distribution and deployment because it keeps all required data in one place: the application executable."[5]

Further use of available tools:

The last tool we will use to load textures in our application is the *PVRTTextureLoadFromPVR* function. This function which is included in IMG's SDK takes as first parameter the name of the file which we are going to load and as a second parameter the resulting texture handle. The third parameter is a *CPVRTString* for error message output.

Finally, it is necessary to set up our filtering mode in order to use mipmaps. The result will be to avoid aliasing artifacts that occur if the ratio between screen pixels and texture pixels is not good. There are two kinds of filtering: minification (when the texture should be compressed to the size of the polygon in which should fit) and magnification (when the texture should stretch in order to fill up the polygon). The decision between minification and magnification is done automatically by the hardware, but the developer has control over the type of filtering that he will use in each case.

This is all done by the *glTexParameter* fuction. In the first parameter we declare if our texture is 2D or 3D, then if the following type of filtering refers to the minification or the magnification filter, and finally we decide which type of filter to use depending on how much performance we are willing to sacrifice in order to get high visual quality. For the magnification, we can choose between *GL_LINEAR* and *GL_NEAREST* filter. In the first case, a single point is taken from the texture nearest to the texture coordinate while in the second case an average of four samples is taken. When it comes to the minification filter we certainly have more options as we are allowed to choose not only between linear or nearest filters, but also a combination of filters which take samples from the mip-map levels of the image. The main aspect that sets our decision on deciding which filter to use, is the hardware. Knowing the features of the hardware on which we will run the application, help us balance between performance and quality. In our case we chose a linear filter for

[5] IMG forum

magnification and a bilinear fetch from the closest mip level for minification as our application should run efficiently on portable devices on which resources are limited. The mip-map filtering mode is usually the best choice when it comes to minification as it provides good quality.

```
……
……
……
if(PVRTTextureLoadFromPVR(c_aeTextureFile, &m_uiTexture[29]) != PVR_SUCCESS)
{
	PVRShellSet(prefExitMessage, "ERROR: Cannot load the 30th texture\n");
	return false;
}
	//////////////////  MesPad
if(PVRTTextureLoadFromPVR(c_afTextureFile, &m_uiTexture[30]) != PVR_SUCCESS)
{
	PVRShellSet(prefExitMessage, "ERROR: Cannot load the 31th texture\n");
	return false;
}

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

3.7: Picking Techniques

When we use the term picking in graphics, we refer to selection of objects via a mouse click or a tap when it comes to mobile devices. There are several methods to implement picking in OpenGL ES. We will mention the main methods and after this we will go through our choice in the project and how did we end up with this solution.

To start with, the most straight forward technique when it comes to picking is ray tracing. Before implementing the ray tracing technique, it is quite useful to create a bounding box around every object, especially if the object does not have a homogenous form. Then, you fire a ray from the user's side, starting at the pixel that was clicked and then run an intersection test to determine which object has been intersected first by the ray. This technique is very efficient in 3D graphics world where not only we find many objects in the screen but also many of them could be placed one behind the other. That way we can easily determine which object is closest to our camera. The disadvantage of this technique comes when we have moving objects on screen. In that case, since firing a ray and having the intersection test takes some time, there is a chance that until the procedure to be completed the object to change position. So the user may click on the object, but there is a chance that the results of the intersection test to show he has not tapped on it.

Another way to determine which object is selected by the user is colour picking. What happens in this case is that after we apply one flat colour to each object, we try to determine which colour is under the pixel the user has selected, and so we get to know which object is chosen. However, there is one major drawback in this technique. The only way in which we can check onto what colour the user has

clicked on, is by using *glReadPixels* command which essentially slows down the application significantly.

To further explain this we have to take a closer look at hardware side of a system. What happens generally is that the CPU and the GPU are working in parallel. Since they have different amounts of workloads to deal with, all calls are queued and served when the GPU is ready to do so. When we call the *glReadPixels* command we actually wait for the GPU to have finished all processing, since an undone texture is useless. So as a result the CPU, which is always a bit ahead of the GPU in terms of the stream that is being processed, actually stays idle waiting for the GPU to finish processing. So *glReadPixels* should be used only if there is no other way to perform our tasks.

After taking sometime tracking down all available options about picking, what I realised is that all solutions offered online and in bibliography where actually trying to deal with 3D issues, like objects hiding one behind another. So, since our application is 2D and our objects are not moving around, there was no need to get involved with so complex techniques. The solution that seemed simpler was to locate the user click on our screen, and since we know where the app's objects are located in each section of the menu, we would decide accordingly what our next action should be.

In order to achieve this, we need a variable –it is called "pressed" in our case- which takes its value depending on the click position and on the screen of the interface we are into. To get a mouse event and determine the coordinates of it, we use the *PVRShellGet()* tool function which is used to bring data from *PVRShell*. By passing as an argument the *prefPointerLocation* key word we can take at all times the mouse position. We temporarily pass the return value of the function in a vector, as it returns two values – one for the x and one for the y coordinate - , and then we store the first position of the vector in a new x variable and the second position in a y variable. So at each frame that is rendered we store the new coordinates of the mouse at these variables. To get the a mouse event we pass the argument *prefButtonState* at the *PVRShellGet* function and in that case it returns value 1 if the mouse button is pressed or otherwise returns 0. Here follows an example of how the "pressed" variable takes its value depending on mouse events and how the x and y variables contain the mouse position at the frame.

```
	// Depending on click position pressed takes the appropriate value
if(0 != PVRShellGet(prefButtonState) && x<0.25 && y<0.25)		{pressed=1;}
else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y<0.25)
                                                                {pressed=2;}
else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.25 && y<0.5)
                                                                {pressed=3;}
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.25 && y<0.5)
                                                                {pressed=4;}
else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.5  && y<0.75)
                                                                {pressed=5;}
else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y>0.5 && y<0.75)
                                                                {pressed=6;}
else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.75)
                                                                {pressed=7;}
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.75)	{pressed=8;}
```

```
	// Takes the mouse position
if (NULL != (vec2PointerLocation = (float *)PVRShellGet(prefPointerLocation)))
{
	x = vec2PointerLocation[0];
	y = vec2PointerLocation[1];
}
```

As we mentioned before, this is a simpler solution comparing to the other two techniques because we avoid a lot of algorithm and code complexity. On the other hand, it has not only advantages but also disadvantages. The main problem to deal with in this technique is potential unwanted "clicks" that may occur from time to time. For example you will realize that sometimes if you click on an object for more than a moment, you might get to interact with an object from the next scene that will appear. That same problem might also come up when typing a message or a call number, where double clicking on a character without intention can be annoying. In order to get past this, we should take care so that objects that lead to changing scene do not lay one behind the other when possible. For example, an object at the main menu would be better if it did not have the same coordinates as the "back" arrow from the phone pad, so even if we click prolonged at the arrow we will not press anything in the main menu by mistake.

Another way, in case we cannot go through the previous compromise, would be to call the *Sleep()* function and idle the system for a few milliseconds so that we give the user the necessary time for pressing each button. This solution, though sounds simple and straight forward, must be used moderately because it slows down the application a lot, and since this app is meant for real time embedded devices, performance is very important.

3.8: Main Menu rendering

To begin with, every section of our interface is implemented in a different function for code readability reasons. All these functions are called in *RenderScene()* depending on the values of our parameters.

The Main Menu function is probably the most simple and easy to go through comparing to all other functions. What we have to do first is bind the projection matrix to the associated uniform variable in the shader. We can get the location of the variable with *glGetUniformLocation* command. Then we pass the matrix in this variable with *glUniformMatrix4fv*. Accordingly, we find the location of the model view matrix and the light direction and pass them in the shader.

The last part of the *glUniform* command is not always the same though it is kind of self-explanatory. As you probably already suspected, it depends on the kind of uniform. If you want to set a simple uniform you have to use *glUniform{number}{if}* where number '1' stands for a float element, boolean or integer, '2' stands for vec2, '3' for vec3 and '4' for vec4. If you want to set an array you just place a "v" at the end – stands for vector – and if you want to set a matrix data type then you have to use GlUniformMatrix{2,3,4}fv depending on what you want: mat2, mat3 or mat4. There are two important things to remember here: the first one is that one uniform can be used from both shaders as long as it is declared in both of them. The other thing to notice is that glUniform* command is set to the current program in use. So it must be used after we have selected our program – through the *UseProgram* command.

```
/*
	Bind the projection model view matrix (PMVMatrix) to the
	corresponding uniform variable in the shader.
	This matrix is used in the vertex shader to transform the vertices.
*/
int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
/*
	Bind the Model View Inverse Transpose matrix to the shader.
	This matrix is used in the vertex shader to transform the normals.
*/
zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);

	// Bind the Light Direction vector to the shader
zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
glUniform3f( zLocation, 0, 0, 1);
```

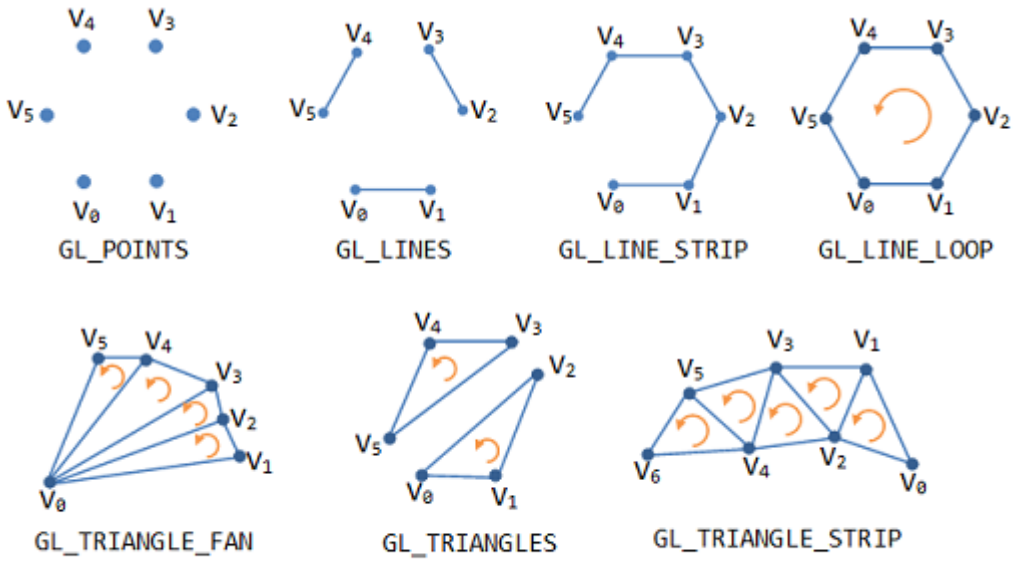
Further on, we bind the Vertex Buffer Object with *glBindBuffer*, and we enable the vertex attribute at the index *VERTEX_ARRAY* with *glEnableVertexAttribArray*. Then we have to point to the data for this vertex attribute with *glVertexAttribPointer*. There are two ways to store vertex attributes. Either put them all together in a single buffer and form an array of structures, like in our case, or put them in separate buffers and form a structure of arrays. Taking all arguments in *glVertexAttribPointer* from the beginning, we have the vertex attribute index which is 0, the number of components in our vertex array that in our case is three (primitives, texture and normals), the data format, an indicator which specifies if the non-floating data format should be normalized, the stride which is the interval of elements from primitive to primitive and finally a pointer to the data.

```
	// Bind the VBO
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
	// Pass the vertex data
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
```

Then we have to go through the same procedure for textures and normals before we draw our triangles and finally unbind the buffer. Here, we show the interaction of all objects with the mouse event, with the use of the declared variable "pressed", which changes value depending on the coordinates on which the user has clicked. So we bind the appropriate texture and project it on screen: the normal object texture or the "green tick". That of course goes only for the sections of the menu that are not enhanced.

```
if(pressed==1)
{
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
}
else
{
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[0]);
}
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride,
(void*) (3 * sizeof(GLfloat)));

	// Pass the normals data
glEnableVertexAttribArray(NORMAL_ARRAY);

glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride,
(void*) (5 * sizeof(GLfloat)) /* Normals start after the position and uvs */);
```

glDrawArrays is used to draw the primitives in our screen. We have to specify the kind of primitive we want to render, the stating vertex index and the number of indices to be drawn, which is 6 for drawing a square (two triangles). There are multiple kinds of primitives that we can select to draw. Of course we can draw points, lines and triangles as we have already mentioned but we can also draw some combinations formed of these three basic elements too. These would be a line strip, a line loop, which is a strip that ends where it started, a triangle strip and a triangle fan. After we complete the drawing we can finally unbind the current buffer.



OpenGL Primitives

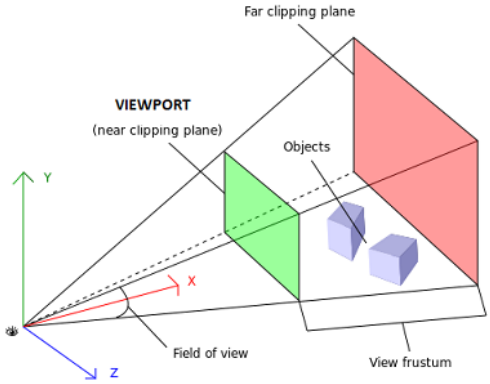
```
        // Draws a non-indexed triangle array
    glDrawArrays(GL_TRIANGLES, 0, 6);
        // Unbind the VBO
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

In the end of this function you can notice an object that is only rendered if a variable called "AlActivation" is of value 1. This is for the rendering of the sign that indicates if the alarm clock is switched on, but we will talk about this more later on.

```
    // This obj will be rendered only if the alarm is activated
if(AlActivation==1)
{
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[3]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[20] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride,
(void*) (3 * sizeof(GLfloat)));

glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride,
(void*) (5 * sizeof(GLfloat)) );

glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);

}
```

3.9: Sliding

Another feature implemented in our smartphone interface is the sliding of the screen. In order to further explain the implementation we first have to explain the term of the viewport. The term refers to a 2D rectangle that is used to project a scene of the virtual camera. It is actually the portion of the total image that is visible from the user.



The command that allows the developer to change the viewport is *glViewport*. The command has four arguments. The first two handle the x and y coordinates of the viewport and the other two handle the zoom of the viewport in the view frustum. The way to achieve our goal, slide our viewport, would be to zoom in a certain area of our screen and then to pass a variable argument in *glViewport*, in order to create a loop and slide down or up depending on the user's mouse click and our current position. What is important to notice here is that *RenderScene()* is called for every frame that appears on screen. So it is a loop function that breaks the loop only when we decide to terminate the application. As a result, using another loop to slide down the screen would lead to purposeless loops! The way to solve this problem is to use an "if" statement. When the user decides to slide the screen, we diminish or increase the argument variable in *glViewport* at each frame, until a lower or upper bound, which sets our final view of the scene.

```
if(UpDown==0 && pressed!=7 && pressed!=2 && pressed!=8)
{
	if(kapa>-350){kapa-=SCREEN_SPEED;} // Change viewport position if needed
	glViewport(0,kapa,600,1100);
	Main_Menu();

.
.
.
.
.

else if(UpDown==1 && pressed!=7 && pressed!=2 && pressed!=8)
{
	if(kapa<0){kapa+=SCREEN_SPEED;} // Change viewport position if needed
	glViewport(0,kapa,600,1100);
	Main_Menu();
```

3.10: Screen saver implementation

In all smartphones released in the market, there is some protection mechanism in case the screen stays idle for some time. That would be either to use of a moving screen saver or a screen switch off (which combines both protecting the screen and saving battery power). Since the switch off of the screen does not present any interest from the developer's point of view, in this project you will find a screen saver implementation.

The first thing that we should consider about this is the way to determine for how long the screen has been idle so as to enable the screen saver. The other important issue to find a way to disable the screen saver when a mouse-event occurs, and return back to the screen we were before.

Taking one thing at a time let us see how to solve the first part. Here comes again our valuable IMG SDK, to give a solution that simplifies things a lot. The tool function offered to us hears the name *PVRShellGetTime()* and since it is first called in our application, it enables a clock counting time. So, in *InitVIew()* where we initialize the basis of our application we call *PVRShellGetTime()* and pass its return value into a variable called "StartTime".

```
m_ulStartTime = PVRShellGetTime();  // Initialize StartTime
```

Thereafter, at every frame (in *RenderScene()*) we call again our tool function, passing its return value to a variable called *CurrentTime*. Subtracting *CurrentTime* minus *StartTime* gives us the time that has passed since the application started. But since we do not only care when the app started but also when a mouse event occurred, at every mouse click that happens we reset *StartTime* by passing a new return value from *PVRShellGetTime().* That way we can check at each frame how much time has passed since the last mouse-event, and if that time exceeds a limit of our choice, we enable the screen saver. It is needless to say, that after the screensaver has been

enabled, in case a mouse-click occurs, the *StartTime* resets and so we disable the screen saver and return to our previous screen.

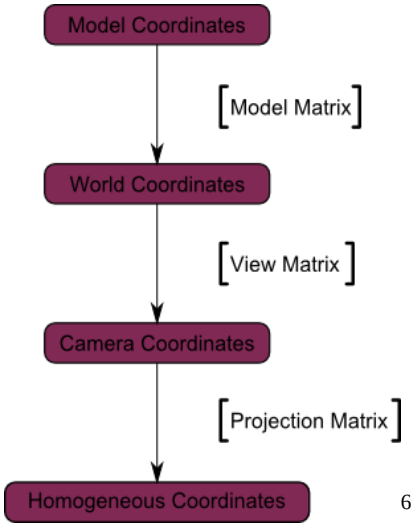
```
	// Time passed since last mouse click
unsigned long ulCurrentTime = PVRShellGetTime() - m_ulStartTime;

if(0 != PVRShellGet(prefButtonState))
{
	m_ulStartTime = PVRShellGetTime(); // After mouse event reset CurrentTime
	Sleep(50);			//  To avoid pressing any Main Menu buttons
after ScreenSaver return

}

.
.
.
.

if(ulCurrentTime>TIMER)	// If currentTime exceeds TIMER call ScreenSaver(in ms)
{
	glViewport(0,0,600,750);
	ScreenSaver();
}
```

As you probably noticed, the screen saver enabling depends on a mouse-click and not from mouse moving as it happens for example in a personal computer. That is because this application is meant for an embedded touch screen device where touching the screen is actually like having the mouse clicked continuously. So that is the event that should trigger the screen saver disabling rather than just a mouse move.

Moving on the screen saver itself, it is interesting to see how the spinning of the object is implemented. In order to rotate it, we use different projection and model view matrices comparing to what we used in the other screens. We pass sine and cosine values in the matrices that depend from an "*angle*" variable, which controls the angle, and increments constantly frame after frame. The rotation takes place around the y-axis. The right hand rule defines the positive direction of all axes and so that explains why our object moves anti-clockwise when the angle is incremented.

```
	// Different ModelView and Projection Matrices
	// in order to achieve object rotation

float aModelViewIT[] =
{
	cos(m_fAngle),		0,	sin(m_fAngle),
	0,			1,	0,
	-sin(m_fAngle),		0,	cos(m_fAngle)
};


float aPMVMatrix[] =
{
	cos(m_fAngle),		0,	sin(m_fAngle),		0,
	0,			1,	0,			0,
	-sin(m_fAngle),		0,	cos(m_fAngle),		0,
	0,			0,	0,			1
};


int i32Location = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
glUniformMatrix4fv( i32Location, 1, GL_FALSE, aPMVMatrix);
i32Location = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
glUniformMatrix3fv( i32Location, 1, GL_FALSE, aModelViewIT);
i32Location = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
glUniform3f( i32Location, 0, 0, 1);

	// Increments the angle of the view
m_fAngle += .02f;
```

I believe here is the right moment to tell a few more things about matrices, that will help us understand a bit better all the above. Every object in our screen is defined by a set of vertices that represent its x, y and z coordinates. What the model matrix does is that it transforms the coordinates of the object in order to bring it to the centre of our world space. We perceive this world space in our screen in the same way like watching it through a camera. So the camera is actually the origin of our world space. In order to move the world we use the view matrix. Now we can better understand the use of our model-view matrix in our program. Finally, the projection matrix is the one which gives us the perspective view of the world, by configuring the size of the objects depending on their distance from the camera.



[6]

3.11: Pad implementation

The first part of the smartphone interface - other than the Main Menu - that we will go through, concerning the implementation, is the pad that is used for phone calls. The primary problem that we faced here was the typing of the numbers to call. There are two ways one can use in order to reach the desirable result. One is certainly way more efficient than the other, and it is only provided by the tools of Imagination Technologies.

So, generally speaking, there is no standard way of printing text on screen using OpenGL ES. One way to overcome this issue would be to create a set of textures with all letters, numbers and symbols that you would need, thereafter create a set of objects which would have the screen coordinates on which you would like to print and then combine the above and render the text that you want. Obviously, this solution is neither efficient when it comes to performance, nor suitable for the developer as it requires spending a lot of time and effort on something really uninteresting from the implementation's point of view.

The other solution comes from the toolkit of IMG and it is called *CPVRTPrint* 3D class. This is a tool function that allows us to draw text anywhere in our app window just by using the keyboard. We can also choose the size and the color of the letters drawn. The source code of the function is open and it can be modified by anyone in order to cover further possible needs.

The class uses its own shaders in order to draw the text and though any changes we make to our shaders don't affect the text drawn. That is very important to remember for two reasons.

First is that no matter the implementation to our objects to be moving, spinning, in a 3D frustum or any other effect, the text will be static and 2D and will not be affected by all these procedures. If we want to make any changes regarding the text we will have to interfere in the source code of the *Print3D* class.

Now, before to further talk about the second reason which we always have to keep in mind that *Print3D* uses its own shaders, let us remember something that we

[6] *http://www.opengl-tutorial.org*

mentioned earlier when we talked about OpenGL ES. “The only restriction is that one program object should have exactly one vertex and one fragment shader attached onto it”, which leads us to the conclusion that since *Print3D* uses its own shaders, it also uses its own program. All the rendering that is implemented in *RenderScene*, has its basis on our shaders, and all commands actually refer to the last Program used! So, if we put *UseProgram* in the *InitView* function, which is called only at the beginning of our application, when we will call *CPVRTPrint3D*, the used program will change and as a result at the next loop of *RenderScence* we will be using the shaders of *CPVRTPrint3D*! So, the moment we will flash the context of *Print3D* to the screen, everything will be lost from screen. Instead, we should put the *UseProgram* command at the beginning of *RenderScene* function, and so at each loop we will refresh to our previous shaders. This, although may seem obvious, it took me quite some time to notice and thus I highlight it. Especially if you have no experience when it comes to OpenGL ES, it can be proved to be tricky.

```
bool OGLES2Texturing::RenderScene()
{
glUseProgram(m_uiProgramObject); // Reuse our shaders as Print3D uses its own
                                        shaders when called

       // Clears the color and depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
.
.
.
.
```

Next, we should see the use of the class. The first thing to do is create an instance of the class. Then, we should see the arguments passed when we are calling the instance we declared. The first two arguments refer to the coordinates of the text. The third argument determines the size of the text, the next one decides the color of the text in an ARGB format, and finally we put the string we want to project in quotation marks. All this won’t work until we flush the text at the screen.

The next thing to consider is how we will put the characters which the user will choose on screen. First thing to do is declare a variable that will change value at every frame, depending on where the user has clicked. Then, it is wise to put the text in an array which will be filled according to the value of the previous variable. Finally, we pass this array as an argument to the *Print3D* call. We have also set a counter, which restricts the typed number length to 20 characters.

```
NumPad();   // Called to set the PrNum value according to user's click

if(PrNum==0)
{
	if(counter<20)                  // to avoid text overflow
	strcat(NumToCall,"0");      // add character to text
	counter++;                      // increment counter
}
else if(PrNum==1)
{
	if(counter<20)
	strcat(NumToCall,"1");
	counter++;
}
else if(PrNum==2)
{
	if(counter<20)
	strcat(NumToCall,"2");
	counter++;
}
.
.
.
.

	// If no erasure command has been given, the number is flushed on screen

if(erase==0)
{
	m_Print3D.Print3D(10.0f, 45.0f, 1.0f, 0xFF302020, NumToCall);
	m_Print3D.Flush();
	Sleep(50);
	PrNum=100;  // Change PrNum value to avoid constant printing of the pressed
		    number
}
```

Finally, we copy an empty string to our array that contains the number in order to erase the number if the user chooses so by pressing the appropriate button; or if he decides to go back at the main menu and we also flush the appropriate text – "your call is being diverted"- if the user decides to call the keyed in number.

```
	// If erasure command has been given copy an empty string to the array
else if (erase==1)
{
	erase=0;                    // Reset erase
	strcpy(NumToCall,empty);
	counter=0;                  // Reset counter
}
```

```
        // if call button is pressed and a number is dialed print message
if(dial==1 && NumToCall[0]!=NULL)
{
        m_Print3D.Print3D(10.0f, 50.0f, 0.8f, 0xFF605020, "your call is being
        diverted...");
        m_Print3D.Flush();
}

        // If "back" arrow is pressed:
if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.83 )
{
        pressed=0;                  // Change value to return to main menu
        strcpy(NumToCall,empty);    // Erase dialed number
        dial=0;                     // Erase printed message
        counter=0;                  // Reset counter
}
```

3.12: Alarm clock implementation

As it may be suspected at the presentation of the GUI, the alarm clock function has some interesting staff to be seen. To start with, the first issue that was faced here was the implementation of the mechanism that would spin the time when the user would press the plus-minus buttons. Depending on the user's tap we have created a four way branch. Each branch is for for incrementing or decreasing, the hour or the minute indication respectively. In these branches we ensure that the hour indication circles between zero and twenty three and the minute indication does not exceed fifty nine of diminishes in values less than zero. Each digit is independent from the one next to it, so we had to make sure that the digits act in pairs.

```
        // The following algorithm handles the hour incrementing
        // when the user presses the "+" button
else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y<0.15)
{
        if(lhour==0 || lhour==1)
        {
                rhour++;
                if(rhour>9)
                {
                        rhour=0;
                        lhour++;
                }
        }
        else if(lhour==2)
        {
                rhour++;
                if(rhour>3)
                {
                        lhour=0;
                        rhour=0;
                }
        }

}
```

```
	// The following algorithm handles the hour diminishing
	// when the user presses the "-" button
else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y>0.15 && y<0.30)
{

	if(lhour==0)
	{
		rhour--;
		if(rhour<0)
		{
			lhour=2;
			rhour=3;
		}
	}
	else if(lhour==1 || lhour==2)
	{
		rhour--;
		if(rhour<0)
		{
			lhour--;
			rhour=9;
		}
	}
}
	// The following algorithm handles the minute incrementing
	// when the user presses the "+" button
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y<0.15)  // minute  +
{
	rmin++;
	if(rmin>9)
	{
		rmin=0;
		lmin++;
		if(lmin>5)
		{
			lmin=0;
			rmin=0;
		}
	}
}
	// The following algorithm handles the minute diminishing
	// when the user presses the "-" button
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.15 && y<0.30)
{
	rmin--;
	if(rmin<0)
	{
		rmin=9;
		lmin--;
		if(lmin<0)
		{
			lmin=5;
			rmin=9;
		}
	}
}
```

The next issue here is that we have to use *CPVRTPrint3D* to display the time on the screen and the tool function cannot take a number as an argument. This can be addressed with "*itoa*" function, which transform an integer to string and so we are able to pass the integer time in a matrix as a string, and then pass the matrix as an argument in *Print3D*.

```
		// Turns the time intergers into strings
		// so that we can project it on screen
		// with CPVRTPrint3D class
	itoa(rhour,buffe,10);
	itoa(lhour,buffer,10);
	itoa(rmin,buf,10);
	itoa(lmin,buff,10);

	m_Print3D.Print3D(99.0f, 30.0f, 1.5f, 0xFF202070, buf);
	m_Print3D.Print3D(94.0f, 30.0f, 1.5f, 0xFF202070, buff);
	m_Print3D.Print3D(91.0f, 30.0f, 1.5f, 0xFF202070, ":");
	m_Print3D.Print3D(87.0f, 30.0f, 1.5f, 0xFF202070, buffe);
	m_Print3D.Print3D(82.0f, 30.0f, 1.5f, 0xFF202070, buffer);
	m_Print3D.Flush();
```

In this function, we can also see two more variables that were previously initialized at the *InitView* function. "*AlActivation*" and "*snooze*" are used to record when the alarm is activated and to keep track of the user's preferences about the snooze delay respectively. The rendering of two objects is also depended on *AlActivation*. One is located in the main menu and the other one is found in the alarm section. Both are used to signify the use of the alarm clock. The value of the snooze variable is used to determine the string that will be projected under the snooze button.

```
else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y<0.16 )
{
	AlActivation = 1;
}
else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.16 && y<0.32 )
{
	AlActivation = 0;
}
.
.
.
.
switch(snooze)
{
case 0:
	m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "snooze off");
	m_Print3D.Flush();
	break;
```

```
case 1:
       m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "1 min");
       m_Print3D.Flush();
       break;
case 2:
       m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "3 min");
       m_Print3D.Flush();
       break;
case 3:
       m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "5 min");
       m_Print3D.Flush();
       break;
case 4:
       m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "10 min");
       m_Print3D.Flush();
       break;

}
```

3.13: Messages implementation

In the main part of the messages section, the thing that worths attention is the flag "*sent*" which is triggered when the user is sending a message. In this case we have a counter which increments by one, and after we transform this integer to a string, we print it out next to "sent messages". That way we keep track of how many messages the user has sent and at the same time we project it on screen.

```
if (send==1)
{
       send=0;
       incr++;        // increments the sent messages vakue
}

itoa(incr,bu,10);
m_Print3D.Print3D(114.0f, 19.0f, 0.4f, 0xFF307030, bu);
m_Print3D.Print3D(113.0f, 19.0f, 0.4f, 0xFF307030, "(     )");
m_Print3D.Print3D(34.0f,  5.0f, 0.4f, 0xFF3010070, "Create Message");
m_Print3D.Print3D(35.0f, 19.0f, 0.4f, 0xFF307030, "Incoming Messages");
m_Print3D.Print3D(35.0f, 32.0f, 0.4f, 0xFF902070, "Back");
m_Print3D.Print3D(95.0f,  5.0f, 0.4f, 0xFF307030, "email");
m_Print3D.Print3D(95.0f, 19.0f, 0.4f, 0xFF902070, "Sent messages");
m_Print3D.Print3D(95.0f, 32.0f, 0.4f, 0xFF307030, "Draft");
m_Print3D.Flush();
```

The inner part of the messages, which is the creation part, is more complex. First we call a function which changes the value of the variable "*MesNum*" depending on the user tap on the keyboard. Then we decide which character to display on screen based on the value of this variable.

The main issue here is the implementation of the text display in different lines, as a problem appears when each line reaches the end of the screen. The tool function of IMG that displays text, takes as an argument only the coordinate from where the text will begin. Other than that, it cannot realise when we are exceeding the screen's limits, or if we are printing messages above other rendered objects. That is up to the developer to manage and control.

The solution to this problem comes as following. We create as many arrays as many lines we want the user to be able to use. In our case this number is three arrays, and so three lines of text. Then, we figure out how many characters fit in one line in our screen, by making as many rendering trials as we need, and thereafter we create a counter to be able to calculate the number of characters the user has typed. In our screen, the selected font allows 46 characters per line before to get out of the screen's dimensions. So, we keep track of the counter's value, and when this exceeds the 46th character, we pass the next coming character to the second array which will be projected at the next line. The same applies for the second and third line.

```
MesPad();

switch(MesNum)
{
case 0:
       if(counter<46)                        // to avoid text overflow
       strcat(text,"Q");                     // add character to first line of text
       else if(counter>=46 && counter<92){strcat(text1,"Q");}  // if 1st line is
                                                              full add to the 2nd one
       else if(counter>=92 &&counter<138){strcat(text2,"Q");}  // if 2nd line is
                                                              full add to the 3rd one
       counter++;                                               // increment counter
       break;
case 1:
       if(counter<46)
       strcat(text,"W");
       else if(counter>=46 && counter<92){strcat(text1,"W");}
       else if(counter>=92 &&counter<138){strcat(text2,"W");}
       counter++;
       break;
case 2:
       if(counter<46)
       strcat(text,"E");
       else if(counter>=46 && counter<92){strcat(text1,"E");}
       else if(counter>=92 &&counter<138){strcat(text2,"E");}
       counter++;
       break;
```

For clearing up the text written, we pass a NULL value to the last character in the array. Depending on which array we perform the erasure, we have to be careful. Because the counter value with which we keep track of the amount of the characters, contains the total amount of the characters and not necessarily the right array position. That applies only for the first array that is connected with the first line of text. In the case of the other two arrays, we have to subtract 46 and 92 for the second and third array respectively, if we are to pass the counter as a pointer to the array.

```
        // Erase the last element of the text
else if(erase==1)
{
        erase=0;
        if(counter<46)
        {
                text[counter]=NULL;
        }
        else if(counter>=46 && counter<92)
        {
                text1[counter-46]=NULL;  // Subtract elements of the 1st line array
        }
        else if(counter>=92 && counter<138)
        {
                text2[counter-92]=NULL; // Subtract elements of the 1st line and
                                           second line array
        }
        counter--;                 // Subtract one from counter
        Sleep(30);
}
```

3.14: Performance tips

In handheld devices like smartphones, tablets and navigators, performance is a constant target to aim for. Because of their small size, it is difficult for these devices to provide powerful hardware to work on, while at the same time, users' demands to get what they ask from the product that very moment is constantly growing. So I consider it important to discuss some ways of how we can achieve better performance through our programming, some of which I came across while implementing this project.

To start with, when we know where our programme will be applicable, it is important to understand our target device. Knowing the GPU architecture is essential, but not enough on its own. Every System on Chip (SoC) has different characteristics such as CPU efficiency or memory bandwidth. Moreover, every device has difference operating system, or even uses different features of the same operating system comparing to other devices, and so it is possible that background processes conflict with our application and slow the system down. The Tile Based Differed Rendering Technique already provides a good starting point for performance enhancement as it restricts memory access to a minimum. Besides that, it is important to know the specifications of our target device and reading through the manufacturer's handbook may prove valuable.

Something else that developers should always have in mind is keeping state changes within a program to a minimum. That means that changing shaders during the rendering state should be avoided because it costs a lot when it comes to performance. Note here, Hidden Surface Removal has already helped us towards achieving this, because it allows us to sort objects by render state and not by depth. By the same reasoning, if a group of objects is static, it should be grouped into a single mesh and be rendered at the same state.

There are even more simple solutions to keep performance high, like for example keeping geometry complexity low. We don't need to draw details and use polygons excessively, that will never be seen on screen. Likewise, it is a waste to use

many polygons to draw a quad, since it can always be done simply and quickly. Furthermore, always use Vertex Buffer Objects to store your vertex data as they are handled by the driver and there is no need to copy arrays from the client side at each draw call. It is wise to store your vertex data interleaved, meaning that it is better to put all data for every vertex in a block, and then another block for the following vertex and so on, into a single array, instead of using many arrays each one for one kind of data, for example an array of normals and a different array of coordinates. So, store all your vertices data in the same Vertex Buffer object – unless there is a mixture or static and dynamic data – and when possible put the objects that always appear together in the same VBO.

Another thing to remember is that vertex shaders always expect attributes to be of type float. Any other type should be transformed into float, and that task falls on USSE. As a result, using other type of variable attributes burdens the USSE and slows down the application. Precision selection in shaders is also an important factor that can boost performance. In general, the best way to arrive at the right precision is to start from lowp and then gradually increase the precision until the desired result comes upon your screen. Other than that, there are some common rules to follow when it comes to choose precision. Highp precision should be used for vertex position calculations as well as for world, model and projection matrices. Medium precision offers a performance improvement comparing to the highp and it should be used when possible instead of hiphp precision. Finally lowp precision is useful for representing colours and low precision textures.

Texture optimization can also provide solutions when it comes to performance. Except very few cases, we have to remember that larger textures do not offer better quality. There is no point in putting a 512x512 size texture in an 8x8 area of screen. As we mentioned in the beginning of this presentation, texture compression can be a life saver. Using PVRTC compression reduces memory footprint of a loaded texture and thus it does not burden the system memory bandwidth. The difference with common compressed image file format – bitmap, jpg, png – is that these techniques actually reduce storage footprint rather than memory footprint. In simple words, that means than on run time the picture will unzip and occupy a lot of RAM.

Conclusion / Future *work*

After completing my diploma thesis, and having the opportunity to review and reexamine it from the beginning while writing this report, I can finally have an integrated opinion on the result of this work. As mentioned in the beginning, the target of this thesis was not to implement as many as possible features for a smartphone interface. So it goes without saying that moving towards this direction, without importing much of anything innovative, it is quite easy to complete the rest of the features that were not implemented.

What is more interesting though at this time is to see where this work provides a good reference for future similar work and what it could be done differently in order to get a better result.

As we have already repeatedly stated, performance is handheld devices is an important factor which determines the quality of our work. In this case, a combination of the performance tips previously provided along with various hints described in the

code presentation makes this project a good example of high performance implementation when it comes to graphics. That would be more obvious if we would load it in a smartphone, which is quite easy as the application is meant to be portable. Moreover, the overall organization of the code is quite efficient as it improves readability and provides a good structure example. Another set of features that worth notice is the SDK tools and functions as their use simplifies a lot of implementation parts that otherwise would be quite difficult to deal with. Finally, the simplistic picking technique, according to the author's opinion, could provide a viable solution even in a 3D interface smartphone environment, as long as there is no overlapping between objects.

On the other hand there are some parts that in a real mobile device would be differently handled. The most obvious part is texture selection. Probably this particular theme is outside of the scope of this project, but it is inevitable to mention that the selected textures are just a decent effort of finding appropriate images online. In a modern retail device, most of the textures used, are designed by professionals in graphics software used for animation. In addition to this, most devices have fancy graphics features, like trembling objects when selected or gradually fading away menus that could be added to this project also. Another technique that can be questioned is the one used for sliding. Besides the fact that is kind of unusual, it causes many consequative renderings of the image – that means state changes – which is not clear if they would diminish the performance of an embedded device. The usual practice for similar procedures is texture sliding, but since our rendering context is simple, there is a chance that it would not make a real difference. Finally, I think it would be very interesting to try and load this interface in a smartphone in order to see its behavior in the real environment for which it is intended.

To conclude, this project can provide a solid start up to anyone who wants to meet the world of graphics as it did for me too, while at the same time it has aspects which could be further enhanced and examined in order to give a fully functional, portable interface of any smartphone.

Sources-Bibliography

- OpenGL ES 2.0 Programming Guide, Aaftab Munshi, Dan Ginsburg, Dave Shreiner, Addison-Wesley publishing
- C++ Primer, 5th Edition, Stanley B. Lippman, Josee LaJoie, Barbara E. Moo, Addison-Wesley publishing
- http://www.imgtec.com/
- IMG PowerVR forum (http://forum.imgtec.com/categories/powervr-graphics)
- Imagination technologies SDK (available at www.imgtec.com)
- http://www.khronos.org
- http://db-in.com/blog/category/opengl/Wikipedia
- http://www.opengl-tutorial.org/
- http://en.wikipedia.org/
- http://www.gamedev.net/page/index.html
- http://schabby.de/

Appendix

This appendix was compiled regarding certain explanation on some parts of the code which could possibly raise questions. These details were not referred in the main chapter where the code was presented either because the author thought that it would not be appropriate to spend time in tiring details or because they had no relevance with the rest of the parts that were analysed.

I. At the inclusion of the libraries needed in our program, one can notice the following part of code that could possibly raise questions.

```
#if defined(__APPLE__) && defined (TARGET_OS_IPHONE)
#import <OpenGLES/ES2/gl.h>
#import <OpenGLES/ES2/glext.h>
#else
#include <GLES2/gl2.h>
#endif
```

The macros *APPLE* and *TARGET_OS_IPHONE* are defined in the source code of *PVRShell* and help our application become portable in different kind of devices. The IMG framework recognizes the device that is loaded onto, and accordingly decides to load the appropriate libraries so that the application will run unabruptly. Since we run the program on a personal computer the libraries gl.h and glext.h will not be included and instead the gl2.h will be used.

II. Reading the names of many variables, you will see that some initials before the actual name of the variable. These help us keep track of the type of the variable so that we are not obligated to always search the top lines of the code. So for example “ui” shows that the variable is an unsigned int and “f” shows that the variable is a float. This technique is usual at very big programs where the amount of lines of code and the large number of declared variables makes it difficult to remember all details.

III. Another point that requires further analysis is the declaration of *Print3D*. *CPVRTPrint3D* is declared at the source code with the use of “*typedef*”. *Typedef* is a keyword used in C and C++ in order to form complex types from more basic ones and assign simple names to the combinations. In this case this method is followed for a similar reason. Since every compiler has different convention when it comes to naming variables, setting up a define in our source code saves us time, in a way that if for example a gcc compiler does not accept this definition of ours, we can go and apply changes only to the definition of our source code and not to the whole of our program. That is very important for the application as our goal is always to be portable and able to cooperate with many different platforms with minimum changes.

IV. As it is also mentioned in the comments on the code, *Print3D* class needs to know the viewport dimensions and whether the text will be rotated or not, in order to display text. We can get this information using the SDK tools as shown below.

```
bool bRotate = PVRShellGet(prefIsRotated) && PVRShellGet(prefFullScreen);

if(m_Print3D.SetTextures(0, PVRShellGet(prefWidth), PVRShellGet(prefHeight),
bRotate) != PVR_SUCCESS)

	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot initialise Print3D\n");
		return false;
	}
```

V. In the beginning of our initialization function, we use *CPVRTResourceFile* resource file helper class. Resource files can be placed on disk next to the executable or in a platform dependent read path. We need to tell the class where that read path is. Additionally, it is possible to wrap files into cpp modules and link them directly into the executable. In this case no path will be used. Files on disk will override "memory files".

Code

```
#include <stdio.h>
#include <cstdlib>    //for itoa() Func
#include <string.h>
#include <math.h>
#include <windows.h> // for Sleep() Func


#if defined(__APPLE__) && defined (TARGET_OS_IPHONE)
#import <OpenGLES/ES2/gl.h>
#import <OpenGLES/ES2/glext.h>
#else
#include <GLES2/gl2.h>
#endif

#include "PVRShell.h"
#include <OGLES2Tools.h>



#define VERTEX_ARRAY   0
#define TEXCOORD_ARRAY 1
#define NORMAL_ARRAY   2
// Screensaver Timer in ms
#define TIMER      20000
// Changes viewport coordinates defining the sliding speed
#define SCREEN_SPEED   3

// Size of the texture we create
#define TEX_SIZE     128

// Pvr texture files
const char c_aTextureFile[] = "agenda.pvr";    //0
const char c_bTextureFile[] = "alarm.pvr";
const char c_cTextureFile[] = "camera.pvr";
```

```
const char c_dTextureFile[] = "contacts.pvr";
const char c_eTextureFile[] = "files.pvr";
const char c_gTextureFile[] = "messages.pvr";  //5
const char c_hTextureFile[] = "music.pvr";
const char c_iTextureFile[] = "tools.pvr";
const char c_jTextureFile[] = "back.pvr";
const char c_kTextureFile[] = "fun.pvr";
const char c_lTextureFile[] = "google.pvr";   //10
const char c_mTextureFile[] = "sim.pvr";
const char c_nTextureFile[] = "youtube.pvr";
const char c_oTextureFile[] = "phonecall.pvr";
const char c_pTextureFile[] = "pad.pvr";
const char c_qTextureFile[] = "IMGlogo.pvr";
const char c_rTextureFile[] = "YesNo.pvr";
const char c_sTextureFile[] = "erase.pvr";
const char c_tTextureFile[] = "tick.pvr";
const char c_uTextureFile[] = "PlusMinus.pvr";
const char c_vTextureFile[] = "notification.pvr"; //20
const char c_wTextureFile[] = "off.pvr";
const char c_xTextureFile[] = "on.pvr";
const char c_yTextureFile[] = "snooze.pvr";
const char c_zTextureFile[] = "bell.pvr";
const char c_aaTextureFile[] = "CreateMessage.pvr";
const char c_abTextureFile[] = "Draft.pvr";
const char c_acTextureFile[] = "email.pvr";
const char c_adTextureFile[] = "Incoming.pvr";
const char c_aeTextureFile[] = "Outgoing.pvr";
const char c_afTextureFile[] = "MesPad.pvr";

// Matrix to store displayed call-num
char NumToCall[20]={};
// Matrices to store message text
char text[46]={};
char text1[46]={};
char text2[46]={};
// Empty matrix used for clearing call number
char empty[20]={};
// Variables used for screen sliding
int kapa=-350;
```

```
int lamda = 0;
// Matrices to store alarm set time
char bu[10],buf[10],buff[10],buffe[10],buffer[10];

// Default projection and modelview matrix
float aPMVMatrix[] =
	{
		1,	0,	0,	0,
		0,	1,	0,	0,
		0,	0,	1,	0,
		0,	0,	0,	1
	};

			float aModelViewIT[] =
	{
		1,	0,	0,
		0,	1,	0,
		0,	0,	1
	};

class OGLES2Texturing : public PVRShell
{
	// Print3D class used to display text
	CPVRTPrint3D	m_Print3D;

	// The vertex and fragment shader OpenGL handles
	GLuint m_uiVertexShader, m_uiFragShader;

	// The program object containing the 2 shader objects
	GLuint m_uiProgramObject;

	// Texture handle
	GLuint m_uiTexture[31];
	// Timer handle
	unsigned long	m_ulStartTime;
	unsigned long	StartButton;

	// VBO handle
	GLuint m_ui32Vbo[26];
```

```
        unsigned int m_ui32VertexStride;
        // Rotation angle
        float  m_fAngle;

        // Handle of mouse coordinates
        float *vec2PointerLocation;
        float x,y;
        // Handle of mouse click
        GLuint UpDown;
        GLuint LeftRight;
        GLuint pressed;
        GLuint mes;
        GLuint send;
        GLuint incr;
        // Handle of mouse click (call pad)
        GLuint PrNum;
        GLuint MesNum;
        // Handle of mouse click (alarm)
        GLuint AlActivation;
        GLuint snooze;
        GLint rmin;
        GLint lmin;
        GLint rhour;
        GLint lhour;
        // Handle of text overflow
        GLuint counter;
        // Dial handle
        GLuint dial;
        GLuint erase;



public:
        // Default PVRShell functions
        virtual bool InitApplication();
        virtual bool InitView();
```

```
	virtual bool ReleaseView();
	virtual bool QuitApplication();
	virtual bool RenderScene();
	// Menu functions
	virtual void Main_Menu();
	virtual void AlarmClock();
	virtual void Pad();
	virtual void NumPad();
	virtual void ScreenSaver();
	virtual void Messages();
	virtual void WriteMes();
	virtual void MesPad();
};


/*!****************************************************************************
 @Function		InitApplication
 @Return		bool		true if no error occured
 @Description	Code in InitApplication() will be called by PVRShell once per
				run, before the rendering context is created.
				Used to initialize variables that are not dependant on it
				(e.g. external modules, loading meshes, etc.)
				If the rendering context is lost, InitApplication() will
				not be called again.
******************************************************************************/
bool OGLES2Texturing::InitApplication()
{
	/*
		CPVRTResourceFile is a resource file helper class. Resource files can
		be placed on disk next to the executable or in a platform dependent
		read path. We need to tell the class where that read path is.
		Additionally, it is possible to wrap files into cpp modules and
		link them directly into the executable. In this case no path will be
		used. Files on disk will override "memory files".
	*/

	// Get and set the read path for content files
	CPVRTResourceFile::SetReadPath((char*)PVRShellGet(prefReadPath));
```

```
	// Get and set the load/release functions for loading external files.
	// In the majority of cases the PVRShell will return NULL function pointers implying that
	// nothing special is required to load external files.

	CPVRTResourceFile::SetLoadReleaseFunctions(PVRShellGet(prefLoadFileFunc), PVRShellGet(prefReleaseFileFunc));

	// Set window's dimensions
	PVRShellSet(prefWidth,1200);
	PVRShellSet(prefHeight,1500);
	// Initialize rotation angle
	m_fAngle = 0;

	return true;
}

/*!****************************************************************************
 @Function		QuitApplication
 @Return		bool		true if no error occured
 @Description	Code in QuitApplication() will be called by PVRShell once per
				run, just before exiting the program.
				If the rendering context is lost, QuitApplication() will
				not be called.
******************************************************************************/
bool OGLES2Texturing::QuitApplication()
{
	return true;
}

/*!****************************************************************************
 @Function		InitView
 @Return		bool		true if no error occured
 @Description	Code in InitView() will be called by PVRShell upon
				initialization or after a change in the rendering context.
				Used to initialize variables that are dependant on the rendering
				context (e.g. textures, vertex buffers, etc.)
******************************************************************************/
bool OGLES2Texturing::InitView()
{
	/*
```

```
		Initialize the textures used by Print3D.
		To properly display text, Print3D needs to know the viewport dimensions
		and whether the text should be rotated. We get the dimensions using the
		shell function PVRShellGet(prefWidth/prefHeight). We can also get the
		rotate parameter by checking prefIsRotated and prefFullScreen.
	*/
	bool bRotate = PVRShellGet(prefIsRotated) && PVRShellGet(prefFullScreen);

	if(m_Print3D.SetTextures(0, PVRShellGet(prefWidth), PVRShellGet(prefHeight), bRotate) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot initialise Print3D\n");
		return false;
	}

	// Fragment and vertex shaders code
	const char* pszFragShader = "\
		uniform sampler2D sampler2d;\
		varying mediump float varDot;\
		varying mediump vec2  varCoord;\
		void main (void)\
		{\
		    gl_FragColor.rgb = texture2D(sampler2d,varCoord).rgb * varDot;\
		    gl_FragColor.a = 1.0; \
		}";

	const char* pszVertShader = "\
		attribute highp vec4  myVertex;\
		attribute mediump vec3       myNormal;\
		attribute mediump vec4       myUV;\
		uniform mediump mat4  myPMVMatrix;\
		uniform mediump mat3  myModelViewIT;\
		uniform mediump vec3  myLightDirection;\
		varying mediump float varDot;\
		varying mediump vec2  varCoord;\
		void main(void)\
		{\
			gl_Position = myPMVMatrix * myVertex;\
			varCoord = myUV.st;\
			mediump vec3 transNormal = myModelViewIT * myNormal;\
```

```
                        varDot = max( dot(transNormal, myLightDirection), 0.0 );\
                }";


        // Create the fragment shader object
        m_uiFragShader = glCreateShader(GL_FRAGMENT_SHADER);

        // Load the source code into it
        glShaderSource(m_uiFragShader, 1, (const char**)&pszFragShader, NULL);

        // Compile the source code
        glCompileShader(m_uiFragShader);

        // Check if compilation succeeded
        GLint bShaderCompiled;
    glGetShaderiv(m_uiFragShader, GL_COMPILE_STATUS, &bShaderCompiled);
        if (!bShaderCompiled)
        {
                // An error happened, first retrieve the length of the log message
                int i32InfoLogLength, i32CharsWritten;
                glGetShaderiv(m_uiFragShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);

                // Allocate enough space for the message and retrieve it
                char* pszInfoLog = new char[i32InfoLogLength];
         glGetShaderInfoLog(m_uiFragShader, i32InfoLogLength, &i32CharsWritten, pszInfoLog);

                /*
                        Displays the message in a dialog box when the application quits
                        using the shell PVRShellSet function with first parameter prefExitMessage.
                */
                char* pszMsg = new char[i32InfoLogLength+256];
                strcpy(pszMsg, "Failed to compile fragment shader: ");
                strcat(pszMsg, pszInfoLog);
                PVRShellSet(prefExitMessage, pszMsg);

                delete [] pszMsg;
                delete [] pszInfoLog;
                return false;
        }
```

```
    // Loads the vertex shader in the same way
    m_uiVertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(m_uiVertexShader, 1, (const char**)&pszVertShader, NULL);
    glCompileShader(m_uiVertexShader);
glGetShaderiv(m_uiVertexShader, GL_COMPILE_STATUS, &bShaderCompiled);
    if (!bShaderCompiled)
    {
            int i32InfoLogLength, i32CharsWritten;
            glGetShaderiv(m_uiVertexShader, GL_INFO_LOG_LENGTH, &i32InfoLogLength);
            char* pszInfoLog = new char[i32InfoLogLength];
     glGetShaderInfoLog(m_uiVertexShader, i32InfoLogLength, &i32CharsWritten, pszInfoLog);
            char* pszMsg = new char[i32InfoLogLength+256];
            strcpy(pszMsg, "Failed to compile vertex shader: ");
            strcat(pszMsg, pszInfoLog);
            PVRShellSet(prefExitMessage, pszMsg);

            delete [] pszMsg;
            delete [] pszInfoLog;
            return false;
    }

    // Create the shader program
m_uiProgramObject = glCreateProgram();

    // Attach the fragment and vertex shaders to it
glAttachShader(m_uiProgramObject, m_uiFragShader);
glAttachShader(m_uiProgramObject, m_uiVertexShader);

    // Bind the custom vertex attribute "myVertex" to location VERTEX_ARRAY
glBindAttribLocation(m_uiProgramObject, VERTEX_ARRAY, "myVertex");
    // Bind the custom vertex attribute "myUV" to location TEXCOORD_ARRAY
glBindAttribLocation(m_uiProgramObject, TEXCOORD_ARRAY, "myUV");

    // Link the program
glLinkProgram(m_uiProgramObject);

    // Check if linking succeeded in the same way we checked for compilation success
GLint bLinked;
```

```
    glGetProgramiv(m_uiProgramObject, GL_LINK_STATUS, &bLinked);

	if (!bLinked)
	{
		int i32InfoLogLength, i32CharsWritten;
		glGetProgramiv(m_uiProgramObject, GL_INFO_LOG_LENGTH, &i32InfoLogLength);
		char* pszInfoLog = new char[i32InfoLogLength];
		glGetProgramInfoLog(m_uiProgramObject, i32InfoLogLength, &i32CharsWritten, pszInfoLog);

		char* pszMsg = new char[i32InfoLogLength+256];
		strcpy(pszMsg, "Failed to link program: ");
		strcat(pszMsg, pszInfoLog);
		PVRShellSet(prefExitMessage, pszMsg);
		delete [] pszMsg;
		delete [] pszInfoLog;
		return false;
	}

	// Actually use the created program
	// glUseProgram(m_uiProgramObject);

	// Sets the sampler2D variable to the first texture unit
	glUniform1i(glGetUniformLocation(m_uiProgramObject, "sampler2d"), 0);

	// Sets the clear color
	glClearColor(0.99f, 0.99f, 0.99f, 1.0f);

	/*
		Loads the texture using the tool function PVRTTextureLoadFromPVR.
		The first parameter is the name of the file and the
		second parameter returns the resulting texture handle.
		The third parameter is a CPVRTString for error message output.
		This function can also be used to conveniently set the filter modes. If
		those parameters are not given, OpenGL ES defaults are used.
		Setting a mipmap filter on a mipmap-less texture will result in an error.
	*/

////////////////////// Agenda 0
	if(PVRTTextureLoadFromPVR(c_aTextureFile, &m_uiTexture[0]) != PVR_SUCCESS)
```

```
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the first texture\n");
		return false;
	}
	//glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
	//glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

//////////////////////  Alarm 1
		if(PVRTTextureLoadFromPVR(c_bTextureFile, &m_uiTexture[1]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the second texture\n");
		return false;
	}

//////////////////   Camera 2
	if(PVRTTextureLoadFromPVR(c_cTextureFile, &m_uiTexture[2]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the third texture\n");
		return false;
	}

//////////////////   Contacts 3
		if(PVRTTextureLoadFromPVR(c_dTextureFile, &m_uiTexture[3]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the fourth texture\n");
		return false;
	}

//////////////////   Files 4
	if(PVRTTextureLoadFromPVR(c_eTextureFile, &m_uiTexture[4]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the fifth texture\n");
		return false;
	}

//////////////////   Messages 6
	if(PVRTTextureLoadFromPVR(c_gTextureFile, &m_uiTexture[5]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 6th texture\n");
```

```
                return false;
        }

////////////////////   Music 7
        if(PVRTTextureLoadFromPVR(c_hTextureFile, &m_uiTexture[6]) != PVR_SUCCESS)
        {
                PVRShellSet(prefExitMessage, "ERROR: Cannot load the 7th texture\n");
                return false;
        }

////////////////////   Tools 8
        if(PVRTTextureLoadFromPVR(c_iTextureFile, &m_uiTexture[7]) != PVR_SUCCESS)
        {
                PVRShellSet(prefExitMessage, "ERROR: Cannot load the 8th texture\n");
                return false;
        }

////////////////////   Back 9
        if(PVRTTextureLoadFromPVR(c_jTextureFile, &m_uiTexture[8]) != PVR_SUCCESS)
        {
                PVRShellSet(prefExitMessage, "ERROR: Cannot load the 9th texture\n");
                return false;
        }


////////////////////  fun 10
        if(PVRTTextureLoadFromPVR(c_kTextureFile, &m_uiTexture[9]) != PVR_SUCCESS)
        {
                PVRShellSet(prefExitMessage, "ERROR: Cannot load the 10th texture\n");
                return false;
        }

        ////////////////////  google 11
        if(PVRTTextureLoadFromPVR(c_lTextureFile, &m_uiTexture[10]) != PVR_SUCCESS)
        {
                PVRShellSet(prefExitMessage, "ERROR: Cannot load the 11th texture\n");
                return false;
        }
        ////////////////////// sim
```

```
	if(PVRTTextureLoadFromPVR(c_mTextureFile, &m_uiTexture[11]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 12th texture\n");
		return false;
	}
		////////////////////  youtube
	if(PVRTTextureLoadFromPVR(c_nTextureFile, &m_uiTexture[12]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 13th texture\n");
		return false;
	}
	////////////////////  phonecall
	if(PVRTTextureLoadFromPVR(c_oTextureFile, &m_uiTexture[13]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 14th texture\n");
		return false;
	}
		////////////////////  pad
	if(PVRTTextureLoadFromPVR(c_pTextureFile, &m_uiTexture[14]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 15th texture\n");
		return false;
	}
		////////////////////  ImgLogo
	if(PVRTTextureLoadFromPVR(c_qTextureFile, &m_uiTexture[15]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 16th texture\n");
		return false;
	}
	    ////////////////////  YesNo
	if(PVRTTextureLoadFromPVR(c_rTextureFile, &m_uiTexture[16]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 17th texture\n");
		return false;
	}
		////////////////////  Erase
	if(PVRTTextureLoadFromPVR(c_sTextureFile, &m_uiTexture[17]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 18th texture\n");
```

```
		return false;
	}
		//////////////////  tick
	if(PVRTTextureLoadFromPVR(c_tTextureFile, &m_uiTexture[18]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 19th texture\n");
		return false;
	}
	///////////  PlusMinus
		if(PVRTTextureLoadFromPVR(c_uTextureFile, &m_uiTexture[19]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 20th texture\n");
		return false;
	}
		//////////////////  notification
	if(PVRTTextureLoadFromPVR(c_vTextureFile, &m_uiTexture[20]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 21st texture\n");
		return false;
	}
	////////////////////   off
		if(PVRTTextureLoadFromPVR(c_wTextureFile, &m_uiTexture[21]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 22nd texture\n");
		return false;
	}
		//////////////////  on
	if(PVRTTextureLoadFromPVR(c_xTextureFile, &m_uiTexture[22]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 23rd texture\n");
		return false;
	}
		//////////////////  snooze
		if(PVRTTextureLoadFromPVR(c_yTextureFile, &m_uiTexture[23]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 24th texture\n");
		return false;
	}
		//////////////////  bell
```

```
		if(PVRTTextureLoadFromPVR(c_zTextureFile, &m_uiTexture[24]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 25th texture\n");
		return false;
	}
		//////////////////  CreateMessage
		if(PVRTTextureLoadFromPVR(c_aaTextureFile, &m_uiTexture[25]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 26th texture\n");
		return false;
	}
		//////////////////  Draft
		if(PVRTTextureLoadFromPVR(c_abTextureFile, &m_uiTexture[26]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 27th texture\n");
		return false;
	}
		//////////////////  email
		if(PVRTTextureLoadFromPVR(c_acTextureFile, &m_uiTexture[27]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 28th texture\n");
		return false;
	}
		//////////////////  Incoming
		if(PVRTTextureLoadFromPVR(c_adTextureFile, &m_uiTexture[28]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 29th texture\n");
		return false;
	}
		//////////////////  Outgoing
		if(PVRTTextureLoadFromPVR(c_aeTextureFile, &m_uiTexture[29]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 30th texture\n");
		return false;
	}
		//////////////////  MesPad
		if(PVRTTextureLoadFromPVR(c_afTextureFile, &m_uiTexture[30]) != PVR_SUCCESS)
	{
		PVRShellSet(prefExitMessage, "ERROR: Cannot load the 31th texture\n");
```

```
        return false;
    }

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);


    glGenBuffers(26, &m_ui32Vbo[0]);
    m_ui32VertexStride = 8 * sizeof(GLfloat); // 3 floats for the pos, 2 for the UVs , 3 for the normals

    // objects: 0   1  2  3
    //          4   5  6  7
    //          8   9 10 11
    //          12 13 14 15
    //          16 17 18 19
    //          20 21 22 23
                                                    // Position         UVs         Normals
    GLfloat zerVertices[] = {-0.51f,0.70f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                             -0.96f,0.70f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                             -0.96f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 0
                             -0.51f,0.70f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                             -0.96f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             -0.51f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, zerVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);


    GLfloat oneVertices[] = { 0.02f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                              0.02f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                              0.47f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,        // #1 PlusMinus obj
                              0.47f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                              0.02f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                              0.47f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[1]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, oneVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat twoVertices[] = {0.02f,0.70f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
```

```
                        0.47f,0.70f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                        0.47f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,                    // 2
                        0.02f,0.70f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                        0.47f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                        0.02f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f };
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[2]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, twoVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat thrVertices[] = {0.85f,0.925f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                        0.96f,0.925f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                        0.96f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,            // 3 // changed to small
                        0.85f,0.925f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                        0.96f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                        0.85f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f };
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[3]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, thrVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);


    GLfloat forVertices[] = {-0.51f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                         -0.96f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                         -0.96f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 4
                         -0.51f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                         -0.96f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.51f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[4]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, forVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);



    GLfloat fivVertices[] = {-0.02f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                        -0.47f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                        -0.47f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 5
                        -0.02f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                        -0.47f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                        -0.02f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[5]);
```

```
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , fivVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);


        GLfloat sixVertices[] = {0.02f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                 0.47f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                 0.47f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,                    // 6
                                 0.02f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                 0.47f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                 0.02f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f};
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[6]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, sixVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat sevVertices[] = { 0.51f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                  0.96f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                  0.96f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,        // 7
                                  0.51f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                  0.96f,0.66f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                  0.51f,0.66f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f};
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[7]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, sevVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat eitVertices[] = {-0.51f,0.02f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                 -0.96f,0.02f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                 -0.96f,0.32f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 8
                                 -0.51f,0.02f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                 -0.96f,0.32f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                 -0.51f,0.32f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f };
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[8]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , eitVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat ninVertices[] = {0.51f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                 0.51f,1.00f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                 0.96f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,        // #2 PlusMinus obj
                                 0.96f,1.00f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                 0.51f,0.36f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
```

```
                          0.96f,0.36f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[9]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , ninVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat tenVertices[] = { 0.02f,0.02f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                              0.47f,0.02f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                              0.47f,0.32f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,        // 10
                              0.02f,0.02f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                              0.47f,0.32f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                              0.02f,0.32f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[10]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, tenVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat eleVertices[] = { 0.51f,0.02f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                             0.96f,0.02f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                             0.96f,0.32f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,        // 11
                             0.51f,0.02f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                             0.96f,0.32f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.51f,0.32f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f};

    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[11]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, eleVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat twlVertices[] = {-0.51f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                            -0.96f,-0.32f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                            -0.96f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 12
                            -0.51f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                            -0.96f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                            -0.51f,-0.02f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[12]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, twlVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);


    GLfloat trnVertices[] = {-0.02f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                            -0.47f,-0.32f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
```

```
                        -0.47f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,        // 13
                        -0.02f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                        -0.47f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                        -0.02f,-0.02f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[13]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, trnVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat frnVertices[] = {0.02f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.47f,-0.02f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.47f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,                  // 14
                             0.02f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.47f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                             0.02f,-0.32f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f };
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[14]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, frnVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    GLfloat ffnVertices[] = {0.51f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.96f,-0.02f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.96f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,                  // 15
                             0.51f,-0.02f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             0.96f,-0.32f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                             0.51f,-0.32f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f };
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[15]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, ffnVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);


    GLfloat sxnVertices[] = {-0.51f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                             -0.96f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                             -0.96f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,        // 16
                             -0.51f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                             -0.96f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                             -0.51f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f};
    glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[16]);
    glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, sxnVertices, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
GLfloat svnVertices[] = {-0.02f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.47f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.47f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,        // 17
                         -0.02f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.47f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                         -0.02f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f};
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[17]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , svnVertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);


GLfloat etnVertices[] = {0.02f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                         0.47f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         0.47f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,        // 18
                         0.02f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                         0.47f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                         0.02f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f};
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[18]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, etnVertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

GLfloat nnnVertices[] = { 0.51f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                          0.96f,-0.36f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                          0.96f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,        // 19
                          0.51f,-0.36f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                          0.96f,-0.66f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                          0.51f,-0.66f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f};
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[19]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, nnnVertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

GLfloat tweVertices[] = {-0.51f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.96f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.96f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,        // 20
                         -0.51f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         -0.96f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                         -0.51f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f };
```

```
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[20]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , tweVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat tweoneVertices[] = {-0.02f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                    -0.47f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                    -0.47f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,        // 21
                                    -0.02f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                    -0.47f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
                                    -0.02f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f};
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[21]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride , tweoneVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat twetwoVertices[] = { 0.02f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.47f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.47f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,        // 22
                                    0.02f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.47f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                    0.02f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f};
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[22]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, twetwoVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat twethrVertices[] = { 0.51f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.96f,-0.70f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.96f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,        // 23
                                    0.51f,-0.70f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f,
                                    0.96f,-1.00f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                    0.51f,-1.00f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f};

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[23]);
        glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, twethrVertices, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        GLfloat padVertices[] = { 0.90f, 0.90f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                                 0.90f,-0.10f,0.0f,  1.0f,0.0f , 0.0f,0.0f,1.0f,
                                -0.90f,-0.10f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,        // pad
                                -0.90f,-0.10f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f,
```

```
                         0.90f, 0.90f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                        -0.90f, 0.90f,0.0f,  0.0f,1.0f , 0.0f,0.0f,1.0f};
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[24]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, padVertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);


GLfloat scrVertices[] = { 0.90f, 0.90f,0.0f,  1.0f,1.0f , 0.0f,0.0f,1.0f,
                         0.90f, 0.00f,0.0f,  1.0f,0.5f , 0.0f,0.0f,1.0f,  //screen saver obj
                        -0.90f, 0.00f,0.0f,  0.0f,0.5f , 0.0f,0.0f,1.0f,
                        -0.90f, 0.00f,0.0f,  0.0f,0.5f , 0.0f,0.0f,1.0f,
                         0.90f, 0.00f,0.0f,  1.0f,0.5f , 0.0f,0.0f,1.0f,
                        -0.90f,-0.90f,0.0f,  0.0f,0.0f , 0.0f,0.0f,1.0f };

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[25]);
glBufferData(GL_ARRAY_BUFFER, 6 * m_ui32VertexStride, scrVertices, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);



m_ulStartTime = PVRShellGetTime();  // Initialize StartTime
counter = 0;                        // initialize Counter
dial    = 0;                        // Initialize call button
erase   = 0;                        // Initialize call number erasure
UpDown  = 0;                        // Initialize central menu sliding
pressed = 0;                        // Initialize central menu button state
snooze  = 0;                        // Initialize snooze button state
rmin    = 0;                        // Initialize + - buttons state
lmin    = 0;
rhour   = 0;
lhour   = 0;
mes     = 0;                        // Initialize message screen state
send    = 0;                        // Initialize sent messages state
incr    = 0;                        // Initialize counter that keeps sent messages
LeftRight = 0;                      // Initialize message menu sliding
AlActivation = 0;                   // Initialize alarm state
StartButton  = 0;                   // Initialize screensaver timer
```

```
	return true;
}

/*!****************************************************************************
 @Function		ReleaseView
 @Return		bool		true if no error occured
 @Description	Code in ReleaseView() will be called by PVRShell when the
				application quits or before a change in the rendering context.
******************************************************************************/
bool OGLES2Texturing::ReleaseView()
{
	// Frees the textures
	glDeleteTextures(1, &m_uiTexture[0]);
	glDeleteTextures(1, &m_uiTexture[1]);
	glDeleteTextures(1, &m_uiTexture[2]);
	glDeleteTextures(1, &m_uiTexture[3]);
	glDeleteTextures(1, &m_uiTexture[4]);
	glDeleteTextures(1, &m_uiTexture[5]);
	glDeleteTextures(1, &m_uiTexture[6]);
	glDeleteTextures(1, &m_uiTexture[7]);
	glDeleteTextures(1, &m_uiTexture[8]);
	glDeleteTextures(1, &m_uiTexture[9]);
	glDeleteTextures(1, &m_uiTexture[10]);
	glDeleteTextures(1, &m_uiTexture[11]);
	glDeleteTextures(1, &m_uiTexture[12]);
	glDeleteTextures(1, &m_uiTexture[13]);
	glDeleteTextures(1, &m_uiTexture[14]);
	glDeleteTextures(1, &m_uiTexture[15]);
	glDeleteTextures(1, &m_uiTexture[16]);
	glDeleteTextures(1, &m_uiTexture[17]);
	glDeleteTextures(1, &m_uiTexture[18]);
	glDeleteTextures(1, &m_uiTexture[19]);
	glDeleteTextures(1, &m_uiTexture[20]);
	glDeleteTextures(1, &m_uiTexture[21]);
	glDeleteTextures(1, &m_uiTexture[22]);
	glDeleteTextures(1, &m_uiTexture[23]);
	glDeleteTextures(1, &m_uiTexture[24]);
	glDeleteTextures(1, &m_uiTexture[25]);
```

```
        glDeleteTextures(1, &m_uiTexture[26]);
        glDeleteTextures(1, &m_uiTexture[27]);
        glDeleteTextures(1, &m_uiTexture[28]);
        glDeleteTextures(1, &m_uiTexture[29]);
        glDeleteTextures(1, &m_uiTexture[30]);

        // Release Vertex buffer object.
        glDeleteBuffers(1, &m_ui32Vbo[0]);
        glDeleteBuffers(1, &m_ui32Vbo[1]);
        glDeleteBuffers(1, &m_ui32Vbo[2]);
        glDeleteBuffers(1, &m_ui32Vbo[3]);
        glDeleteBuffers(1, &m_ui32Vbo[4]);
        glDeleteBuffers(1, &m_ui32Vbo[5]);
        glDeleteBuffers(1, &m_ui32Vbo[6]);
        glDeleteBuffers(1, &m_ui32Vbo[7]);
        glDeleteBuffers(1, &m_ui32Vbo[8]);
        glDeleteBuffers(1, &m_ui32Vbo[9]);
        glDeleteBuffers(1, &m_ui32Vbo[10]);
        glDeleteBuffers(1, &m_ui32Vbo[11]);
        glDeleteBuffers(1, &m_ui32Vbo[12]);
        glDeleteBuffers(1, &m_ui32Vbo[13]);
        glDeleteBuffers(1, &m_ui32Vbo[14]);
        glDeleteBuffers(1, &m_ui32Vbo[15]);
        glDeleteBuffers(1, &m_ui32Vbo[16]);
        glDeleteBuffers(1, &m_ui32Vbo[17]);
        glDeleteBuffers(1, &m_ui32Vbo[18]);
        glDeleteBuffers(1, &m_ui32Vbo[19]);
        glDeleteBuffers(1, &m_ui32Vbo[20]);
        glDeleteBuffers(1, &m_ui32Vbo[21]);
        glDeleteBuffers(1, &m_ui32Vbo[22]);
        glDeleteBuffers(1, &m_ui32Vbo[23]);
        glDeleteBuffers(1, &m_ui32Vbo[24]);
        glDeleteBuffers(1, &m_ui32Vbo[25]);

        // Release Print3D Textures
        m_Print3D.ReleaseTextures();

        // Frees the OpenGL handles for the program and the 2 shaders
        glDeleteProgram(m_uiProgramObject);
```

```
	glDeleteShader(m_uiVertexShader);
	glDeleteShader(m_uiFragShader);
	return true;
}
/*!****************************************************************************
 @Function		RenderScene
 @Return		bool		true if no error occured
 @Description	Main rendering loop function of the program. The shell will
				call this function every frame.
				eglSwapBuffers() will be performed by PVRShell automatically.
				PVRShell will also manage important OS events.
				Will also manage relevent OS events. The user has access to
				these events through an abstraction layer provided by PVRShell.
******************************************************************************/
bool OGLES2Texturing::RenderScene()
{
	glUseProgram(m_uiProgramObject); // Reuse our shaders as Print3D uses its own shaders when called

	// Clears the color and depth buffer
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// Takes the mouse position
	if (NULL != (vec2PointerLocation = (float *)PVRShellGet(prefPointerLocation)))
    {
	 x = vec2PointerLocation[0];
		y = vec2PointerLocation[1];
	}

	// Time passed since last mouse click
	unsigned long ulCurrentTime = PVRShellGetTime() - m_ulStartTime;

	if(0 != PVRShellGet(prefButtonState))
	{
		m_ulStartTime = PVRShellGetTime(); // After mouse event reset CurrentTime
		Sleep(50);                    //  To avoid pressing any Main Menu buttons after ScreenSaver return
	}
	if(0 != PVRShellGet(prefButtonState) && x>0.9 && y<0.1)
	{
		UpDown=0;
```

```
	}
	else if(0 != PVRShellGet(prefButtonState) && x<0.1 && y>0.9)
	{
		UpDown=1;
	}

	if(ulCurrentTime>TIMER)		// If currentTime exceeds TIMER call ScreenSaver(in ms)
	{
		glViewport(0,0,600,750);
		ScreenSaver();
	}
	else
	{
		if(UpDown==0 && pressed!=7 && pressed!=2 && pressed!=8)
		{
			if(kapa>-350){kapa-=SCREEN_SPEED;} // Change viewport position if needed
			glViewport(0,kapa,600,1100);
			Main_Menu();
			// Depending on click position pressed takes the appropriate value
			if(     0 != PVRShellGet(prefButtonState) && x<0.25 && y<0.25                    )		{pressed=1;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y<0.25          )		{pressed=2;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.25 && y<0.5)		{pressed=3;}
			else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.25 && y<0.5           )		{pressed=4;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.5  && y<0.75          )		{pressed=5;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y>0.5 && y<0.75)		{pressed=6;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.75          )		{pressed=7;}
			else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.75                    )		{pressed=8;}

		}
		else if(UpDown==1 && pressed!=7 && pressed!=2 && pressed!=8)
		{
			if(kapa<0){kapa+=SCREEN_SPEED;} // Change viewport position if needed
			glViewport(0,kapa,600,1100);
			Main_Menu();
			// Depending on click position pressed takes the appropriate value
			if(     0 != PVRShellGet(prefButtonState) && x<0.25 && y<0.25                    )		{pressed=5;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y<0.25          )		{pressed=6;}
			else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.25 && y<0.5)		{pressed=7;}
			else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.25 && y<0.5           )		{pressed=8;}
```

```
		else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.5  && y<0.75         )		{pressed=9;}
		else if(0 != PVRShellGet(prefButtonState) && x<0.75 && x>0.5  && y>0.5 && y<0.75)		{pressed=10;}
		else if(0 != PVRShellGet(prefButtonState) && x<0.5  && x>0.25 && y>0.75         )		{pressed=11;}
		else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.75                   )		{pressed=12;}

	}
	else if(pressed==7)
	{
		glViewport(0,0,600,750);
		Pad();
	}
	else if(pressed==2)
	{
		glViewport(0,0,600,750);
		AlarmClock();
	}
	else if(pressed==8 && mes==0)
	{
		if(0 != PVRShellGet(prefButtonState) && x>0.95 && y>0.3 && y<0.6)
		{
			LeftRight=1;
		}
		else if(0 != PVRShellGet(prefButtonState) && x<0.05 && y>0.3 && y<0.6)
		{
			LeftRight=0;
		}
		switch(LeftRight)
		{
		case 0:
			if(lamda<0){lamda+=SCREEN_SPEED;}  // Change viewport position if needed
			glViewport(lamda,-630,1160,1350);
			Messages();
			break;
		case 1:
			if(lamda>-580){lamda-=SCREEN_SPEED;} // Change viewport position if needed
			glViewport(lamda,-630,1160,1350);
			Messages();
			break;
		}
```

```
		}
		else if(pressed==8 && mes==1)
		{
			glViewport(0,0,600,750);
			WriteMes();
		}
	}
	return true;

}

/*!****************************************************************************
 @Function		NewDemo
 @Return		PVRShell*		The demo supplied by the user
 @Description	 This function must be implemented by the user of the shell.
		The user should return its PVRShell object defining the
		behaviour of the application.
******************************************************************************/

PVRShell* NewDemo()
{
	return new OGLES2Texturing();
}



/*************************************************
 @Function		Main_Menu
 @Return		-
 @Description	This function is called in RenderScene Function
		in order to render the main menu objects. Textures
		rendered differ depending on the user's mouse click
*************************************************/
void OGLES2Texturing::Main_Menu(){

	/*
		Bind the projection model view matrix (PMVMatrix) to the
		corresponding uniform variable in the shader.
```

```
                This matrix is used in the vertex shader to transform the vertices.
        */
        int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
        glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
        /*
                Bind the Model View Inverse Transpose matrix to the shader.
                This matrix is used in the vertex shader to transform the normals.
        */
        zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
        glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);

        // Bind the Light Direction vector to the shader
        zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
        glUniform3f( zLocation, 0, 0, 1);

        // Bind the VBO
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
        // Pass the vertex data
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        // Pass the texture coordinates data depending on the value of "pressed" variable
        if(pressed==1)
        {
                glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
        }
        else
        {
                glBindTexture( GL_TEXTURE_2D, m_uiTexture[0]);
        }
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        // Pass the normals data
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) /* Normals start after
the position and uvs */);
        // Draws a non-indexed triangle array
        glDrawArrays(GL_TRIANGLES, 0, 6);
        // Unbind the VBO
        glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[2]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[1]);
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)));
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);


glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[5]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
if(pressed==3)
{
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
}
else
{
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[6]);
}
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)));
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[7]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
if(pressed==4)
{
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
```

```
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[3]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);


	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[8]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==5)
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[4]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);


	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[10]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==6)
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	}
```

```
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[7]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);



	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[13]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[13]);
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);



	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[15]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[5]);
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[16]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==9 )
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[2]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[18]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==10)
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[10]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);


	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[21]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
```

```
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==11)
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[9]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[23]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	if(pressed==12)
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[18] );
	}
	else
	{
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[12]);
	}
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	// This obj will be rendered only if the alarm is activated
	if(AlActivation==1)
	{
	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[3]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
```

```
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[20] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        }



}


/***********************************************
 @Function     Pad
 @Return       -
 @Description  This function is called when "call now"
               is selected in Main Menu. Handles all
               rendering in the call pad section of the
               interface.
***********************************************/
void OGLES2Texturing::Pad()
{


        int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
        glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
        glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
        glUniform3f( zLocation, 0, 0, 1);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[24]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[14] );
```

```
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[20]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[8] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[19]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[17] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[23]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[16] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
NumPad();   // Called to set the PrNum value according to user's click

if(PrNum==0)
{
        if(counter<20)            // to avoid text overflow
        strcat(NumToCall,"0");   // add character to text
        counter++;                // increment counter
}
else if(PrNum==1)
{
        if(counter<20)
        strcat(NumToCall,"1");
        counter++;
}
else if(PrNum==2)
{
        if(counter<20)
        strcat(NumToCall,"2");
        counter++;
}
else if(PrNum==3)
{
        if(counter<20)
        strcat(NumToCall,"3");
        counter++;
}
else if(PrNum==4)
{
        if(counter<20)
        strcat(NumToCall,"4");
        counter++;
}
else if(PrNum==5)
{
        if(counter<20)
        strcat(NumToCall,"5");
        counter++;
}
```

```
        else if(PrNum==6)
        {
                if(counter<20)
                strcat(NumToCall,"6");
                counter++;
        }
        else if(PrNum==7)
        {
                if(counter<20)
                strcat(NumToCall,"7");
                counter++;
        }
        else if(PrNum==8)
        {
                if(counter<20)
                strcat(NumToCall,"8");
                counter++;
        }
        else if(PrNum==9)
        {
                if(counter<20)
                strcat(NumToCall,"9");
                counter++;
        }
        else if(PrNum==10)
        {
                if(counter<20)
                strcat(NumToCall,"#");
                counter++;
        }
        else if(PrNum==11)
        {
                if(counter<20)
                strcat(NumToCall,"*");
                counter++;
        }
        // If no erasure command has been given, the number is flushed on screen
        if(erase==0)
        {
```

```
                m_Print3D.Print3D(10.0f, 45.0f, 1.0f, 0xFF302020, NumToCall);
                m_Print3D.Flush();
                Sleep(50);
                PrNum=100;  // Change PrNum value to avoid constant printing of the pressed number
        }
        // If erasure command has been given copy an empty string to the array
        else if (erase==1)
        {
                erase=0;                // Reset erase
                strcpy(NumToCall,empty);
                counter=0;              // Reset counter
        }

        // if call button is pressed and a number is dialed print message
        if(dial==1 && NumToCall[0]!=NULL)
        {
                m_Print3D.Print3D(10.0f, 50.0f, 0.8f, 0xFF605020, "your call is being diverted...");
                m_Print3D.Flush();
        }

        // If "back" arrow is pressed:
        if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.83 )
        {
                pressed=0;               // Change value to return to main menu
                strcpy(NumToCall,empty);  // Erase dialed number
                dial=0;                 // Erase printed message
                counter=0;              // Reset counter
        }
        else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.83 && y<0.91)
        {
                dial=1;
        }
        else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.91)
        {
                dial=0;
        }
        else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y<0.83 && y>0.66)
        {
                erase=1;
```

```
	}


}


/*****************************************************************
@Function		NumPad
@Return			-
@Description	This function is called in Pad Function to initialize
				PrNum depending on click position
*****************************************************************/
void OGLES2Texturing ::NumPad()
{
	if(0 != PVRShellGet(prefButtonState) && x>0.06 && x<0.275 && y<0.19 && y>0.07)
	{
		PrNum=0;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.275 && x<0.5 && y<0.19 && y>0.07)
	{
		PrNum=1;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y<0.19 && y>0.07)
	{
		PrNum=2;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.725 && x<0.94 && y<0.19 && y>0.07)
	{
		PrNum=3;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.06 && x<0.275 && y<0.36 && y>0.24)
	{
		PrNum=4;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.275 && x<0.5 && y<0.36 && y>0.24)
	{
		PrNum=5;
```

```
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y<0.36 && y>0.24)
	{
		PrNum=6;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.725 && x<0.94 && y<0.36 && y>0.24)
	{
		PrNum=7;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.06 && x<0.275 && y<0.53 && y>0.41)
	{
		PrNum=8;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.275 && x<0.5 && y<0.53 && y>0.41)
	{
		PrNum=9;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y<0.53 && y>0.41)
	{
		PrNum=10;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.725 && x<0.94 && y<0.53 && y>0.41)
	{
		PrNum=11;
	}

}


/****************************************************
@Function	ScreenSaver
@Return		-
@Description	This function is called when no mouse
		event occurs for sometime.
****************************************************/
void OGLES2Texturing::ScreenSaver()
{
	// Different ModelView and Projection Matrices
	// in order to achieve object rotation
```

```
        float aModelViewIT[] =
{
        cos(m_fAngle), 0,      sin(m_fAngle),
        0,                             1,      0,
        -sin(m_fAngle),        0,      cos(m_fAngle)
};


float aPMVMatrix[] =
{
        cos(m_fAngle), 0,      sin(m_fAngle), 0,
        0,                             1,      0,                              0,
        -sin(m_fAngle),        0,      cos(m_fAngle), 0,
        0,                             0,      0,                              1
};


int i32Location = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
glUniformMatrix4fv( i32Location, 1, GL_FALSE, aPMVMatrix);
i32Location = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
glUniformMatrix3fv( i32Location, 1, GL_FALSE, aModelViewIT);
i32Location = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
glUniform3f( i32Location, 0, 0, 1);

// Increments the angle of the view
m_fAngle += .02f;

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[25]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[15] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
}

/************************************************
@Function      AlarmClock
@Return        -
@Description   This function is called when the user
               nagivates into the alarm clock of the
               interface.
************************************************/
void OGLES2Texturing::AlarmClock()
{
	int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
	glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
	zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
	glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);
	zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
	glUniform3f( zLocation, 0, 0, 1);

	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[1]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[19] );
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[9]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[19] );
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[22] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[4]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[21] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[8]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[23] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
glEnableVertexAttribArray(NORMAL_ARRAY);
glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
glDrawArrays(GL_TRIANGLES, 0, 6);
glBindBuffer(GL_ARRAY_BUFFER, 0);

glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[20]);
glEnableVertexAttribArray(VERTEX_ARRAY);
glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
glBindTexture( GL_TEXTURE_2D, m_uiTexture[8] );
glEnableVertexAttribArray(TEXCOORD_ARRAY);
```

```
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);


	// Obj rendered only if the alarm is activated
	if(AlActivation==1)
	{
		glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[23]);
		glEnableVertexAttribArray(VERTEX_ARRAY);
		glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
		glBindTexture( GL_TEXTURE_2D, m_uiTexture[24] );
		glEnableVertexAttribArray(TEXCOORD_ARRAY);
		glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
		glEnableVertexAttribArray(NORMAL_ARRAY);
		glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
		glDrawArrays(GL_TRIANGLES, 0, 6);
		glBindBuffer(GL_ARRAY_BUFFER, 0);
	}

	if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.83 )
	{
		pressed=0;
	}
	else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y<0.16 )
	{
		AlActivation = 1;
	}
	else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.16 && y<0.32 )
	{
		AlActivation = 0;
	}
	else if(0 != PVRShellGet(prefButtonState) && x<0.25 && y>0.32 && y<0.48 )
	{
		Sleep(20);
		snooze+=1;   // Snooze value changes depending on mouse events
		if(snooze>4) // Reset snooze
```

```
		snooze=0;
	}

	// The following algorithm handles the hour incrementing
	// when the user presses the "+" button
	else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y<0.15)
	{
		if(lhour==0 || lhour==1)
		{
			rhour++;
			if(rhour>9)
			{
				rhour=0;
				lhour++;
			}
		}
		else if(lhour==2)
		{
			rhour++;
			if(rhour>3)
			{
				lhour=0;
				rhour=0;
			}
		}

	}
	// The following algorithm handles the hour diminishing
	// when the user presses the "-" button
	else if(0 != PVRShellGet(prefButtonState) && x>0.5 && x<0.725 && y>0.15 && y<0.30) // hour  -
	{

		if(lhour==0)
		{
			rhour--;
			if(rhour<0)
			{
				lhour=2;
				rhour=3;
```

```
                }
        }
        else if(lhour==1 || lhour==2)
        {
                rhour--;
                if(rhour<0)
                {
                        lhour--;
                        rhour=9;
                }
        }
}
// The following algorithm handles the minute incrementing
// when the user presses the "+" button
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y<0.15)  // minute  +
{
        rmin++;
        if(rmin>9)
        {
                rmin=0;
                lmin++;
                if(lmin>5)
                {
                        lmin=0;
                        rmin=0;
                }
        }
}
// The following algorithm handles the minute diminishing
// when the user presses the "-" button
else if(0 != PVRShellGet(prefButtonState) && x>0.75 && y>0.15 && y<0.30)  // minute -
{
        rmin--;
        if(rmin<0)
        {
                rmin=9;
                lmin--;
                if(lmin<0)
                {
```

```
                        lmin=5;
                        rmin=9;
                }
        }
}

// Turns the time intergers into strings
// so that we can project it on screen
// with CPVRTPrint3D class
itoa(rhour,buffe,10);
itoa(lhour,buffer,10);
itoa(rmin,buf,10);
itoa(lmin,buff,10);

m_Print3D.Print3D(99.0f, 30.0f, 1.5f, 0xFF202070, buf);
m_Print3D.Print3D(94.0f, 30.0f, 1.5f, 0xFF202070, buff);
m_Print3D.Print3D(91.0f, 30.0f, 1.5f, 0xFF202070, ":");
m_Print3D.Print3D(87.0f, 30.0f, 1.5f, 0xFF202070, buffe);
m_Print3D.Print3D(82.0f, 30.0f, 1.5f, 0xFF202070, buffer);
m_Print3D.Flush();

// Snooze cases
switch(snooze)
{
case 0:
        m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "snooze off");
        m_Print3D.Flush();
        break;
case 1:
        m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "1 min");
        m_Print3D.Flush();
        break;
case 2:
        m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "3 min");
        m_Print3D.Flush();
        break;
case 3:
        m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "5 min");
        m_Print3D.Flush();
```

```
                break;
        case 4:
                m_Print3D.Print3D(5.0f, 40.0f, 0.8f, 0xFF302020, "10 min");
                m_Print3D.Flush();
                break;

        }


}


/*****************************************************
@Function       Messages
@Return         -
@Description    This function is called when the user
                nagivates into the messages of the
                interface.
*****************************************************/
void OGLES2Texturing::Messages(){

        int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
        glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
        glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
        glUniform3f( zLocation, 0, 0, 1);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[8]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[8] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);
```

```
        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[0]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[25] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[4]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[28] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[2]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[27] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[6]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[29] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
```

```
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[10]);
	glEnableVertexAttribArray(VERTEX_ARRAY);
	glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
	glBindTexture( GL_TEXTURE_2D, m_uiTexture[26] );
	glEnableVertexAttribArray(TEXCOORD_ARRAY);
	glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
	glEnableVertexAttribArray(NORMAL_ARRAY);
	glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
	glDrawArrays(GL_TRIANGLES, 0, 6);
	glBindBuffer(GL_ARRAY_BUFFER, 0);

	if (send==1)
	{
		send=0;
		incr++; // increments the sent messages vakue
	}

	itoa(incr,bu,10);
	m_Print3D.Print3D(114.0f, 19.0f, 0.4f, 0xFF307030, bu);
	m_Print3D.Print3D(113.0f, 19.0f, 0.4f, 0xFF307030, "(     )");
	m_Print3D.Print3D(34.0f,  5.0f, 0.4f, 0xFF3010070, "Create Message");
	m_Print3D.Print3D(35.0f, 19.0f, 0.4f, 0xFF307030, "Incoming Messages");
	m_Print3D.Print3D(35.0f, 32.0f, 0.4f, 0xFF902070, "Back");
	m_Print3D.Print3D(95.0f,  5.0f, 0.4f, 0xFF307030, "email");
	m_Print3D.Print3D(95.0f, 19.0f, 0.4f, 0xFF902070, "Sent messages");
	m_Print3D.Print3D(95.0f, 32.0f, 0.4f, 0xFF307030, "Draft");
	m_Print3D.Flush();

	/*if (send==1)
	{
		send=0;
		incr++;
		itoa(incr,bu,10);
		m_Print3D.Print3D(95.0f, 22.0f, 0.4f, 0xFF307030, "bu");
```

```
                m_Print3D.Flush();
        }*/

        if(0 != PVRShellGet(prefButtonState) && x<0.45 && y>0.65 && LeftRight==0 )
        {
                Sleep(100);
                pressed=0;
        }
        else if(0 != PVRShellGet(prefButtonState) && x<0.45 && y<0.30 && LeftRight==0 )
        {
                mes=1;
        }
}

/****************************************************
@Function       WriteMes
@Return         -
@Description    This function is called when the user
                nagivates into the section of the
                interface intended for creating messages
****************************************************/
void OGLES2Texturing::WriteMes()
{

        int zLocation = glGetUniformLocation(m_uiProgramObject, "myPMVMatrix");
        glUniformMatrix4fv( zLocation, 1, GL_FALSE, aPMVMatrix);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myModelViewIT");
        glUniformMatrix3fv( zLocation, 1, GL_FALSE, aModelViewIT);
        zLocation = glGetUniformLocation(m_uiProgramObject, "myLightDirection");
        glUniform3f( zLocation, 0, 0, 1);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[24]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[30] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
```

```
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        glBindBuffer(GL_ARRAY_BUFFER, m_ui32Vbo[20]);
        glEnableVertexAttribArray(VERTEX_ARRAY);
        glVertexAttribPointer(VERTEX_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, 0);
        glBindTexture( GL_TEXTURE_2D, m_uiTexture[8] );
        glEnableVertexAttribArray(TEXCOORD_ARRAY);
        glVertexAttribPointer(TEXCOORD_ARRAY, 2, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (3 * sizeof(GLfloat)));
        glEnableVertexAttribArray(NORMAL_ARRAY);
        glVertexAttribPointer(NORMAL_ARRAY, 3, GL_FLOAT, GL_FALSE, m_ui32VertexStride, (void*) (5 * sizeof(GLfloat)) );
        glDrawArrays(GL_TRIANGLES, 0, 6);
        glBindBuffer(GL_ARRAY_BUFFER, 0);

        MesPad();

        switch(MesNum)
        {
        case 0:
               if(counter<46)                                          // to avoid text overflow
               strcat(text,"Q");                                      // add character to first line of text
               else if(counter>=46 && counter<92){strcat(text1,"Q");} // if 1st line is full add to the 2nd one
               else if(counter>=92 &&counter<138){strcat(text2,"Q");} // if 2nd line is full add to the 3rd one
               counter++;                                                     // increment counter
               break;
        case 1:
               if(counter<46)
               strcat(text,"W");
               else if(counter>=46 && counter<92){strcat(text1,"W");}
               else if(counter>=92 &&counter<138){strcat(text2,"W");}
               counter++;
               break;
        case 2:
               if(counter<46)
               strcat(text,"E");
               else if(counter>=46 && counter<92){strcat(text1,"E");}
               else if(counter>=92 &&counter<138){strcat(text2,"E");}
               counter++;
               break;
```

```
        case 3:
                if(counter<46)
                strcat(text,"R");
                else if(counter>=46 && counter<92){strcat(text1,"R");}
                else if(counter>=92 &&counter<138){strcat(text2,"R");}
                counter++;
                break;
        case 4:
                if(counter<46)
                strcat(text,"T");
                else if(counter>=46 && counter<92){strcat(text1,"T");}
                else if(counter>=92 &&counter<138){strcat(text2,"T");}
                counter++;
                break;
        case 5:
                if(counter<46)
                strcat(text,"Y");
                else if(counter>=46 && counter<92){strcat(text1,"Y");}
                else if(counter>=92 &&counter<138){strcat(text2,"Y");}
                counter++;
                break;
        case 6:
                if(counter<46)
                strcat(text,"U");
                else if(counter>=46 && counter<92){strcat(text1,"U");}
                else if(counter>=92 &&counter<138){strcat(text2,"U");}
                counter++;
                break;
        case 7:
                if(counter<46)
                strcat(text,"I");
                else if(counter>=46 && counter<92){strcat(text1,"I");}
                else if(counter>=92 &&counter<138){strcat(text2,"I");}
                counter++;
                break;
        case 8:
                if(counter<46)
                strcat(text,"O");
                else if(counter>=46 && counter<92){strcat(text1,"O");}
```

```
            else if(counter>=92 &&counter<138){strcat(text2,"O");}
            counter++;
            break;
        case 9:
            if(counter<46)
            strcat(text,"P");
            else if(counter>=46 && counter<92){strcat(text1,"P");}
            else if(counter>=92 &&counter<138){strcat(text2,"P");}
            counter++;
            break;
        case 10:
            if(counter<46)
            strcat(text,"A");
            else if(counter>=46 && counter<92){strcat(text1,"A");}
            else if(counter>=92 &&counter<138){strcat(text2,"A");}
            counter++;
            break;
        case 11:
            if(counter<46)
            strcat(text,"S");
            else if(counter>=46 && counter<92){strcat(text1,"S");}
            else if(counter>=92 &&counter<138){strcat(text2,"S");}
            counter++;
            break;
        case 12:
            if(counter<46)
            strcat(text,"D");
            else if(counter>=46 && counter<92){strcat(text1,"D");}
            else if(counter>=92 &&counter<138){strcat(text2,"D");}
            counter++;
            break;
        case 13:
            if(counter<46)
            strcat(text,"F");
            else if(counter>=46 && counter<92){strcat(text1,"F");}
            else if(counter>=92 &&counter<138){strcat(text2,"F");}
            counter++;
            break;
        case 14:
```

```
            if(counter<46)
            strcat(text,"G");
            else if(counter>=46 && counter<92){strcat(text1,"G");}
            else if(counter>=92 &&counter<138){strcat(text2,"G");}
            counter++;
            break;
        case 15:
            if(counter<46)
            strcat(text,"H");
            else if(counter>=46 && counter<92){strcat(text1,"H");}
            else if(counter>=92 &&counter<138){strcat(text2,"H");}
            counter++;
            break;
        case 16:
            if(counter<46)
            strcat(text,"J");
            else if(counter>=46 && counter<92){strcat(text1,"J");}
            else if(counter>=92 &&counter<138){strcat(text2,"J");}
            counter++;
            break;
        case 17:
            if(counter<46)
            strcat(text,"K");
            else if(counter>=46 && counter<92){strcat(text1,"K");}
            else if(counter>=92 &&counter<138){strcat(text2,"K");}
            counter++;
            break;
        case 18:
            if(counter<46)
            strcat(text,"L");
            else if(counter>=46 && counter<92){strcat(text1,"L");}
            else if(counter>=92 &&counter<138){strcat(text2,"L");}
            counter++;
            break;
        case 19:
            if(counter<46)
            strcat(text,"Z");
            else if(counter>=46 && counter<92){strcat(text1,"Z");}
            else if(counter>=92 &&counter<138){strcat(text2,"Z");}
```

```
                counter++;
                break;
        case 20:
                if(counter<46)
                strcat(text,"X");
                else if(counter>=46 && counter<92){strcat(text1,"X");}
                else if(counter>=92 &&counter<138){strcat(text2,"X");}
                counter++;
                break;
        case 21:
                if(counter<46)
                strcat(text,"C");
                else if(counter>=46 && counter<92){strcat(text1,"C");}
                else if(counter>=92 &&counter<138){strcat(text2,"C");}
                counter++;
                break;
        case 22:
                if(counter<46)
                strcat(text,"V");
                else if(counter>=46 && counter<92){strcat(text1,"V");}
                else if(counter>=92 &&counter<138){strcat(text2,"V");}
                counter++;
                break;
        case 23:
                if(counter<46)
                strcat(text,"B");
                else if(counter>=46 && counter<92){strcat(text1,"B");}
                else if(counter>=92 &&counter<138){strcat(text2,"B");}
                counter++;
                break;
        case 24:
                if(counter<46)
                strcat(text,"N");
                else if(counter>=46 && counter<92){strcat(text1,"N");}
                else if(counter>=92 &&counter<138){strcat(text2,"N");}
                counter++;
                break;
        case 25:
                if(counter<46)
```

```
                strcat(text,"M");
                else if(counter>=46 && counter<92){strcat(text1,"M");}
                else if(counter>=92 &&counter<138){strcat(text2,"M");}
                counter++;
                break;
        case 26:
                if(counter<46)
                strcat(text," ");
                else if(counter>=46 && counter<92){strcat(text1," ");}
                else if(counter>=92 &&counter<138){strcat(text2," ");}
                counter++;
                break;
        case 27:
                if(counter<46)
                strcat(text,"@");
                else if(counter>=46 && counter<92){strcat(text1,"@");}
                else if(counter>=92 &&counter<138){strcat(text2,"@");}
                counter++;
                break;
        case 28:
                if(counter<46)
                strcat(text,".");
                else if(counter>=46 && counter<92){strcat(text1,".");}
                else if(counter>=92 &&counter<138){strcat(text2,".");}
                counter++;
                break;
        }

        // Flush text to screen
        if(erase==0)
        {
                m_Print3D.Print3D(7.0f, 45.0f, 0.6f, 0xFF302020, text);
                m_Print3D.Print3D(7.0f, 50.0f, 0.6f, 0xFF302020, text1);
                m_Print3D.Print3D(7.0f, 55.0f, 0.6f, 0xFF302020, text2);
                m_Print3D.Flush();
                Sleep(20);
                MesNum=100; // Change to random value to aoid constant printing
        }
```

```
	// Erase the last element of the text
	else if(erase==1)
	{
		erase=0;
		if(counter<46)
		{
			text[counter]=NULL;
		}
		else if(counter>=46 && counter<92)
		{
			text1[counter-46]=NULL;  // Subtract elements of the 1st line array
		}
		else if(counter>=92 && counter<138)
		{
			text2[counter-92]=NULL; // Subtract elements of the 1st line and second line array
		}
		counter--;                 // Subtract one from counter
		Sleep(30);
	}

	if( (0 != PVRShellGet(prefButtonState) && x<0.20 && y>0.80)  ||  send==1  )
	{
		Sleep(100);
		pressed==8;           // Takes it back to main message menu
		mes=0;
		strcpy(text,empty);  // Empties written text on exit
		strcpy(text1,empty);
		strcpy(text2,empty);
	}

}


/*****************************************************************
@Function      MesPad
@Return        -
@Description   This function is called in MesPad Function to initialize
               MesNum depending on click position
*****************************************************************/
```

```
void OGLES2Texturing ::MesPad()
{
	if(0 != PVRShellGet(prefButtonState) && x>0.05 && x<0.13 && y<0.17 && y>0.12)
	{
		MesNum=0;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.14 && x<0.22 && y<0.17 && y>0.12)
	{
		MesNum=1;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.23 && x<0.31 && y<0.17 && y>0.12)
	{
		MesNum=2;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.32 && x<0.40 && y<0.17 && y>0.12)
	{
		MesNum=3;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.41 && x<0.49 && y<0.17 && y>0.12)
	{
		MesNum=4;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.50 && x<0.58 && y<0.17 && y>0.12)
	{
		MesNum=5;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.59 && x<0.67 && y<0.17 && y>0.12)
	{
		MesNum=6;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.68 && x<0.76 && y<0.17 && y>0.12)
	{
		MesNum=7;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.77 && x<0.85 && y<0.17 && y>0.12)
	{
		MesNum=8;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.86 && x<0.92 && y<0.17 && y>0.12)
```

```
	{
		MesNum=9;
	}

	else if(0 != PVRShellGet(prefButtonState) && x>0.09 && x<0.17 && y>0.18 && y<0.23)
	{
		MesNum=10;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.18 && x<0.26 && y>0.18 && y<0.23)
	{
		MesNum=11;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.27 && x<0.35 && y>0.18 && y<0.23)
	{
		MesNum=12;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.36 && x<0.44 && y>0.18 && y<0.23)
	{
		MesNum=13;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.45 && x<0.53 && y>0.18 && y<0.23)
	{
		MesNum=14;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.54 && x<0.62 && y>0.18 && y<0.23)
	{
		MesNum=15;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.63 && x<0.71 && y>0.18 && y<0.23)
	{
		MesNum=16;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.72 && x<0.80 && y>0.18 && y<0.23)
	{
		MesNum=17;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.81 && x<0.89 && y>0.18 && y<0.23)
	{
		MesNum=18;
```

```
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.18 && x<0.26 && y>0.24 && y<0.29)
	{
		MesNum=19;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.27 && x<0.35 && y>0.24 && y<0.29)
	{
		MesNum=20;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.36 && x<0.44 && y>0.24 && y<0.29)
	{
		MesNum=21;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.45 && x<0.53 && y>0.24 && y<0.29)
	{
		MesNum=22;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.54 && x<0.62 && y>0.24 && y<0.29)
	{
		MesNum=23;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.63 && x<0.71 && y>0.24 && y<0.29)
	{
		MesNum=24;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.72 && x<0.80 && y>0.24 && y<0.29)
	{
		MesNum=25;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.23 && x<0.53 && y>0.30 && y<0.35)
	{
		MesNum=26;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.54 && x<0.65 && y>0.30 && y<0.35)
	{
		MesNum=27;
	}
	else if(0 != PVRShellGet(prefButtonState) && x>0.66 && x<0.77 && y>0.30 && y<0.35)
	{
```

```
                MesNum=28;
        }
        else if(0 != PVRShellGet(prefButtonState) && x>0.83 && x<0.92 && y>0.24 && y<0.29)
        {
                erase=1;
        }
        else if(0 != PVRShellGet(prefButtonState) && x>0.75 && x<0.92 && y>0.30 && y<0.35)
        {
                send=1;
        }
}
```