Programming with ALGE Vanilla SDK

Developing Platform Independent
3D Graphics Applications

CHAPTER 1
Quick Start Guide

In this chapter you will plunge in Vanilla World and develop your first ALGE Application from scratch and deploy it on multiple devices using the ALGE SDK.

The ALGE Kit includes the following:

Platform Templates

Your SDK includes Platform Templates or IDE project folders. These templates are meant to develop and port the Apps to the particular platform. Essentialy you may debug and develop your Apps (even iOS Apps) in MS Visual Studio. Or conversely you can develop your Windows Apps in XCode. You will get the hang of this idea progressively.

Essential Includes

Every ALGE SDK App is required to include the core header file core CBase_<version>.h, this directory includes the requisite headers.

Tools and Utilities

ALGE is not a full fledge game development engine however it provides some basic tools to import and export assets. E.g Blender ALX export script is provided to export 3D Assets to ALX format , the ALX Viewer to view 3D assets. An extension config tool is provided to package an extension.

Sample Apps

Few Sample Apps are provided for demonstration and for your HelloWorldly desires. Any one of these Apps can be used as a baseline of your App Idea thus avoiding writing boilerplate code.

Extensions

The extensions as the name signifies extend the functionality available for you. e.g Physics Library like Bullet3D is available as an extension. A game engine like Irrichlit can also form an extension. Refer to Adrenaline zone on Alge website to share and

find new extensions.

Documentation

The standard documentation shipped with SDK in html, pdf or docx provided for reference and kickstarts. Few Sample Videos of Alge-Aldernaline channel are also included in the DVD distribution.

This book deals with introduction and kickstarts and would introduce the ALGE Paradigm of programming. So expect Shedding off some irrelevant styles and get ready to embrace some tried and tested techniques.

Transitioning to Vanilla Coding

Platform OEM's always like you to stick to their particular toolset and associated technologies. Apple wants people to use Objective-C and buy XCode and Microsoft obsessively directs and wants people to use .NET and buy MS Technologies. If u develop a App for Android and then later when it would decide to port it to Symbian or iPhone, what's your best course of action? Invest in Porting!! And after investing you would also like to retain your platform specific workforce. What does this have to do with vanilla? Take a few steps back and See. You are in the Game of making your product cut through the market and shine before anyone one else does this faster than you, Period !! ALGE SDK enables you to do that. You Think, design, code, debug in a platform independent way and it lets you quickly port your Apps using automation.

Of course you would require ***'the platform'*** for deployment and pre-release testing of your Application. By Vanilla what I mean is to use the platform and tools of your choice for development code the App in pure C / C++ as inherently it's a portable language across the OpenGL world.

Visual Studio is the ALGE's preferred development environment due to its debugging facilities. This does not mean that you won't be able to program for ALGE if you own a Mac, XCode now also have elegant facilities and ALGE SDK has no issues with that too.

Why C/C++ ?

C++ is a widely used *lingua facia* for game developers all around the world. It is inherently portable and is designed for portability across platforms. You can achieve unmatched performance in comparison with other languages. You can develop mixed mode structured and object oriented both. OpenGL ES is a C API so you program closer to the metal. For 3D graphics certain C++ features are indispensable. Operator overloading makes it possible to perform vector math in a syntactically natural way. Templates allow the reuse of vector and matrix types using a variety of underlying numerical representations. ALGE SDK includes all Glue code necessary to interface your C/C++ vanilla code to the platform specific code, on iOS ALGE uses Objective-C and for Android it uses Java Native Interface to talk to the platform. But as a developer your purview remains only Vanilla C/C++.

Getting Started

Assuming you are a developer who has good handle on C/C++ and are perplexed from where to dwell further in App making and monetization. You would like to present your idea to the world as quickly as possible to the app store users.

If u own a Windows Machine you can develop using ALGE and sell your app on Windows Marketplace, Intel AppUp, Android Marketplace right out of the box. If u own a Mac you can test and port your App on iTunes Store, Mac App Store and Android Marketplace. If U own both you can develop on one machine and port to many possible outlets, Windows Marketplace, Tunes App Store, Mac App Store, Android Marketplace and the Intel AppUp Store. You may also distribute the Android Apps to third party stores around the globe.

When u have a ALGE compliant release candidate ready you can use ALGE templates to port it right away. By the way you won't have to rewrite your app as when moving from one machine to another as your App is ALGE compliant i.e Vanilla. So you have maximum reach and a fairly broad user base unmatched by any other SDK offerings. I suspect you are now feeling the Adrenaline rush as part of ALGE process, and Wait the moment you would be finished by the prototype you would be enthralled by the possible prompt porting schemes ready to comply to your instructions.

A common repository or version control system on your networked machines is recommended to transfer App codebase. If u prefer to use cloud 'Windows Live Mesh' is a very good way to maintain a common repository for a beginner.

Make your plan to get hold of test devices for every platform you intend to use. You would probably already own a device that supports OpenGL ES. I started from buying a cheap iPod Touch 2G and an Android G1 and released apps for iPhone 4, Galaxy Tab, and even iPad. The Apps paid off well for upgrades.

Jumping into Apple territory means you need a Mac (could be cheap Mac Mini) but that's worth investing for a starter. Few SDK paradigms claim to write complete iOS App on Windows PC without even requiring the platform device. Come On!! Getting hold of a low spec device on which you users will actually use and feel your app is not at all a bad bet considering the usability and lovability of these devices. After all you are required to invest essentially in App Developer programs and beta testers, why not invest in a device of the OEM in whose ecosystem you are going to benefit from.

As Android tools for both OSX and Windows are available, you can even start developing on a Windows Workstation and release Apps to Android community. Monetizing Android Apps require you to incorporate Ad based services and the Good News is that ALGE SDK has pre-configured Admob (In App Ads) and Airpush (Out of App Ads) SDK so your first Hello World App also shows up the Ads out of the box.

Installing the ALGE SDK

The installation is pretty straightforward. Extract the package to your repository and you are good to Go. First step would be to use and Run existing provided Applications to get the idea of the Caps. Gather your devices, iPod touch, iPhone , Android Phone, iPad, Atom Notebook etc to automatically build an existing App from source. ALGE SDK Folder can be copied to another machine and would run as all paths are relative. It also simplifies to zip your App Folder to backup all your work periodically.

Hello ALGE World

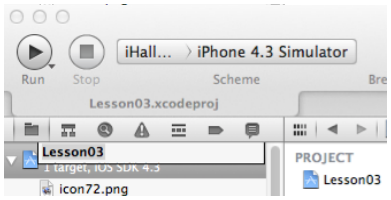
Under the platforms folder you can find IDE project templates. To take a start Let's compile and a defacto demo application **'Lesson03'**. In fact this is an Alge compatible port of Lesson03 of the defacto OpenGL tutorials of NeHe (original is found at http://nehe.gamedev.net). This App displays a colored rectangle and a colored Quad on the Screen. You would be glad that Lesson 01 and Lesson 02 are not required as initialization of GL Context and window creating has already been catered for by ALGE framework.

Running the first Demo App – XCode (iOS)

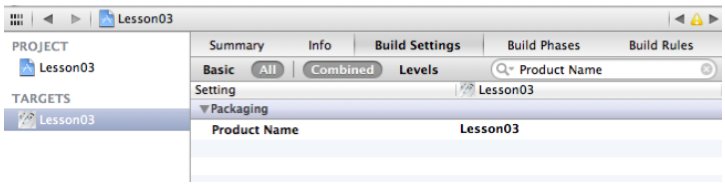
Check that you have properly configured iOS SDK and XCode IDE installed on your Mac. You can counter check it by creating and running standard OpenGL ES Template provided in XCode.

Coming to ALGE, In XCode open up the Project file (.xcodeproj) in provided in the Platforms/Alge-iPhone directory.

Step#1 – Name your Project; in this case 'Lesson03'.



Step#2 – Name your Product; in this case 'Lesson03'.



Step#3 – Specify Resource Files; in this case SKIP THIS STEP as no resource files are required.

Step#4 – Set Candidate

change
#define CANDIDATE "../../../Apps/APPNAME/App.hpp"
to
#define CANDIDATE "../../../Apps/Lesson03/App.hpp"

Step#5 – Set Active Device Scheme and Run

You should see the XCode firing up the iOS Simulator and your First Demo App is Running.

Lets Generalize the Configuration Steps to Develop/Run/Port any App using XCode in four easy to remember steps.

Step A – Name your Project/ Product;
Step B – Add Resource files to Project; (described later)
Step C – Set Candidate;
Step D – Select Scheme and Run.

Running the first Demo App – Visual Studio

In Visual Studio open up the Solution Alge.sln. The default project contains a file name CANDIDATE.h that defines the App Directory of the Candidate Application. Edit CANDIDATE.h to point to Lesson03 App.hpp file as under. The Candidate.h header file serves two functions. Firstly it is used to include the App source at build time. And secondly at runtime it is used by SDK to extract the name of the application.

change
#define CANDIDATE "../../../Apps/APPNAME/App.hpp"
to
#define CANDIDATE "../../../Apps/Lesson03/App.hpp"

The build tools of Alge SDK determine that you are interested in building Application class under Lesson03 folder that resides in Apps directory of SDK.

Hit Build> Clean, Rebuild All and Run. You will see the Demo App running in a 320x480 Win32 container Window.

Note:-

To specify a longer name of the Application eg 'My First App' is defined by substituting underscores instead of spaces.

```
#define CANDIDATE "../../Apps/My_First_App/App.hpp"
```

App Structure

The Application is a ***single file hpp*** containing class definitions and implementation. The main class name is **App**

e.g Contents of File `AppName/App.hpp`

```
//DEFINES

class App {
        public:
        //Essential public ALGE Variables Objects and Methods
        //USER Variables, Objects and Methods
};
```

The App is identified not by the name of class nor by its File name, it is identified by the Parent Folder in which the App.hpp resides. Another important convention to remember is that the App's data files should reside in Data subfolder. E.g all icons, and resource files (or assets) would be placed in Data folder within the App folder. You may cross examine other sample apps (app==folder) to get a hold on this scheme. Here are few basic standard conventions:-

- Every App Folder resides in Apps directory of SDK
- Every App has a self contained **Data** folder
- Every App is a encapsulated in a class named **App** contained in file App.hpp
- App Folder name '**IS THE'** App's Name

Getting used to these conventions is a good idea at the beginning to avoid confusions later.

Essential ALGE Variables Objects and Methods

The App class should contains few essential elements that are to be used as-is.

a. Two **PEG** Objects input and output, for communicating with platform.
b. Init Method Signature
c. DeInit Method Signature
d. Render Method Signature

These are the essential ingredients and may not be altered or modified else builds will fail.

Every App class is expected to contain public instances of class PEG named **input** and **output**. Your Application can read input messages queued for processing through the '**input PEG**' and send or request platform services through **'output PEG'**. PEG is not an acronym, it can be considered as a peg that joins items or a channel of communication of messages and associated data.

IMPORTANT:- Your App has ***one chance per message***, if the queue is read and message is ignored, the message is removed the queue assuming you don't intend to act on it.

The **Init Method** is called once the Platform is ready with OpenGL context and loading other preliminaries. The path in which the app can read and write data in a vanilla way is passed to you as a parameter. Simply put you can even use core C stdio functions on every platform using this path. This is also used to initialize the resource manger(s) in your application to read and write to assets.

The **DeInit** method is called before the App quits to give you an opportunity to deallocate any user objects and perform de-initialization of the App.

The **Render** method is called in a loop. This is the place where you have full control available to render in the OpenGL context, Normally you would be performing these blocks of operation.

(a) **processInput()** You must Process the input queue to act upon messages.

a. Issue OpenGL Calls.

b. Anything other thing you would like to do, Update Animation and Physics Library etc.

Every render method is accompanied with a float elapsed time variable (in seconds) to be used for time based animation and also give you scaled raw accelerometer values for computing device orientation. .The elapsed time is provided in seconds, you can easily compute millis (if required) by multiplying this value by 1000. The Seconds offer a more natural way to visualize your Animation code. Luckily, You are not required to setup anything in this regard, if the platform is supporting these features you will get these values by default.

USER Variables, Objects and Methods

All your Application specific variables and objects can be defined and used within the class in the same hpp file. It is recommended that you restrict your App's code in the same hpp file, you may use your own objects as a separate hpp code file and include in the App.hpp defines section, but still a better way is to write re-usable code is through the Extension Scheme of the SDK. You can use inner classes and avoid using class wizards that auto generate class file pairs of Cpp and h files one per class. If you are used to using C/C++ cpp and header files separately, consider making all reusable code as an Extension, and retain only the App specific code in the App.hpp File.

About Messaging and Commands

The Messaging scheme of ALGE is first come first serve, impermanent messages and a parameter pair Param1 and Param2. You would find that though only two variables are used per message but they are sufficient to pass everything and they can also pass pointers to structures and objects. For most of the Commands only one Parameter is good enough.

IMPORTANT:- Ensure that passing pointer addresses are pointing to valid memory location that should not go out of scope until the message is processed.

Every Message is expected to be sent as

MessageID,Param1,Param2

Message ID's are Constant integer defines in Commands.h Header files.

E.g To send a message to play a sound these two messages are used CMD_SNDSET and CMD_SNDPLAY.

These are called as under

```
output.pushP(CMD_SNDSET ,$ "MySong.mp3", 0);
output.pushP(CMD_SNDPLAY,$ "MySong.mp3", 0);
```

CMD_SNDSET is generally required to be called once in your init method.

And CMD_SNDPLAY message is sent whenever you would like to play the sound file e.g on a particular event.

e.g consider the following code to play a mp3 file over a blank screen:-

```
1.    #include "../../Base/CBaseXX.h"
2.    class App {
3.          public:
4.                PEG input, output;
5.          void Init(char *path) {
6.                output.pushP(CMD_SNDSET0,$ "MySong.mp3",0);
7.                output.pushP(CMD_SNDPLAY0,NOPARAMS);
```

```
8.          }
9.          void DeInit() {}
10.         void Render(float sec, int aX, int aY, int aZ)() {}
11.   };
```

That's Everything you need to do to play your wave file irrespective of Platform!!. The same code will be valid for iPhone, OSX, Windows, Android and others.

Lets revisit some characteristics of the this 11 line APPLICATION.

1. It uses Core Command defines and utility functions.
2. Define everything your app does is within your class App.
3. It only includes the essentials, except Line 6,7 concerning sound, each and every line is **'Essential'.**
4. You would be encountering the remaining 9 lines in each and every ALGE App.
5. In essence you are just concerned about specifying a sound file and playing in a `***Vanilla Way'***

Give it a 'Push' !!

You can push and pull messages from the PEG's, pushP is used for parameters of type POINTER; After all the file name is a const char*, Also $ is a helper define that means ((void*)(char*)) cast.

The PEG Objects also include another variant to support integer values, thru call pushI , if u need to pass Integer parameters.

The commands are equivalent:-

```
output.pushI(CMD_COMMAND, 12, 13);        // pass as integer
```

or

```
int p1 =  12;
int p2 =  13;

output.pushP(CMD_COMMAND, &p1, &p2);      // pass pointers to integers
```

you can use pull commands to read inputs to you app

```
PEG::CMD* cmd = input.pull();
int width, height;
If (cmd->command==CMD_SCREENSIZE) {
      width = cmd->i1; height = cmd->i2;
}
```

It is highly recommended to separate the input processing in a separate function as seen in examples processInput() that is executed once or less in every render call e.g.

```
class App {

public:
	PEG input, output

void processInput() {
	PEG::CMD* cmd = input.pull();
	int width, height;
		If (cmd->command==CMD_TOUCHMOVE) {
			x = cmd->i1; y = cmd->i2;
		}
		If (cmd->command==CMD_SCREENSIZE) {
			width = cmd->i1; height = cmd->i2;
		}
	}

	void DeInit() {}
	void Render(…) {
		processInput();
		// Other Rendering calls
	}

};//~App
```

Let's Return to 3D Graphics

Now its time to break the Good News!!

a. ALGE is mainly intended for loop structured 3D graphics applications.

b. Init Method is called after platform specific OpenGL context and window creation. The Init is for the init gl calls concerning your App. So you can start issuing GL calls right away.

c. Render method provides you maximum opportunity to draw your stuff and run app logic, the moment ALGE code gets execution handle it processes your output queue acts on all pending commands and again re-enters your Render function with a time elapsed variable.

d. The FPS is not restricted to a fixed value, rather you get elapsed time variable to perform frame rate independent animations. Essentially this elapsed time is equivalent to

processing time of all pending ALGE output queue messages.

e. DeInit provides you the last opportunity to de-init your objects and allocation before exit.

Why Apps Require Clean all before builds

As the main source hpp file is external to project structure the IDE knows, it cannot properly determine if the file was touched by editor. After editing the App.hpp file the main cpp file remains untouched therefore the complied skips the includes. This problem can be resolved by either touching the main file to mark it dirty after every editing session or by adding the application's hpp file in the project structure. One method is to keep the alternate editor on and manually touch the main file to mark it dirty. This would avoid cleaning all and only main file along with all includes will we recompiled and could save time particularly when extensions with numerous files are also in use. Clean all also caters for the issue but would necessitate recompiling all files thus induce additional time in build cycle. Newer IDE versions wont pose you this issue at all.

Extensions

The extensions provide a way to incorporate additional functionality to the SDK. The extensions are also by nature vanilla in way that they only contain classes or closely related functions for ease of use or to hide complexity of the code. The user includes and source is packaged in a folder without any outside dependencies however an extension parent folder can include other files relative to the root folder. The extension folder may also contain helper classes for ease of use. The purpose of extension scheme is to provide a mechanism to reuse existing vanilla sources. These packages could easily be shared or reused for different applications. A common repository of extensions assure that the code is kept in latest state across all applications. keeping the extensions folder in source control is a good way of sharing the codebase. The SDK includes a tool mkdep that allows for packing your extensions. The tool simply lists all the files in the folder that needs to be compiled.

Some Words on Windows Phone 7

Firstly Windows Phone 7 has no support for OpenGL ES. Well that’s an Aargh !! for everyone comfortable for programming in OpenGL. And you have to Code in C# using XNA, native support is also excluded officially. That’s a tough bet for all cross-platform tool developers.

Any way until Microsoft Releases some good news to enable them, we are holding short. ALGE code is C/C++ compliant and C# is not C/C++, however our hopeful premise is that automatically converting Vanilla C++ and Vanilla C# code both ways ***is not impossible***. So if not full, some partial support can be provided with help of automation. If Microsoft enables p/Invoke Native support (as in Windows CE devices) then certainly our job would accelerate.

Nevertheless we have not abandoned the idea and plan to include an experimental Alge-Phone7 Template in ALGE Releases above 1.0. You may be able to use XNA style calls in ALGE and ALGE style calls right in C# to obtain platform services. Also, MonoXNA is a project that gives us hope to bring XNA style to even OSX and iOS.

Nevertheless, we love MS Tools, and ALGE team has absolutely no intentions to exclude support for remaining Windows platforms like Windows 8, Win32 (98, ME, 7, XP, NT, 2K) and Windows CE (All these platforms support OpenGL and Native libs).

Happy Coding !!