# Nephilim SDK

This game engine is written and maintained by Artur Moreira (artturmoreira@gmail.com).

# Nephilim SDK

This software development kit can be considered a generic purpose game engine. It is aimed at programmers of all levels of expertise, as beginners can achieve great things easily as well as advanced users can leverage more complex features to making top quality games. You will be using C++ to program games with this library, even though scripting is possible natively with AngelScript.

One of the biggest aims of this tool is to provide portability. This means in essence a seamless way to program games for multiple platforms. The same source code will run just the same in platforms such as Windows, Linux, Mac, Android and IOS.

It makes possible to render hardware accelerated graphics in all these platforms, both 2D and 3D, using OpenGL and its mobile variants. Exactly to make the library versatile for a wide range of users, virtually no limits exist in the graphics system. You can use low level systems close to raw OpenGL as well as high level drawing facilities which do everything for you.

Also, all kinds of API's are provided along with the graphics. You will find ways to play sounds, to manage geometry, images, animations and other game specific tasks.

## SFML

This library was initially made using SFML to its fullest but once the engine started to be ported to android and IOS, SFML wouldn't work well anymore as it doesn't support these platforms. Because of this, the use of SFML was slightly dropped as more progress was made in the library. At this point still some SFML's facilities are used in Nephilim, in some platforms, but the rest was re-implemented. Nevertheless, the library is heavily inspired on SFML and takes many ideas from it. This actually makes porting games made with SFML to Nephilim quite fast and easy, as the API's are very similar. Also, while some features were re-implemented, one can still find many fragments of SFML's source within the source code of Nephilim. For all these reasons together, SFML will always be a big part of this engine's development and features, even if it is not being used directly by it. So, this foreword works as a way of saying thanks to Laurent Gomila, the original SFML author, whose code helped this library come true greatly. I do not claim authorship over any source that may have been transposed from the SFML's source into Nephilim and this is a clear statement of Laurent's official authorship of any of those fragments.

## Required tools

In order to take advantage of the SDK to its fullest, there are a few things you should have in your system before starting.

### Premake

The premake build system is primarily used by the sdk for building itself and the samples. The build script targets the version *4.4beta4*, so it is important that you use that version or higher of premake in your system. This tool is amazing and will certainly make our life easier.

### Git

The git repository management system is a great tool. GitHub hosts the sdk latest sources and allows many powerful features like pull requests and forking possible. While you don't absolutely need git in your machine to use the sdk, it will help a great deal when working with it.

## Installing on Linux with premake4

This is a quick getting started tutorial for linux users. Here's a simple way to get Nephilim up and running.

1. Open a terminal and go to the desired directory
2. git clone https://github.com/DevilWithin/Nephilim.git
3. cd Nephilim
4. premake4 gmake
5. cd build/gmake/
6. make OR make config=release

And done! The library is built in /Nephilim/lib and the samples are built in /Nephilim/bin.

Note: If some sample is failing to compile you can just "make SampleName" to build a specific sample directly.

## Installing on Windows with Visual Studio

Once you have the repository in your machine, you can:

1. premake4 vs2010
2. Open the project file in /build/vs2010/ and start building

## Preparing an Android environment on Linux

Not optimized yet.

## Preparing an Android environment on Windows

Not optimized yet.

## Building in the Apple IOS system

Not optimized yet.

## Engine

The engine class works as a program flow manager. It is responsible for initialization of an environment capable to run multiple games at the same time. Then, it takes care of updating and rendering those games, as well as delivering them events. In the end, it will destroy all system resources used by itself and its games.

A typical usage would be as such:

```
Engine myEngine;

myEngine.init();              // Initialization with default settings
myEngine.execute(new Game()); // Pushing one game to the environment

while(myEngine.isRunning())
          myEngine.update();   // While a close order is not issued, update the game

myEngine.shutdown();          // Shutdown everything~
```

Usually, this setup is enough and you can then program your game, using the GameCore class.

## GameCore

In order to run a game instance in the engine, you must declare a new class for your game which inherits GameCore. Only child classes of GameCore are accepted by the engine as executable games. After you have your own game class, it is immediately ready to be run by the engine by calling Engine::execute().

Here's how it could look, in its simpler form:

```
class CoolGame  : public GameCore{ };
```

All your game code must then be implemented inside this custom class, from event handling to rendering. This ensures a proper encapsulation of all your code so the engine can manage any game in a generic way, allowing multiple games to be run at once.

Since the engine commands the gaming environment, it will give orders to the running games through callbacks. Whenever there is an event, it will be delivered to the GameCore child through a virtual function. Whenever a new frame needs to be rendered, it will tell the game to draw its graphics. The engine manages automatically a fixed time step, which can be dynamically configured for each game. This brings many benefits and ensures proper timing within your application. Whenever it's time to perform an update, the engine will also notify your game to do so, along with how much time needs to be updated.

Here are the most common functions you will need to implement, in order to be notified by the engine's environment:

```
void onCreate();
void onRender();
void onEvent(Event &event);
void onUpdate(Time time);
```

They are pretty simple to use. onCreate() is called once when the game initializes. onRender() is called whenever a new frame is needed. onEvent() notifies you that one particular event occurred. This one is very similar to SFML's events, with some additions for touch devices.

Finally, onUpdate() will be called every time the game needs to be stepped forward by an amount of time.

## Main function

Nephilim doesn't force you to use a particular method to implement the main function, as long as you use the Engine class right, it will work fine. However, a fully portable "main" function is hard to achieve. For example, in Windows, you will need at least two types of main functions, one for when you have a console active and another using the WinAPI for showing a window only. In Android and iOS you will find trouble here as the device OS manages the program flow by itself, in Java and Objective C, respectively. Therefore, for these platforms, proper callbacks need to be made in order to ensure the Engine works properly, without using a main() function at all. This adds a complexity you probably don't want to deal with. Luckily, you won't ever have to, if you decide to use the "built-in" main function.

## Generic Main

In the lack of a better solution, Nephilim ships a header file which handles all platforms automatically, allowing a fairly easy approach to a generic main function. In case this header doesn't provide something you need, it can be implemented after a discussion on the topic. Otherwise you will need to do things yourself manually.

Here's how a generic main can look like:

```
#define ANDROID_PACKAGE_NAME com_nephilim_game
#define ANDROID_ACTIVITY_NAME MyGame
#include <Nephilim/GenericMain.h>
#include "MyGameHere.h"

MyGame game;

void init()
{
	_engine->init();
	_engine->execute(&game);
}

void update()
{
	_engine->update();
}

void shutdown()
{
	_engine->shutdown();
}
```

The first two macros are essential if you are developing for android. They will allow you to specify what package name you chose and the name of the Java class for your activity. They must be specified in the exact same format, with case sensitivity. These are used internally to communicate from C++ to Java via JNI.

Then, we include the GenericMain.h header and the one we declared our game in. We allocate memory for our game and then send it to execution by the engine, after initializing it. The three functions init(), update() and shutdown() are hardcoded in GenericMain.h and must be implemented forcefully when using this approach. They will control the execution of your program automatically. This should come pretty intuitively, but nevertheless you probably won't need to touch this code often, or at all. The engine pointer _engine is also hardcoded in

GenericMain.h. It is the engine being used internally to work with all platforms and to run your games.

# Graphics

## Surface

A surface is a class that represents the drawing canvas that you are rendering graphics to. It is not necessarily a window. You can get the current surface used by the engine at all times inside your game class by calling getSurface(). Examples of surfaces are the iPhone screen, the Android device screen, a browser embedded html element, a normal graphical window in Linux, Windows and OS X or even a custom widget inside a known framework such as Qt.

This class then provides you some information that is very helpful when developing a game. You can check the size of the surface; you can check what kind of surface it is and other things.

## Renderer

Tied to every valid surface is always a Renderer. This is a fairly flexible class to render portable graphics. It will draw using OpenGL in desktop systems and OpenGL ES 1.0/2.0 in embedded devices. You do not need to know about OpenGL directly to use it. It will provide both low-level facilities to allow you to exploit graphical performance, as well as high-level drawing functions, which make drawing one entity as simple as calling one function.

You can get the current available Renderer at all times from inside your game class by calling getRenderer().

## Renderer States

Since the Renderer class uses OpenGL for hardware-accelerated graphics, it naturally works as a state-based object, just as OpenGL is a state-based API. This means the Renderer class holds much information about the current state of the graphics system. It will both keep track of OpenGL states as well as set/get them. For this reason, it is probably a bad idea to change the graphics state without using the Renderer interface, like binding a texture manually instead of calling the appropriate function in the renderer.

Still on the states topic, the renderer holds information about many different states and has a clear definition of what the “defaults” are. This helps you return to the original state of the graphics system if you suspect something was damaged somewhere. All defaults are explained next.

### *Render Target*

A render target is a OpenGL framebuffer, in other words, a output of the graphics drawing process. This is usually the surface but can also be an off-screen texture.

Calling Renderer::setDefaultTarget() will always redirect graphical output to the Surface. Otherwise you may call directly Renderer::setTarget() to redirect output to a RenderTexture or other RenderTarget derived object.

## Shaders

A shader is basically a program that is activated in the graphics card and that defines how the rendering happens. You can later define your own custom shaders, but until you are not comfortable with them you don't need to know anything about them.

The renderer class has a default shader to provide the default functionality, which somehow mimics to a small extent the old OpenGL fixed pipeline. This shader is activated by default but in case another shader is activated at any time, you can return back to the default one with Renderer::setDefaultShader().

Note: Shaders are only used when possible. In platforms like OpenGL ES 1.1 shaders don't exist and therefore the renderer uses the fixed pipeline instead. Use of this old renderer is not recommended and is considered deprecated. The fixed pipeline has serious limitations and you lose lots of functionality when using it.

## Textures

Textures are essentially images that are uploaded to the graphics card in order to draw graphics with them.

The renderer owns a one-pixel texture colored in white which you can activate with Renderer::setDefaultTexture(). This is very useful to draw un-textured objects to the screen when the graphics system is expecting a texture to be activated. The white color will merely show a white object or multiply with the object's color, leaving it unchanged. The default shader demands this texture to be active for un-textured objects or you will only get black as final color.

## Blending

This option defines how the color being currently output to the screen will mix with the color that is already there. It is possible to just overwrite the color that is on the screen, or you can mix both colors according to the alpha of the output color. You can multiply both colors; sum them and other options that aren't useful so often.

The default option for this state is Alpha blending, which you can reset with Renderer::setDefaultBlending(). Alpha blending is ideal for 2D graphics as it will allow the alpha channel to be used to account for transparency. You can set blending modes with Renderer::setBlendMode() or enable/disable it completely with Renderer::setBlendingEnabled().

## Depth Test

Depth tests are used in order to draw 3D geometry correctly. In essence, it stores depth information in a buffer for each pixel. When a new pixel is being drawn, it is only drawn if it's closer to the camera, otherwise it's just discarded. This ensures that when drawing 3D models you don't see their back on the front and weird glitches. By default, the depth test is disabled, which is appropriate for 2D graphics.

You can reset this state with Renderer::setDefaultDepthTesting(). Then, you can control it with Renderer::setDepthTestEnabled(). It is needed that you clean the depth buffer yourself when

drawing a new frame, otherwise you won't see anything after the first frame. You can do this with Renderer::clearDepthBuffer().

### Clear color

Whenever you call Renderer::clearColorBuffer(), all the contents of the render target are cleared with a single color. You can define this color with Renderer::setClearColor().

### Viewport

The viewport defines what region of the render target is going to be drawn. Renderer::setDefaultViewport() will always enable rendering on the entire area. You can define other regions using Renderer::setViewport() or Renderer::setViewportInPixels().

### Transforms

The renderer class also accounts for three main transforms that are quite essential for rendering. These transforms are also called matrices and are represented by a mat4 object. The projection matrix defines how the "camera" that looks into your game world behaves. This can be a perspective camera which actually works as a real camera lens, distorting what can be seen according to the distance, aspect ratio and other parameters. Otherwise the projection matrix can be set as a orthogonal matrix, which means there are no perspective distortions, which makes these perfect for 2D graphics.

Then, there is the view matrix. In 3D, this usually means a transform that accounts for the camera's position and orientation while in 2D it's usually not used and left to its default state.

All transforms are set to the identity matrix by default, something you can reset using Renderer::setDefaultTransforms().

### Drawing

You can draw things to the screen in a few ways. If you are an advanced user, you will find in later sections information on how to leverage the Renderer class to its maximum potential. Otherwise you may use the built-in rendering function Renderer::draw() which makes your life much easier. Any object in the engine that inherits from Drawable is accepted as an argument to this function. This is an utility that draws your graphical object in the simplest way possible.

Examples of classes than can be used directly with the renderer draw function are Sprite, Text, RectangleShape, CircleShape, GeometryData, etc.

# Events

An event is a simple structure that carries information about what happened, to be handled by the programmer. It comes through the GameCore function onEvent() whenever one has happened.

You can refer to SFML's documentation for detailed information on events, as they work almost exactly the same here.

For short, an Event object has a *type* member which lets you check what kind of event it is. Then, for each type of event, a different member of the union that lives within it is filled. For example, if the event is a key press, type will be set with Event::KeyPressed and the actual key can be found in event.key.code.

The main addition to events in Nephilim is the fact you can handle touch screen device input through here. Now, whenever a finger is laid in the screen, the type member will be set as Event::TouchPressed. When it is again released, it is set as Event::TouchReleased and finally, while the finger is moving across the screen, you get a Event::TouchMoved. In either case, the coordinates of the touch event are (event.touch.x, event.touch.y).

Note: Multitouch support doesn't yet exist but will be added in the near future.

# Kinesis

Part of the engine dedicates to implementing 2D physics, using the infamous Box2D library. Kinesis is responsible to wrap and extend all 2D physics functionality.

All classes related to this module are prefixed with Kx to avoid conflicts and misdirections.

## KxScene

The most important class to begin with is KxScene, a direct equivalent to a Box2D world. This class is responsible to manage a set of 2D rigid bodies, perform collision detection and update the simulation. Besides the regular Box2D functionality, this class provides methods to make your life easier in many aspects, like automatically scaling the objects to an ideal size for computation, creating quick test bodies and others.

## BxBody

A BxBody corresponds to a single b2Body entity. A BxBody is a single rigid body composed by N shapes.  It can be a rectangle, a circle, an arbitrary convex polygon or even a few of these primitives put together.

## KxDraw

This class is merely a debug drawer. It is able to draw the whole scene to the screen with predetermined debug shapes. You may use this to play around with physics, debug some bug you might be having but never in production code. It is notoriously slow and not very configurable in what can be drawn.

Drawing the KxScene object to the screen is as simple as doing the following code:

```
KxDraw debugDraw(scene);
getRenderer()->draw(debugDraw);
```

## KxMousePicker

In the case you need to pickup bodies with the mouse and drag them around, there is a built-in class that handles that easily. KxMousePicker only needs to be fed some data and it just works.

You need to associate the picker with the scene by setting *KxMousePicker::scene. Then, you just need to inform the picker of the mouse movement, button press and release. Here's a simple example.

```
if(event.type == Event::MouseButtonPressed && event.mouseButton.button == Mouse::Left)
{
        picker.attachAt(event.mouseButton.x, event.mouseButton.y);
}
if(event.type == Event::MouseButtonReleased && event.mouseButton.button == Mouse::Left)
{
        picker.detach();
```

```
}
if(event.type == Event::MouseMoved)
{
        picker.update(event.mouseMove.x, event.mouseMove.y);
}
```