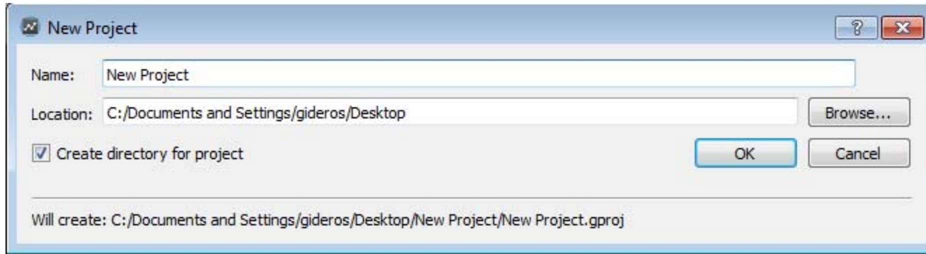
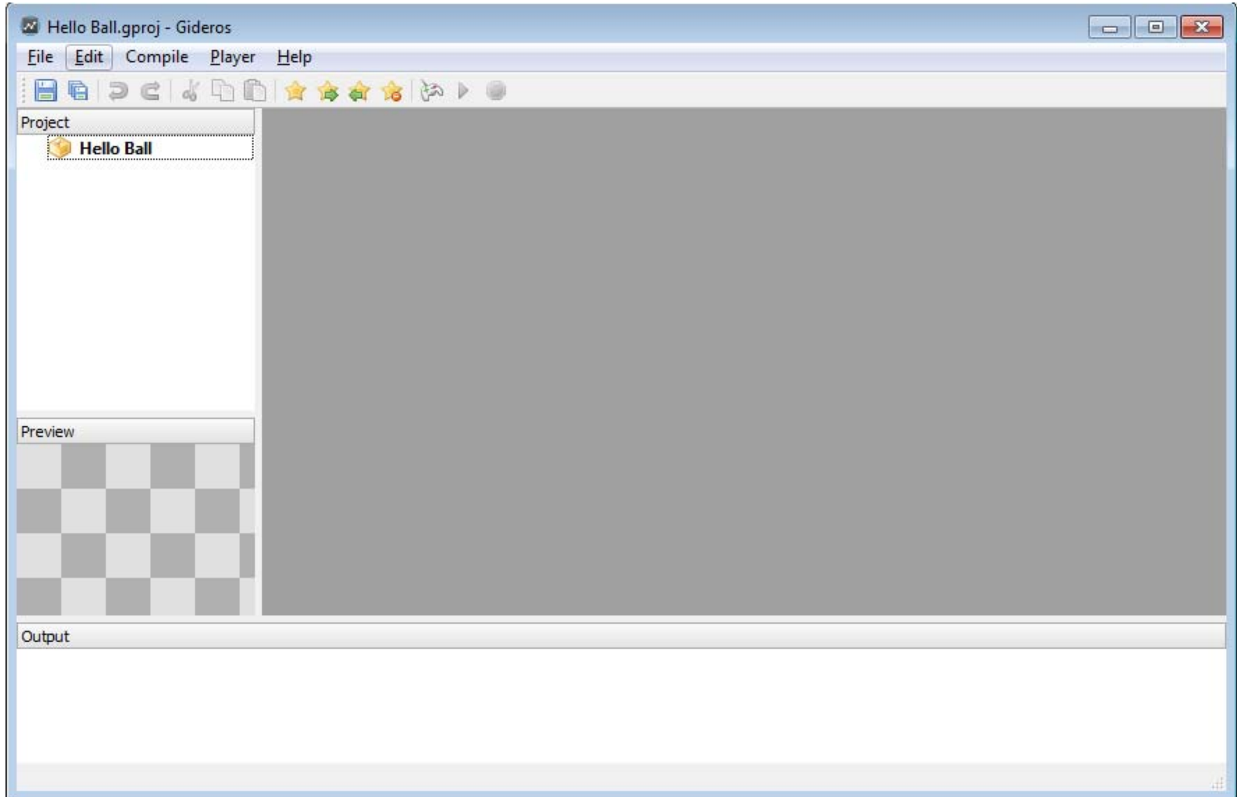
1. Getting Started

Creating New Project

First open Gideros Studio. Then create a new project from “File→New Project” menu. Name your project “HelloBall”:

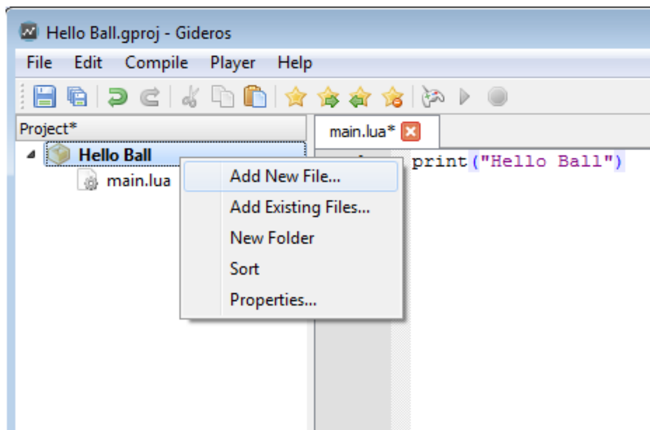


Here is our development environment where you create/manage assets and run your code:

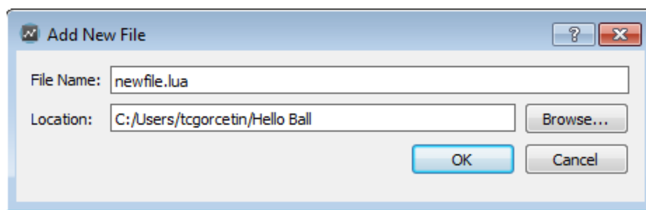


Your First Code

Right click the project name at Library and select “Add New File...” to add your first Lua code:

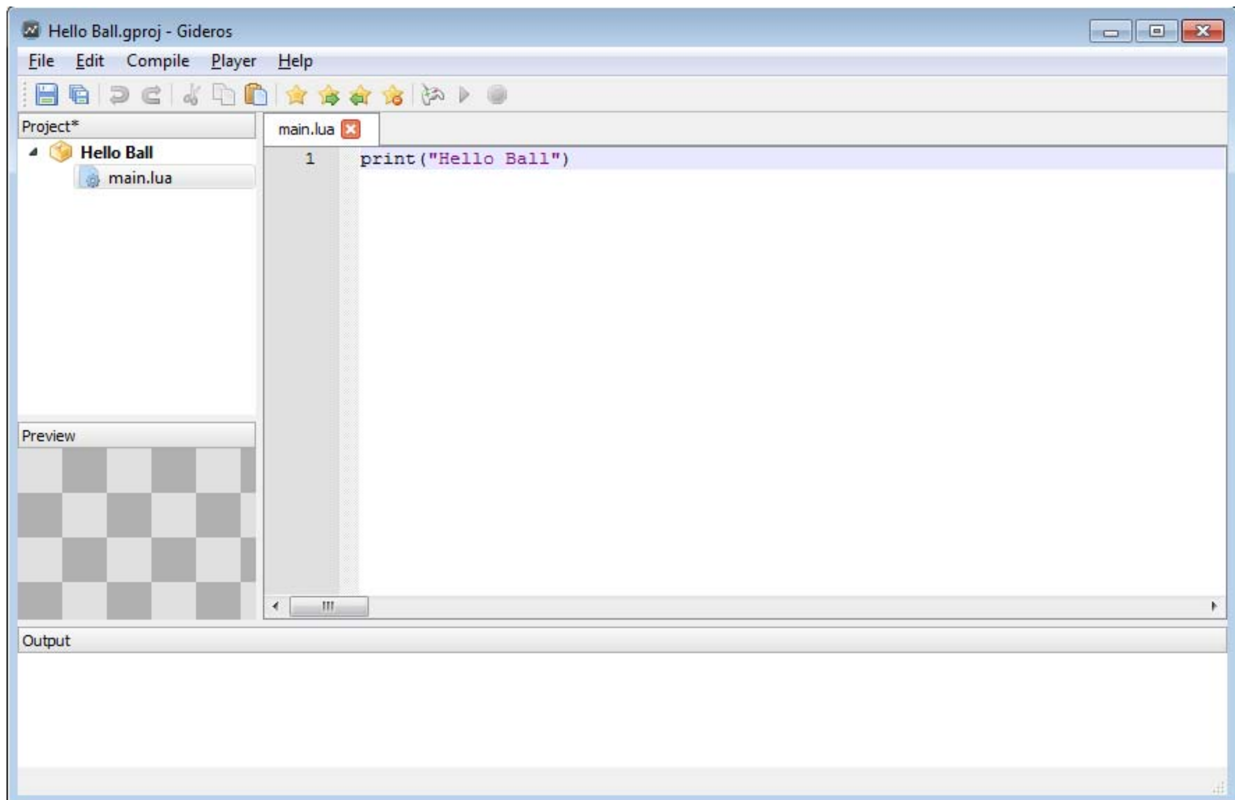


Name your file “main.lua” and click OK.



Now double-click main.lua and write

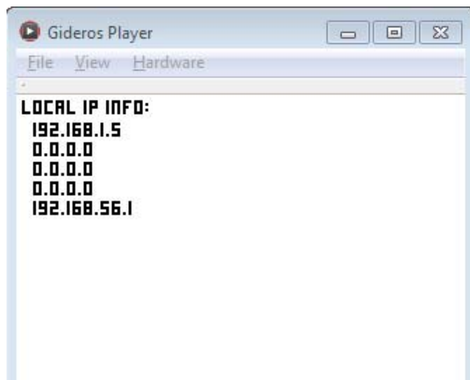
```
print("Hello Ball")
```



Running in Gideros Player

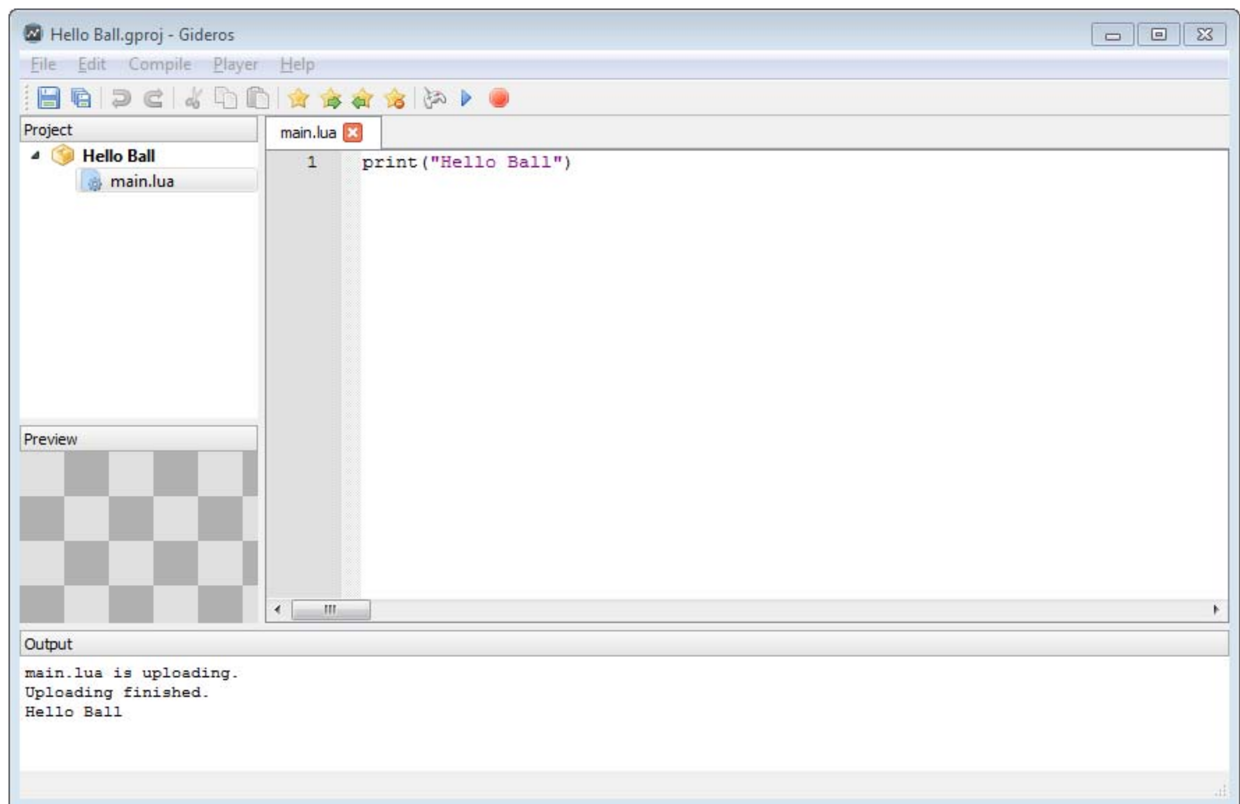
At the first part of this tutorial, we just want to run our code in “Gideos Player” and print “Hello Ball” to the console.

Now select “Player→Start Local Player” to start Gideros Player.



After “Gideros Player” opens, the start and stop icons become enabled.

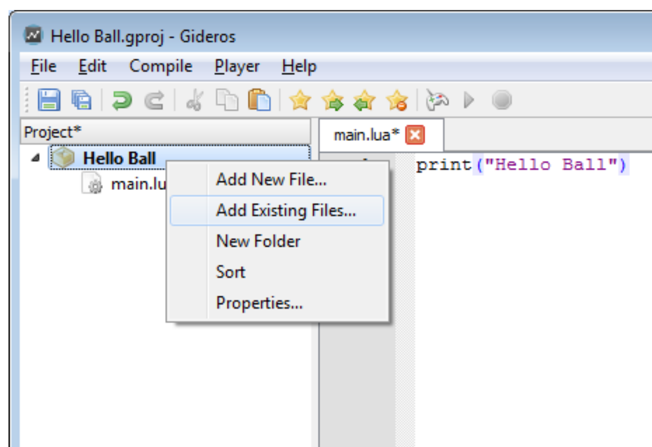
This means “Gideros Studio” connects to “Gideros Player” and ready to upload your code and assets and then run the project. Press start button (or select “Player→Start” from main menu) to run the project:



Now you see the output of your project at the "Output" panel.

Adding Assets

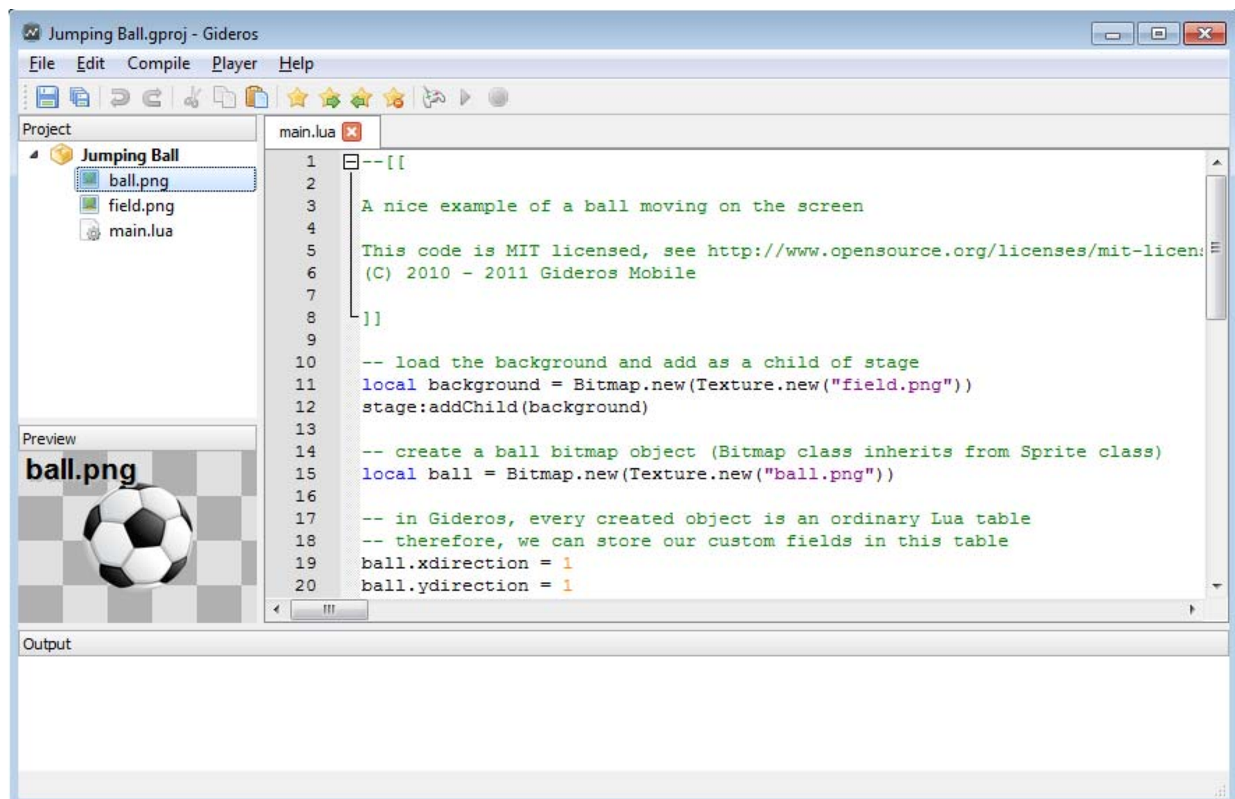
Now let's add some images to our asset library. Download field.png (assets/field.png) and ball.png (assets/ball.png) and copy these images to your project directory. Then, right click the project name at Library and select "Add Existing Files..." to add your image files to the project.



More Code

And then write the code below:

```
local background = Bitmap.new(Texture.new("field.png"))
stage:addChild(background)

local ball = Bitmap.new(Texture.new("ball.png"))

ball.xdirection = 1
ball.ydirection = 1
ball.xspeed = 2.5
ball.yspeed = 4.3

stage:addChild(ball)

function onEnterFrame(event)
    local x = ball:getX()
    local y = ball:getY()

    x = x + (ball.xspeed * ball.xdirection)
    y = y + (ball.yspeed * ball.ydirection)

    if x < 0 then
        ball.xdirection = 1
    end

    if x > 320 - ball:getWidth() then
        ball.xdirection = -1
    end

    if y < 0 then
        ball.ydirection = 1
    end

    if y > 480 - ball:getHeight() then
        ball.ydirection = -1
    end

    ball:setX(x)
    ball:setY(y)
end

stage:addEventListener(Event.ENTER_FRAME, onEnterFrame)
```



After pressing start, you'll have a nice ball moving around and bouncing from the edges:

Running on Device Player

IOS device

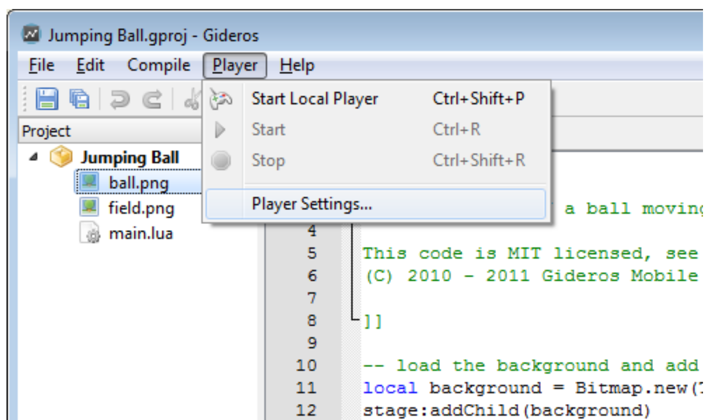
To run project on IOS device you need to build and install the GiderosiOSPlayer XCode project (GiderosiOSPlayer.zip) comes with the installation. You need to be an approved Apple Developer for iOS and need to install the XCode with iOS SDK.

Android device

To run project on Android device you need to install the GiderosAndroidPlayer.apk (comes with the installation) on your device.

Run project

After installing GiderosPlayer to your device, open the player and enter the IP of your device (which will be shown in Gideros Player) from the menu “Player→Player Settings”.



When the Start and Stop buttons appear enabled, press Start to run your code on device.

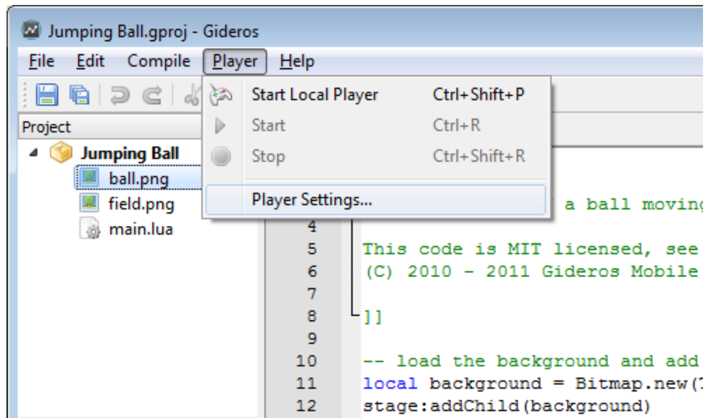
2. Deployment

This document will discuss the build and deployment processes of Gideros Player and your application to devices.

For IOS devices, you must be an approved Apple Developer for iOS and install the iOS SDK (requires Intel-based Mac running Snow Leopard).

For Android devices you'll need Android SDK and Eclipse or other Android project compatible IDE.

Gideros Player for devices allows you to see and test your application on device **instantly**. After you deploy and open Gideros Player on your device, you see the IP of your device. Enter this IP from “Player→Player Settings” menu:



When you press start button, your codes and asset files are transferred to your iOS device via WiFi and your application starts.

Deploy Gideros Player to Devices

For IOS

There is a universal project `GiderosiOSPlayer.zip`, which comes with the installation. You need to extract this zip file and open, build, deploy this project from Xcode.

For Android

There is a ready to use application for Android development `GiderosAndroidPlayer.apk` which comes with the installation. You can install it directly on your usin USB data transfer cable, or memory card on your phone, or uploading it to a server (as dropbox) and accessing it from there.

Deploy Your Application to iOS Devices

To deploy your application to iPhone/iPad you need to:

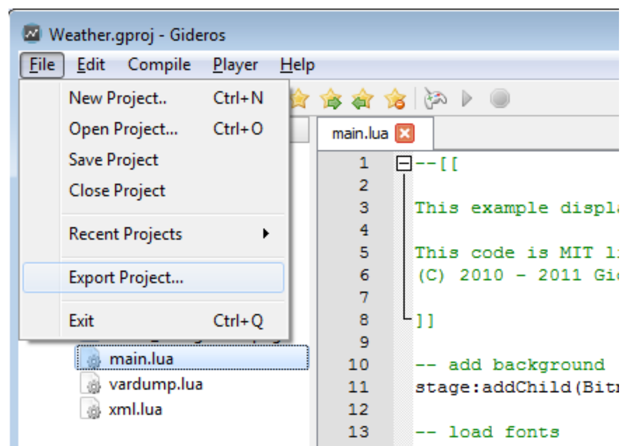
1. Export your application as Xcode project
2. Build and deploy Xcode project to actual device

For the second step, you need Intel-based Mac running Snow Leopard and Apple Developer License.

To deploy your application to Android you need to:

1. Export your application as Android project
2. Import exported project into Eclipse
3. Run project from Eclipse

First step is done through “File→Export Project” menu:



Device player with plugins

Similar to how you can deploy players on devices, you can also build a custom player, with built in plugins for specific platforms.

For example, if you have an Ad framework plugin for Android, you simply build Gideros Android player with this plugin and deploy it to device, to test it as you would with normal Android device player.

To deploy custom player with plugins to Android

1. Export your Gideros project as Android project
2. Add plugin as described in your specific plugin installation instruction
3. In your exported project, inside assets folder there is another assets folder, simply delete it, leaving upper assets folder empty
4. Build this project to run on your device
5. You now have Gideros Android player which behaves exactly the same as normal player, but has the functionalities of applied plugin

To deploy custom player with plugins to IOS

1. Export your Gideros project as IOS project
2. Add plugin as described in your specific plugin installation instruction
3. In your exported project, open AppDelegate.m file
4. Change gdr_initialize(..., false) to gdr_initialize(..., true)
5. You now have Gideros IOS player which behaves exactly the same as normal player, but has the functionalities of applied plugin

3. Classes in Gideros

Lua does not support classes the way that languages like C++, Java and ActionScript do. But Lua is a multi-paradigm language and have roots from prototype-based languages. In Lua, each object can define its own behaviour through metatables. Therefore, it is possible to emulate OO programming and classes in Lua.

(For the detailed discussion of object oriented programming in Lua, please refer to http://www.lua.org/pil/16.html (http://www.lua.org/pil/16.html))

Gideros follows the same paradigm in its API design. Each instance created by Gideros API is a Lua table with a metatable attached.

Creating Instances

Instances in Gideros is created through `new` function. For example, to create a `Sprite`, `Texture`, `Bitmap` and a `Timer` instance:

```
local sprite = Sprite.new()
local texture = Texture.new("image.png")
local bitmap = Bitmap.new(texture)
local timer = Timer.new(1000, 0)
```

Inheritance

`Core.class` function is used to create your own classes through inheritance. You can create your own classes like:

```
MyClass = Core.class()
```

or inherit from Gideros API's own classes (`EventDispatcher`, `Sprite`, etc.). For example, you can create your EventDispatcher class as:

```
MyEventDispatcher = Core.class(EventDispatcher)
```

By using Inheritance, you can design and implement the visual elements of your game separately:

```
StartButton = Core.class(Sprite)     -- create your own start button class
Menu = Core.class(Sprite)            -- create your own menu class

Player = Core.class(Sprite)          -- create your own player class

function Player:walk()
    -- walk logic
end

function Player:jump()
    -- jump logic
end

stage:addChild(Player.new())            -- create and add a player instance to the stage
```

When an instance is created, `init` function is called to do the initialization:

```
Player = Core.class(Sprite)

function Player:init()
    -- do the initialization of Player instance
    self.health = 100
    self.speed = 3
end

local player = Player.new()     -- after Player instance is created, init function is called
```

Whether to use inheritance or not is related to your programming taste. It's possible to implement a whole game without creating custom classes. You can refer to "Jumping Ball" and "Jumping Balls" examples to see the difference between designing your code with classes or not.

---

4. Events

Events are the central mechanism to handle responses and they allow to create interactive applications.

All classes that dispatch events inherit from `EventDispatcher`. The target of an event is a listener function and an *optional* data value. When an event is dispatched, the registered function is called. If the optional data value is given, it is used as a first parameter while calling the listener function.

In Gideros, events can be divided into two categories: *built-in events* which are generated by the system (e.g. ENTER_FRAME event, touch events, timer events, etc.) and *custom events* which can be generated by the user. According to their event type, *built-in events* can be broadcasted to multiple targets (e.g. ENTER_FRAME event, touch events, etc.) or can be dispatched to a single target (e.g. timer event).

ENTER_FRAME Event

The Gideros runtime dispatches the built-in `Event.ENTER_FRAME` event to `Sprite` instances before rendering the screen. Visual changes made by any `Event.ENTER_FRAME` listener function will be visible at next frame.

This first basic example shows a moving sprite one pixel to the right at each frame. In this example, `onEnterFrame` function increases the x-coordinate of a sprite object at each frame:

```
local sprite = Sprite.new()

local function onEnterFrame(event)
    sprite:setX(sprite:getX() + 1)
end

sprite:addEventListener(Event.ENTER_FRAME, onEnterFrame)
```

The next example shows 3 independent sprites moving one pixel to the right at each frame. In this example, we use the optional data parameter to move independent sprites with one common listener function:

```
local sprite1 = Sprite.new()
local sprite2 = Sprite.new()
local sprite3 = Sprite.new()

local function onEnterFrame(self, event)
    self:setX(self:getX() + 1)
end

sprite1:addEventListener(Event.ENTER_FRAME, onEnterFrame, sprite1)
sprite2:addEventListener(Event.ENTER_FRAME, onEnterFrame, sprite2)
sprite3:addEventListener(Event.ENTER_FRAME, onEnterFrame, sprite3)
```

The last example shows subclassing of the `Sprite` class and registering `Event.ENTER_FRAME`:

```
MySprite = Core.class(Sprite)

function MySprite:init()
    self:addEventListener(Event.ENTER_FRAME, self.onEnterFrame, self)
end

function MySprite:onEnterFrame(event)
    self:setX(self:getX() + 1)
end
```

Note: `Event.ENTER_FRAME` event is dispatched to all `Sprite` instances no matter these instances are on the scene tree or not.

Mouse, Touch and Key Events

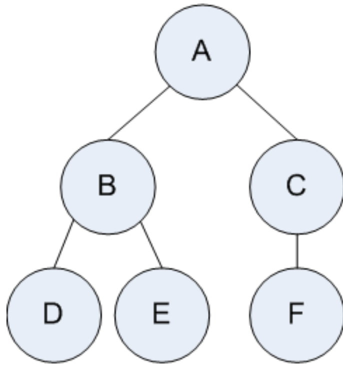
Gideros runtime dispatches mouse and touch events when the the user's finger touches the screen. Mouse events are mainly used in single-touch whereas touch events are used in multi-touch applications. Key events are dispatched when user presses and releases a physical key on the keyboard.

The mouse, touch and key events are dispatched to `Sprite` instances which are on the scene tree. If a `Sprite` instance is not on the scene tree, this instance doesn't receive mouse, touch and key events.

Note: Even if touch or mouse doesn't hit the `Sprite` instance, the instance receive mouse/touch events.

The order of dispatch is determined by the hierachy of the scene tree. The `Sprite` instance that is drawn last (top-most sprite) receives the event first. The next sprite at the bottom of the top-most sprite receives the event second and so on.

For example, assume that we have an sprite hierachy like this:



which is constructed by the code below:

```
local A = Sprite.new()
local B = Sprite.new()
local C = Sprite.new()
local D = Sprite.new()
local E = Sprite.new()
local F = Sprite.new()

A:addChild(B)
A:addChild(C)
B:addChild(D)
B:addChild(E)
C:addChild(F)
```

In this hiearchy, the drawing order is A, B, C, D, E, F while mouse/touch event receive order is F, E, D, C, B, A.

Stopping an Event Dispatch

It is possible to stop the propagation of mouse, touch and key events. To stop an event dispatch, invoke the `Event:stopPropagation()` function on the `Event` object passed to the listener function. In this example below, `MOUSE_DOWN` event is dispatched only to F, E, D and C:

```
local A = Sprite.new()
local B = Sprite.new()
local C = Sprite.new()
local D = Sprite.new()
local E = Sprite.new()
local F = Sprite.new()

-- stop propagation at sprite C
C:addEventListener(Event.MOUSE_DOWN, function(event) event:stopPropagation() end)

A:addChild(B)
A:addChild(C)
B:addChild(D)
B:addChild(E)
C:addChild(F)
```

Timer Events

The `Timer` class is used for executing code at specified time intervals. Each `Timer` object dispatches `Event.TIMER` event at specified frequency.

The steps to use `Timer` class are as follows:

Create a new `Timer` object with specified frequency and specified total number of `Event.TIMER` events to be triggered. For example, the following code sets the frequency to 1000 miliseconds and sets the count to 5.

```lua
local timer = Timer.new(1000, 5)
```

Register to the `Event.TIMER` event with a listener function:

```lua
local function onTimer(event)
    -- will be executed 5 times at 1000 miliseconds intervals
end

timer:addEventListener(Event.TIMER, onTimer)
```

Start the timer.

```lua
timer:start()
```

To stop the timer, you can use `Timer:stop()` function:

```lua
timer:stop()
```

`Event.TIMER_COMPLETE` event is triggered after finishing the specified number of timer events.

```lua
local function onTimerComplete(event)
    -- will be executed after the specified number of timer events (5) are dispatched
end

timer:addEventListener(Event.TIMER_COMPLETE, onTimerComplete)
```

Also, it is possible to pause and resume all the timers in your application. It is very useful when you are implementing a pause/resume functionality in your game:

```lua
Timer.pauseAllTimers()      -- pause all timers. if all timers are alredy paused, does nothing.
Timer.resumeAllTimers()     -- resume all timers. if all timers are alredy running, does nothing.
```

ADDED_TO_STAGE and REMOVED_FROM_STAGE Events

If a sprite is added to the scene tree, the sprite instance and all of it's descendants receive `Event.ADDED_TO_STAGE` event. Similarly, if a sprite is removed from the scene tree, the sprite instance and all of it's descendants receive `Event.REMOVED_FROM_STAGE` event. These events are used to detect when a `Sprite` instance is added to, or removed from, the scene tree. For example, by the help of these events, it is possible to register `Event.ENTER_FRAME` event only for the sprites that are on the scene tree:

```lua
MySprite = Core.class(Sprite)

function MySprite:init()
    self:addEventListener(Event.ADDED_TO_STAGE, self.onAddedToStage, self)
    self:addEventListener(Event.REMOVED_FROM_STAGE, self.onRemovedFromStage, self)
end

function MySprite:onAddedToStage(event)
    self:addEventListener(Event.ENTER_FRAME, self.onEnterFrame, self)
end

function MySprite:onRemovedFromStage(event)
    self:removeEventListener(Event.ENTER_FRAME, self.onEnterFrame, self)
end

function MySprite:onEnterFrame(event)
    -- enter frame logic
end
```

Custom Events

To dispatch a new custom, user defined event, create the event with `Event.new()` function and dispatch it with `EventDispatcher:dispatchEvent()`.

```lua
ClassA = Core.class(EventDispatcher)
ClassB = Core.class(EventDispatcher)

function ClassA:funcA(event)
    print("funcA", self, event:getType(), event:getTarget())
end

local a = ClassA.new()
local b = ClassB.new()

b:addEventListener("myevent", a.funcA, a)   -- when b dispatches an "myevent" event,
                                            -- a.funcA will be called with 'a'
                                            -- as first parameter

b:dispatchEvent(Event.new("myevent"))       -- will print "funcA"
```

List of all Built-in Events

ENTER_FRAME

Dispatched to all `Sprite` instances before rendering the screen.

> **event.frameCount**: The total number of frames that have passed since the start of the application
> **event.time**: Time in seconds since the start of the application
> **event.deltaTime**: The time in seconds between the last frame and the current frame

ADDED_TO_STAGE

Dispatched when target `Sprite` instance is added to the stage.

REMOVED_FROM_STAGE

Dispatched when target `Sprite` instance is removed from the stage.

MOUSE_DOWN

Dispatched to all `Sprite` instances on the scene tree when user presses the mouse button or starts the first touch.

> **event.x**: The x-coordinate of the mouse or touch
> **event.y**: The y-coordinate of the mouse or touch

MOUSE_MOVE

Dispatched to all `Sprite` instances on the scene tree when user moves the mouse or moves the first touch.

> **event.x**: The x-coordinate of the mouse or touch
> **event.y**: The y-coordinate of the mouse or touch

MOUSE_UP

Dispatched to all `Sprite` instances on the scene tree when user releases the mouse button or ends the first touch.

> **event.x**: The x-coordinate of the mouse or touch
> **event.y**: The y-coordinate of the mouse or touch

TOUCHES_BEGIN

Dispatched to all `Sprite` instances on the scene tree when one or more fingers touch down.

> **event.touch**: A table with fields x, y and id which specifies the coordinates and id of the current touch
> **event.allTouches**: Array of all touches where each element contains x, y and id

For example, you can print the x, y and id properties of current touch and all touches as:

```lua
local function onTouchesBegin(event)
    print(event.touch.x, event.touch.y, event.touch.id)
    for i=1,#event.allTouches do
        print(event.allTouches[i].x, event.allTouches[i].y, event.allTouches[i].id)
    end
end
```

TOUCHES_MOVE

Dispatched to all `Sprite` instances on the scene tree when one or more fingers move.

> **event.touch**: A table with fields x, y and id which specifies the coordinates and id of the current touch
> **event.allTouches**: Array of all touches where each element contains x, y and id

TOUCHES_END

Dispatched to all `Sprite` instances on the scene tree when one or more fingers are raised.

> **event.touch**: A table with fields x, y and id which specifies the coordinates and id of the current touch
> **event.allTouches**: Array of all touches where each element contains x, y and id

TOUCHES_CANCEL

Dispatched to all `Sprite` instances on the scene tree when a system event (such as a low-memory warning) cancels a touch event.

> **event.touch**: A table with fields x, y and id which specifies the coordinates and id of the current touch
> **event.allTouches**: Array of all touches where each element contains x, y and id

KEY_DOWN

Dispatched to all `Sprite` instances on the scene tree when user presses a physical key on the keyboard.

> **event.keyCode**: The key code value of the key pressed

KEY_UP

Dispatched to all `Sprite` instances on the scene tree when user releases a physical key on the keyboard.

> **event.keyCode**: The key code value of the key released

APPLICATION_START

Dispatched to all event listeners (broadcast event) right after the application is launched and all Lua codes are executed.

APPLICATION_EXIT

Dispatched to all event listeners (broadcast event) when the application is about to exit. If an application is forced to be terminated (e.g. by double tapping the home button and kill the application), this event may not be dispatched. If you want to save your game state before exiting, save your state also on `APPLICATION_SUSPEND` event.

APPLICATION_SUSPEND

Dispatched to all event listeners (broadcast event) when when the application is about to move from the active to inactive state. When an application is inactive, `Event.ENTER_FRAME` and `Event.TIMER` events are not dispatched until the application is resumed.

APPLICATION_RESUME

Dispatched to all event listeners (broadcast event) when the application is moved from the inactive to active state.

APPLICATION_BACKGROUND

Dispatched to all event listeners (broadcast event) when the application is now in the background.

APPLICATION_FOREGROUND

Dispatched to all event listeners (broadcast event) when the application is about to enter the foreground.

TIMER

Dispatched whenever a `Timer` object reaches an interval specified according to the delay property.

TIMER_COMPLETE

Dispatched whenever a `Timer` object has completed the number of requests specified according to the repeatCount property.

COMPLETE

Dispatched when:

> a sound channel has finished playing.
> after all data is received and placed in the `event.data` field

ERROR

Dispatched when `UrlLoader` fails and terminates the download.

PROGRESS

Dispatched by `UrlLoader` as the notification of how far the download has progressed.

event.bytesLoaded: The number of bytes loaded
event.bytesTotal: The total number of bytes that will be loaded

BEGIN_CONTACT

Dispatched by `b2.World` when two fixtures begin to overlap. This is dispatched for sensors and non-sensors. This event can only occur inside the time step.

event.contact: The contact
event.fixtureA: The first fixture in this contact
event.fixtureB: The second fixture in this contact

END_CONTACT

Dispatched by `b2.World` when two fixtures cease to overlap. This is dispatched for sensors and non-sensors. This may be dispatched when a body is destroyed, so this event can occur outside the time step.

event.contact: The contact
event.fixtureA: The first fixture in this contact
event.fixtureB: The second fixture in this contact

PRE_SOLVE

Dispatched by `b2.World` after collision detection, but before collision resolution.

event.contact: The contact
event.fixtureA: The first fixture in this contact
event.fixtureB: The second fixture in this contact

POST_SOLVE

Dispatched by `b2.World` after collision resolution.

event.contact: The contact
event.fixtureA: The first fixture in this contact
event.fixtureB: The second fixture in this contact
event.maxImpulse: Maximum of the normal impulses

5. File system

In Gideros runtime, there are 3 kinds of directories: **resource**, **document**, and **temporary**.

You can access these directories using the `io` library provided by Lua:

```
io.read("data/list.txt")
```

You don't need to know the exact path of resource, document and temporary directories because Gideros provides an easy way to specify files at these directories.

To sum up:

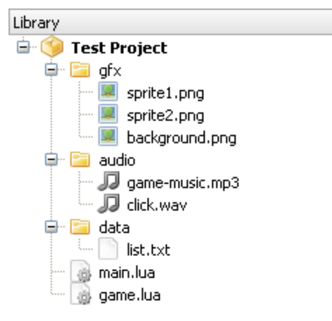
> **Resource** - stores your code and assets (can not be modified by the app)
> **Document** - can be used as persistent storages for files (can be modified by the app)
> **Temporary** - can be used as temporary storages for files (can be modified by the app)

Example of accessing each directory:

```
io.read("file.txt")       --> open file.txt at resource directory to read
io.read("|R|file.txt")    --> open file.txt at resource directory to read (same as above)
io.read("|D|file.txt")    --> open file.txt at documents directory to read
io.read("|T|file.txt")    --> open file.txt at temporary directory to read
```

Resource directory

Your code, image, audio and all other files are reside at *resource directory*. The file and folder structure of your asset library shown below



is stored at real device and Gideros Player like:

```
{resource directory}/gfx/sprite1.png
{resource directory}/gfx/sprite2.png
{resource directory}/gfx/background.png
{resource directory}/audio/game-music.mp3
{resource directory}/audio/click.wav
{resource directory}/data/list.txt
{resource directory}/main.lua
{resource directory}/game.lua
```

Resource directory is the default directory. Therefore, to access the files at resource directory, specify the file path as it is:

```
local sprite1 = Texture.new("gfx/sprite1.png")
local sprite2 = Texture.new("gfx/sprite2.png")
local background = Texture.new("gfx/background.png")
local music = Sound.new("audio/game-music.mp3")
local click = Sound.new("audio/click.wav")
```

Note: Optionally, you can access the files at resource directory by adding `"|R|"` to the begining of the file name (but you don't need to):

```
local sprite1 = Texture.new("|R|gfx/sprite1.png")
```

Note: Resource directory is *read-only* and you should not try to write any files there.

Document directory

You can store application created files at *document directory*. The files created at document directory is permanent between different application sessions. For example, you can create and then read files at document directory to save player progress. To specify a file at document directory, append `"|D|"` to the begining of the file name:

```
io.write("|D|save.txt")
```

The main advantage of *document directory* are that:

Files can be modified
Files can be stored persistantly

That is for example it is recommended to store database files or other user generated information in *document directory*

Here is a quick example how you can copy file from *resource directory* to *document directory*:

```
--function to copy file
local function copy(src, dst)
    local srcf = io.open(src, "rb")
    local dstf = io.open(dst, "wb")

    local size = 2^13      -- good buffer size (8K)
    while true do
        local block = srcf:read(size)
        if not block then break end
        dstf:write(block)
    end

    srcf:close()
    dstf:close()
end

--function to check if file exists
local function exists(file)
    local f = io.open(file, "rb")
    if f == nil then
        return false
    end
    f:close()
    return true
end

--usage
if not exists("|D|database.db") then
    copy("database.db", "|D|database.db")
end
```

Temporary directory

You can create and store temporary files at *temporary directory*. The files created at temporary directory are not guaranteed to exists between different application sessions. They may be deleted after your application session finishes. To specify a file at temporary directory, append `"|T|"` to the begining of the file name:

```
io.write("|T|temp.txt")
```

This storage may be used for example, to display some temporary data, like images downloaded from somewhere:

```lua
--download completed
local function onComplete(event)

        --store image in temporary folder
        local out = io.open("|T|image.png", "wb")
    out:write(event.data)
    out:close()

        --display it to user
    local b = Bitmap.new(Texture.new("|T|image.png"))
        b:setAnchorPoint(0.5, 0.5)
        b:setPosition(160, 240)
    stage:addChild(b)
end

--load image
local loader = UrlLoader.new("http://www.giderosmobile.com/giderosmobile.png")

--add event listener
loader:addEventListener(Event.COMPLETE, onComplete)
```

Lua Files and Execution Order

By default Gideros executes all lua files on both players and in real app.

We can assume that the order of execution is pretty random, but there are two things guaranteed:

- **init.lua** will always be executed first
- **main.lua** will always be executed last

So the best practice is:

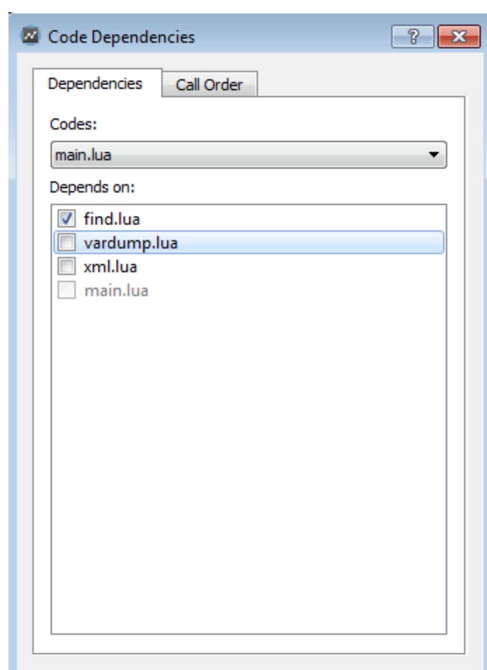
- to add all additional functionality and modifications to existing classes in init.lua
- do all initialization of app etc in main.lua (when all other code was already loaded)
- wrap the code in any other lua file in a scope as function

Yet better create each lua file as a separate Gideros class, either it will be a scene shown in scene manager or some simple object represented by class, but there should not be some plain code executed in these files, only in main.lua

Of course, all can be configured

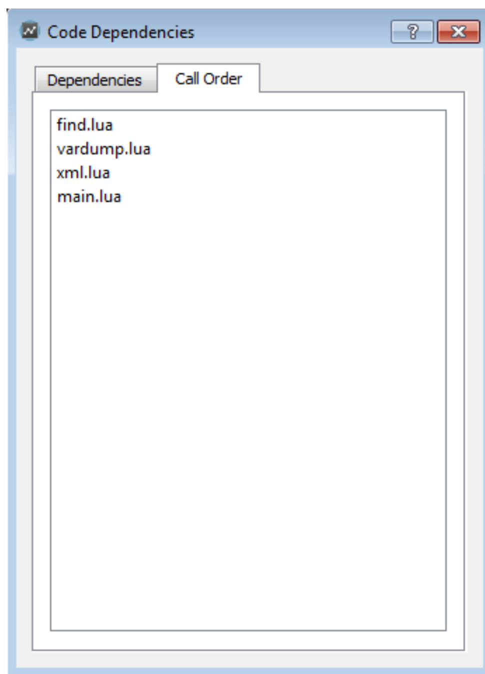
Before an Gideros application starts, all Lua files at asset library are executed one by one. So it is possible to arrange the order of execution by setting the code dependencies between Lua files.

If you right click a Lua file and select “Code Dependencies…” from popup menu, “Code Dependencies” dialog opens:



In this menu, you can set the dependencies between Lua files. For example, if `a.lua` is dependent to `b.lua`, `b.lua` is always executed befure `a.lua`.

If you select “Call Order” tab, you can see the execution order:



Note: The file names `main.lua` and `init.lua` have special meaning: When an application starts, Gideros runtime tries to execute `init.lua` first and `main.lua` last.

---

strict.lua

For the detailed explanation of `strict.lua`, please refer to http://www.lua.org/pil/14.2.html (http://www.lua.org/pil/14.2.html)

`strict.lua` checks uses of undeclared global variables. If `strict.lua` is executed, all global variables must be 'declared' through a regular assignment (even assigning nil will do) in a main chunk before being used anywhere or assigned to inside a function. Although optional, it is a good habit to use it when developing Lua code.

To execute `strict.lua` before all other Lua files, simply add `strict.lua` and `init.lua` to asset library and make `strict.lua` dependent to `init.lua`.

You can download `strict.lua` from here (assets/strict.lua) that originally comes with the Lua distribution.

---

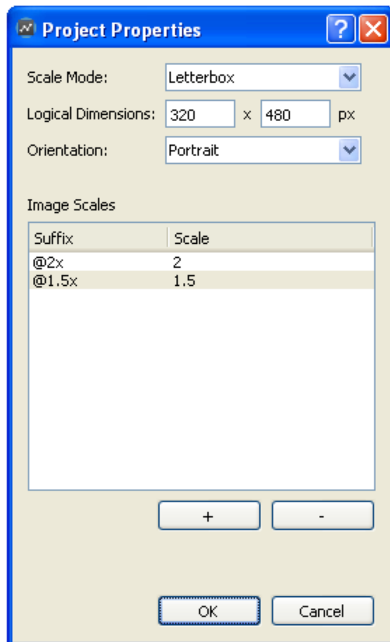
6. Automatic Image Resolution

With the introduction of iPad, iPhone 4 and variety of devices running Android, now there are multiple resolutions that game developers should consider. To efficiently use texture memory and processing power of GPUs, applications should use low-resolution images on devices with low-resolution screens and high-resolution images on devices with high-resolution screens. With the help of *automatic image resolution*, you can bundle both low and high resolution images together with your application and Gideros automatically selects the best resolution according to your scaling.

Automatic image resolution is directly related to automatic screen scaling. For example, if your screen is automatically scaled by 2, then using the double-sized image is the best choice in terms of quality and efficiency.

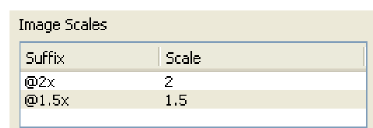
Open the project "Hardware/Automatic Image Resolution" to see automatic image resolution in action. Run this example multiple times by selecting different resolutions on Gideros Player such as 320x480, 640x960 and 480x800. As you can see, for the different resolutions, the original 200x200 image or high-resolution variants (300x300 and 400x400) are selected and displayed automatically.

Now right click on the project name and select "Properties..." to understand how we configure automatic image resolution parameters:



In this example, our logical dimensions are 320x480 and scale mode is Letterbox. So the scaling factor for a device with screen resolution 320x480 (older iPhones) is 1, scaling factor for 480x960 (iPhone 4) is 2 and scaling factor for 480x800 (Samsung Galaxy S) is around 1.5.

As you can see, we've configured image scales as:



So if you have a base image with resolution 200x200 with name "image.png", you provide these 3 images:

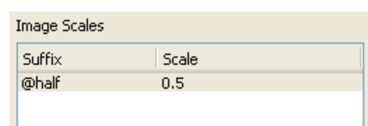
> image.png (200x200)
> image@1.5x.png (300x300)
> image@2x.png (400x400)

and let Gideros pick the appropriate one.

Providing the alternative images (in our example, these are image@1.5x.png and image@2x.png) is optional but you should always provide the base image (image.png). When Gideros cannot find the alternative image to load, it loads the base image. Also size calculations are done according to the size of the base image.

Design for High Resolution Devices

In our example, we set the logical dimensions as 320x480 and provide higher-resolution alternative images. On the other hand, it's possible to set logical dimensions as 480x960 or 768x1024 and provide lower-resolution images. In this case, we can configure the image scales like:



and provide alternative images with suffix “@half” as half-resolution of the original ones.

Bitmap Fonts and Texture Packs

Automatic image resolution works with bitmap fonts and texture packs also. Assume you create a bitmap font with size 20 and export it as “font.txt” + “font.png”. To obtain the double-resolution alternative, export the same font with size 40 as “font@2x.txt” + “font@2x.png”.

A texture pack is a large image which contains many smaller sub-images. There is one thing to remember while creating multi-resolution texture packs: the names of the sub-images must be same across the texture pack variants. A practical way to achieve this is to store different resolution sub-image sets into their own directories. You can check the sample “Hardware/Automatic Texture Pack Resolution” about this.

---

7. Automatic Screen Scaling

To handle a wide variety of resolutions, Gideros provides a functionaly called *Automatic Screen Scaling*.

Before starting your project, you determine your *logical resolution* and position all your sprites according to this. For example, if you determine your logical resolution as 320x480, your upper left corner will be (0, 0), your lower right corner will be (319, 479) and the center coordinate of your screen will be (160, 240). Then according to your scale mode, Gideros automaticly scales your screen according to the real resolution of your hardware.

There are 8 types of scaling modes:

1. **No scale:** No scaling simply tells your application not to scale at all. In this mode, there's no scaling at all, and your stage sits on the upper left corner of the device screen.
2. **No scale, center:** This mode is similar to "no scale", except one minor difference: Stage is centered on the device screen.
3. **Pixel perfect:** This mode is similar to "No scale, center", with a minor difference: Scale value is rounded to values like 1/3, ½, 1, 2 or 3. This mode can be used in games which benefit from pixel art games - it guarantees a crisp look and feel for screen sprites.
4. **Letterbox:** Preserving the aspect ratio, it scales the stage so it fits the content on the device screen. There's a possibility that blank areas may occur on the device. Mostly, developer keeps the background a bit "bigger" in order to eliminate these blank areas and fill it with background.
5. **Crop:** This mode again does preserve the aspect ratio, and ensures that no blank areas are left on the screen. Hence, some part of the background may be outside the visible area.
6. **Stretch:** The whole stage sits on the screen. Since x/y is not preserved, there's a possibility that aspect ratio might change.
7. **Fit width:** This mode fits the width of the stage, preserving the aspect ratio. It's possible that there might be blank areas at the top and bottom of the screen, or these areas may be outside the visible area. If you have more than one screen in your application, and screen transition is from left to right (or vice versa), then this mode is a perfect fit for you.
8. **Fit height:** Similar to fit width, this time guaranteeing to fit the height. Very suitable for shoot'em-up type games.