Pixelperfection in World of Warcraft

In this article I will explain what Pixelperfection in World of Warcraft is and my technique to achieve it. Knowledge of Lua and the WoW API is required.

What is PP?

There are two aspects to PP in World of Warcraft. Placement and anti-aliasing. Placement means that you have full control (absolute positioning/sizing) over the placement and size of pixels you place as a UI designer, no matter what resolution the UI is deployed under. Whether the screen has a resolution of 800x600 or 1440x900, every pixel you design will look the same on every users screen. PP with regard to anti-aliasing simply means disabling anti-aliasing to ensure a razor sharp one-pixel border.

Disabling anti-aliasing

The anti-aliasing technique implemented and used by World of Warcraft is called multisampling. It can be set from 1x (disabled) to 8x through the Video configuration panel accessed in-game. If that doesn't work you must override the anti-aliasing for WoW through a configuration panel for your video card. For nVidia cards you can easily do this through nHancer.

Placement and sizing

There are four major factors that decide how a frame is rendered. Dimensions, anchors, scale and ofcourse the rendering engine. Our friends are dimensions and anchors, our enemies are scale and the rendering engine.

The rendering engine

The WoW rendering engine thinks that every screen has a height of 768 pixels and a width relational to that (for example, on a 4:3 screen the WoW engine thinks the screen has a width of 1024 pixels). But in reality your screen has a different amount of pixels, causing each dimensions' pixels to be blown up by the factor `resoheight/768`, snapped to the closest pixel (AKA rounded). In order to be able to give frames absolute dimensions in pixels we have to compensate for that. By multiplying each dimension with the value `768/resoheight` we do that because those two factors cancel eachother out: `(resoheight/768)*(768/resoheight) = 1`.

Scale

Every frame has a scale factor of its own, but each frame also inherits the scale(s) of the parent tree above that frame. All these scale factors add up to the **effective scale**. The dimensions of your frame will be multiplied by this factor, so we have to multiply the frame's dimensions by `1/effectivescale` (once again, those factors cancel eachother out). How can we get the effective scale? We could use `frame:GetEffectiveScale()` for every frame, but that would be a lot of CPU usage and requires constant updating. Instead we control the effective scale by using three rules when we create our

frames.

- Don't scale your frames.
- Make sure that the frames' parents don't use scale.
- Make sure that the parent tree above your frame ends with the frame UIParent.

These three rules should ensure that every frame created by us will have an effective scale equal to the UIParent's scale, AKA UIScale. Luckily Blizzard uses these *rules* too (mostly).

UIScale
Compensating for the UIScale has an easy part and a hard part. The easy part is the actual compensating, just multiply each dimension with `1/uiscale`. The hard part is actually finding out the UI scale. For some reason Blizzard made it impossible to retrieve the UI Scale CVar during the first code load, it is only available after the event `VARIABLES_LOADED`. I have thought a long time on a solution for this, without success. In the end I just hard-coded a value in my UI package and made sure the UI Scale is set to that (known) value.

Lua implementation

Now our knowledge is up to date, let's implement it in Lua and make it actually work! First a list of everything we need to do:

- Set the multisampling to 1x
- Store the resolution height in a variable.
- Set the UI Scale to a hard-coded known value.
- Multiply every dimension with `768/resoheight` and `1/uiscale`.

Let's start with the first point, setting the multisampling to 1x. This is simple, just call `SetMultisampleFormat(1)`, though this must be done after the event `VARIABLES_LOADED` has fired. Next up is storing the resolution height in a variable. The function `GetScreenResolutions()` returns all current selectable resolutions for WoW and `GetCurrentResolution()` returns the id of the currently selected resolution in the list. If we make a temporary table using the first function and look up in it using the second we will get the current resolution: `({GetScreenResolutions()})[GetCurrentResolution()]`. This is not the height of the resolution, but contains both width and height in a widthxheight string format, so let's extract the height using a little string match. The snippet so far:

```
local reso = ({GetScreenResolutions()})[GetCurrentResolution()]
local resowidth, resoheight = string.match(reso, "(%d+)x(%d+)")

local holder = CreateFrame("Frame")
holder:RegisterEvent("VARIABLES_LOADED")
holder:SetScript("OnEvent", function(holder, event)
    SetMultisampleFormat(1)
end)
```

Setting the UI Scale to a known value is very easy, first put the UI scale in a variable and then call `SetCVar("uiScale", uiscale)` and to make sure this UI Scale is used call `SetCVar("useUiScale", 1)`. We call this again after `VARIABLES_LOADED` to make sure it is not overwritten again when the saved CVars are loaded by WoW.

Now we must multiply every dimension with `768/resoheight` and `1/uiscale`. Two scaling factors combined means that they should be multiplied resulting in the factor `768/resoheight/uiscale`. From now on I call this factor *px* because it is the factor of one *virtual pixel* to one real pixel on your screen. The snippet so far:

```
local reso = ({GetScreenResolutions()})[GetCurrentResolution()]
local resowidth, resoheight = string.match(reso, "(%d+)x(%d+)")
local uiscale = .64
local px = 768/resoheight/uiscale

local holder = CreateFrame("Frame")
holder:RegisterEvent("VARIABLES_LOADED")
holder:SetScript("OnEvent", function(holder, event)
    SetMultisampleFormat(1)
    SetCVar("uiScale", uiscale)
    SetCVar("useUiScale", 1)
end)
```

The scale function

Now we have gathered all information we need in Lua variables, now it is time to use those variables. We are going to make what I call the *scale* function. The scale function takes one parameter, numpixels, and returns the amount of pixels you should supply to the WoW API in order to get the amount of physical pixels you wanted in numpixels. Before numpixels is multiplied by px it is rounded. The function goes like this:

```
local function scale(numpixels)
    return px * floor(numpixels + .5)
end
```

Now we have to do one last thing, implementing this scale function in the `SetSize` and `SetPoint` frame methods. I did this by creating two new global frame methods: `Size` and `Point`. Every object type in the WoW API has it's own method set saved in a Lua *metatable*. Simply adding our own function to that table is enough to add that function to every object that uses that type.

The size and point functions

The size function will be very easy, it is an exact pixelperfect copy of `SetSize`, except with one extra feature: `frame:Size(50)` equals `frame:Size(50, 50)`.

```
local function size(frame, width, height)
    if not height then
        height = width
    end
    frame:SetSize(scale(width), scale(height))
end
```

The Point function will be a little bit harder since it has a variable argument count, and all that we want to scale are the point offsets, so for every argument we scan whether it is a number, if it is, we scale it.

```
local function point(obj, arg1, arg2, arg3, arg4, arg5)
    -- anyone has a more elegant way for this?
    if type(arg1)=="number" then arg1 = scale(arg1) end
```

```
    if type(arg2)=="number" then arg2 = scale(arg2) end
    if type(arg3)=="number" then arg3 = scale(arg3) end
    if type(arg4)=="number" then arg4 = scale(arg4) end
    if type(arg5)=="number" then arg5 = scale(arg5) end
    obj:SetPoint(arg1, arg2, arg3, arg4, arg5)
end
```

And now we have our `Size` and `Point` functions, it is time to inject them. We iterate over every frame, get it's object type, if this object type hasn't already been injected, we inject the functions. We inject FontString and Texture objects as exceptions.

```
local function inject(obj)
    local mt = getmetatable(obj).__index
    mt.Size = size
    mt.Point = point
end

local handled = {["Frame"] = true}
local object = CreateFrame("Frame")
inject(object)
inject(object:CreateTexture())
inject(object:CreateFontString())

object = EnumerateFrames()
while object do
    if not handled[object:GetObjectType()] then
        inject(object)
        handled[object:GetObjectType()] = true
    end

    object = EnumerateFrames(object)
end
```

And we're done! Now we have a few rules that we must obey, but in return we can truly design our UI's pixel by pixel.

The endresult

The rules we must use to get pixelperfection on our frames are the following:

- Every pixelperfect frame must be parented to a parent tree with UIParent on top.
- No pixelperfect frame must use scale and must not be parented to any frame using scale except UIParent.
- In order to size and anchor your frames you must use the methods frame:Size() and frame:Point().

And the endresult snippet is this:

```
local reso = ({GetScreenResolutions()})[GetCurrentResolution()]
local resowidth, resoheight = string.match(reso, "(%d+)x(%d+)")
local uiscale = .64
local px = 768/resoheight/uiscale

local holder = CreateFrame("Frame")
```

```
holder:RegisterEvent("VARIABLES_LOADED")
holder:SetScript("OnEvent", function(holder, event)
    SetMultisampleFormat(1)
    SetCVar("uiScale", uiscale)
    SetCVar("useUiScale", 1)
end)

local function scale(numpixels)
    return px * floor(numpixels + .5)
end

local function size(frame, width, height)
    if not height then
        height = width
    end
    frame:SetSize(scale(width), scale(height))
end

local function point(obj, arg1, arg2, arg3, arg4, arg5)
    -- anyone has a more elegant way for this?
    if type(arg1)=="number" then arg1 = scale(arg1) end
    if type(arg2)=="number" then arg2 = scale(arg2) end
    if type(arg3)=="number" then arg3 = scale(arg3) end
    if type(arg4)=="number" then arg4 = scale(arg4) end
    if type(arg5)=="number" then arg5 = scale(arg5) end
    obj:SetPoint(arg1, arg2, arg3, arg4, arg5)
end
```

Notes

These are the most common issues when it comes down to pixelperfection, but this is not an all-including guide. There are always exceptions you will still run in to. There are a few more exceptions that I did not cover in this article, but I will give you a hint: Font size, backdrop border and tile sizes and other sizes in pixels. These values must be scaled too using the scale function. My suggestion is that you create a global alias for the scale function and that you use that in those cases.

Last updated: 21-09-2011

[world of warcraft] [wow] [tukui] [lua] [programming] [addon]