The Link to the Online and Constantly Updated Version of this Document HERE

HOW TO

The Asset is called colorKit because "colorKit" is the namespace that contains everything

So… If you want to use it simply type "using colorKit;" at the very top of your C# script

"colorKit" has multiple classes (listed below)

So… there are 2 ways to use the functions in those classes

1. type "className.functionName(parameters)"
    a. this works with classes that don't extend monobehaviour (static classes)
    b. but you must know the name of the class where the function you want to use is located
    c. and it requires the that you type "using colorKit;" at the very top of the C# script that uses it
2. create an instance of color with any name (the data of this instance isn't used), then type "anyName.function(parameters)" [more on this in the reasoning section]
    a. so you don't need to know the name class that contains every function
    b. and you don't need to type "using colorKit;" (unless you are using one of colorKit's enumerables as parameters)
    c. but this only works in classes that extend monobehaviour
    d. and it's weird [more on this in the reasoning section]

NOTE: this API also includes another API called "lerpKit" and depending on what functions you use you might have to also include "using lerpKit;" in your script

The User Manual for "Lerp Kit" can be found can be found here

---

DEFINITIONS

Keep in mind that…

- **Color Formats** define the formatting each component of the color has.
- **Data Types** define the data type each component of the color is in.
- **Color Space** defines what color space this particular color is in.

Color Spaces

- Additive
    - Red Green Blue (RGB)
- Subtractive
    - Red Yellow Blue (RYB)
    - Cyan Magenta Yellow Key (CMYK)

Color Formats (as Labeled in Code)

NOTE: a color in ANY Color Space can be represented with ANY Color Format

- "255"
  - Value Range: (0 -> 255)
  - White in RGB Color Space with "255" format: (255, 255, 255)
- "Float"
  - Value Range: (0 -> 1)
  - White in RGB Color Space with "Float" format: (1, 1, 1)
- "Hex"
  - Value Range: (00 -> FF)
  - White in RGB Color Space with "Hex" format: (FF, FF, FF)

Data Types Used To Store Colors

NOTE: a color in ANY color space can be represented with ANY Color Format with ANY Data Type

- 3 Component Colors (RGB, RYB)
  - Color
  - Vector3
  - Float[] (named array)
- 4 Component Colors (CMYK)
  - Vector 4
  - Float[] (named array)

MIXING METHODS

Space Averaging

- SOURCE: Myself
- MAIN IDEA: Linear Interpolation between 2 points in 3D Space or 4D Space
- PROCESS:
  - you start with a base color (the first color) and you add other colors to it
  - LOOP what is below until you run out of colors
    - mix 2 colors assuming they are both in the same quantities
    - get a range of colors (In other words... a line of colors from color A to color B)
    - use some form of linear interpolation to grab the color in between A and B given the ratio of A to B
- NOTES: Given that you are using the distance between 2 points in space…
  - Any Color mixed with this process that has 4 components will use Vector4.Distance which produces really strange results

Color Averaging [originally designed to mix in CMYK]

- SOURCE:
  - https://github.com/AndreasSoiron/Color_mixer/blob/master/color_mixer.js
    - http://www.andreassoiron.com/demos/color_mixer
  - http://jsbin.com/afomim/1/edit?html,css,js,output
- MAIN IDEA:
  - Average Colors
- PROCESS
  - Add up all the Colors
  - Divide all the Components by the total number of colors

Color Component Averaging

- SOURCE: myself
- MAIN IDEA:
  - Average Color Components
- PROCESS:
  - Add up all the Colors' Components
  - Divide all the Component by the total number of times that particular component had a value greater than 0

Each As Percent Of Max [originally designed to mix in RYB]

- SOURCE:
  - https://github.com/camme/ryb-color-mixer
    - Copies mixRYB() function
- MAIN IDEA:
  - ???
- PROCESS:
  - sum up all the color vectors by component
  - find the largest component in the final color
  - every component is found like this...
    - new component = new component / max component * 255

Documentation of "colorKit" namespace

Enumerables

- **desiredMixtureType** { additive, subtractive }
  - not used yet (IGNORE)
- **colorSpace** { RGB, RYB, CMYK }
  - (Red, Green, Blue) | (Red, Yellow, Blue) | (Cyan, Magenta, Yellow, Key/Black)

- **mixingMethod** { spaceAveraging, colorAveraging, colorComponentAveraging, eachAsPercentOfMax }
    - for all the mixingMethods below
        - The colors passed MUST be in… 3 Component -OR- 4 Component Format
        - The colors passed SHOULD be in the same Color Space
        - The colors passed SHOULD be in “255” Format
        - The color returned in the array is ALWAYS in the same Color Space and Color Format as the colors passed to it
        - Opacities will NOT be mixed
    - for colorAveraging -or- colorComponentAveraging…
        - for 4D colors (like cmyk) there are 2 ways to mix… both weird in their own way… chose which to use by changing the public boolean "useVect4Dist" in the mixingMethods class… when it's true you use Vector4.Distance to mix… when its false you use something else… depending on the mixingMethod you choose

<u>Classes</u> and the <u>Functions</u> they contain

(all classes and functions below are public and static)

color<u>Format</u>Conversion class

- float[] _float_to_255(float[] colorFloat)
- string[] _float_to_hex(float[] colorFloat)
- float[] _255_to_float(float[] color255)
- string[] _255_to_hex(float[] color255)
- float[] _hex_to_float(string[] colorHex)
- float[] _hex_to_255(string[] colorHex)

color<u>Type</u>Conversion class

- 2 Component ???
    - float[] vector2_to_array(Vector2 vector2)
    - Vector2 array_to_vector2(float[] array) //the array must be of size 2
- 3 Component Color
    - float[] vector3_to_array(Vector3 vector3)
    - Color vector3_to_color(Vector3 vector3)
    - Vector3 array_to_vector3(float[] array) //the array must be of size 3
    - Color array_to_color(float[] array) //the array must be of size 3
    - Vector3 color_to_vector3(Color color)
    - float[] color_to_array(Color color)
- 4 Component Color
    - float[] vector4_to_array(Vector4 vector4)
    - Vector4 array_to_vector4(float[] array) //the array must be of size 4

rgb2ryb_ryb2rgb class

- float[] rgb255_to_ryb255(float[] rgb255)
- float[] ryb255_to_rgb255(float[] ryb255)

rgb2cmyk_cmyk2rgb class

- float[] rgb255_to_cmyk255(float[] rgb255)
- float[] cmyk255_to_rgb255(float[] cmyk255)

colorDistances class

- float distBetweenColors(Color color1, Color color2, colorSpace colorSpaceUsed)
- float distBetweenColors(float[] color1, float[] color2)

colorCompliments class

- Color complimentary(Color origColor, colorSpace csToUse)
- float[] complimentary(float[] color, int floatLimit)

colorLerping class

- Color colorLerp(Color start, Color end, float lerpValue, colorSpace csToUse)
- float[] colorLerp(float[] start, float[] end, float lerpValue)

colorLerpHelper class

- float calcGuideDistance(Color startColor, Color currColor, Color endColor, colorSpace CS, guideDistance GD)
- float calcLerpValue(Color startColor, Color currColor, Color endColor, float guideDistance, float guideTime, unitOfTime UOT_GD, updateLocation UL, colorSpace CS)
- float calcLerpValue(Color startColor, Color currColor, Color endColor, float lerpVelocity_DperF, colorSpace CS)

colorMixing class (cares about colorSpace)

- Color mixColors(Color[] colors, colorSpace csToUse, mixingMethod mm)
- Color mixColors(Color[] colors, float[] colorQuantities, colorSpace csToUse, mixingMethod mm)

mixingMethods class (doesn't care about colorSpace)

- float[] mixColors(mixingMethod mm, List<float[]> colors)
- float[] mixColors(mixingMethod mm, List<float[]> colors, float[] colorQuantities)

colorOtherOps class

- **print functions**
  - void print(Vector4 vect4)
  - void print(Vector4 vect4, string printLabel)

    - void print(Vector3 vect3)
    - void print(Vector3 vect3, string printLabel)
    - void print(Color color)
    - void print(Color color, string printLabel)
    - void print(Vector2 vect2)
    - void print(Vector2 vect2, string printLabel)
    - void print(float[] array)
    - void print(float[] array, string printLabel)
- **error correcting functions**
    - float[] nanCheck(float[] array)
    - float[] clamp(float[] array, float min, float max)

REASONING

The colorKit is not 1 single script because…

- it would be incredibly massive
- and the colorKit can be logically divided into classes

The colorKit is a collection of static classes and functions because…

- It doesn't make sense to have to address a script within a gameObject to use functions that don't require information from the instance
- all the functions calculate something but don't store any data
- it would allow use to easily convert the class/function into an extension class/function

The colorKit uses namespaces because...

- they can easily be included in a script using the "using" directive
- once included you simply type the className.functionName to use it
- because it allows functions and classes with the same name to co-exist in separate namespaces and consequently reduces the possibility of hard to find bugs

From the Beginning, I wanted these functions to be an extension of the Unity Color Class addressable as such "Color.mixColors()" by using the C# extension classes/functions feature.

Unfortunately, C# doesn't allow extension classes/functions inside of another namespace… and I didn't want to take the classes out of the namespace for the many reasons above.

If we simply removed these classes from the namespace but still kept them seperate as their own class then another problem would arise. This is because (1) all classes and functions within colorkit are static, (2) most classes/functions within colorKit use other classes/functions within colorKit, and (3) all static classes must derive from object and can therefore not extend monobehaviour.

Now, In order to make a particular class an extension class, it has to be static. If you want to call a function from the static extension class (from a different class), the other class would have to extend monobehavior.

So, since we want our classes to be extension classes of Unity's Color class, they would have to be static… but by being static, they can't call other classes that are already extension classes of Unity's Color class, because, they would have to extend from monobehavior… which once again is not possible within a static class.

So instead, I created a separate script and class called "colorExtensionFunctions" that uses the colorKit and calls its functions to form an extension class that contains all the 'references' to all the function in colorKit... all in one place... without all the clutter... or any of the complications.

Unfortunately, C# doesn't allow truly static extension methods (more on this here). So addressing the functions in the "colorExtensionFunctions" script (that simply calls the script in the colorKit namespace) wouldn't work as initially planned (as such "Color.mixColors()").

So instead, we create a dummy instance of the Color class. We don't use any of that instances data anywhere… However, with said instance we can now call things with ease by simply typing "instanceName.functionName()" regardless of what class in colorKit contains that function.