Readme – ImageLoader 1.1.8

Load/Download image(s) with URL and local path using the same API.
Supports batch loading images, queued batch loading images, and the ability to specify the number of loaders to use.
Provides powerful cache management feature for caching your images, and handles file naming, folders, time to keep and delete images, etc.

**** Note that this asset supports load images from public accessible URL or local path, and cache on local storage with Read/Write permission. It is not a native plugin for Load/Save files from the Android Gallery or iOS Photos app.***

Features

- ImageLoader: load single image, support multiple instances for loading images.
- ImageQueuedLoader: load multiple images with a single API, supports limiting the number of loaders, split loading process to avoid blocking the main thread when loading a large number of images.
- ImageBatchLoader: wrapped the ImageLoader & ImageQueuedLoader for easily loading multiple images. Load multiple images with a single API.
- Load image(s) from local storage(application paths), and load image(s) with Url(s) from web using the same API.
- Detect actual image Extension Name and MIME Type.
- Flexible sequential file naming system (for batch download)
- Easy, powerful images cache management, with rich settings:
  - Retry option
  - Timeout option
  - Cache mode: NoCache, UseCached, Replace
  - Cache file limit per folder
  - Cache as per URL, or Cache using specific Filename/Sequence
  - Particular cache(Save/Load) directory and folder
  - Sequential file naming formats
  - Min. keep file time & file expire(max.) time
- Example scenes included.

Bonus Features

EasyIO Plugin: supports saving & loading image, text, class object, and file byte array on different platforms. Provides the ability for your app to write files in IndexedDB, so the cache management system can work correctly on WebGL.

The ImageLoader scripts are placed in our ImageBox namespace (IMBX), you can start with the IMBX keyword every time, or add the namespace at the beginning of your script:

```
using IMBX;
```

Now, you can start integrating proper loader method(s) in your code, let's move on to the following sections for the APIs we have.

(1) Image Loader

Load/Download single image from Web or local storage(in-app, Application paths). This is the basic loader unit used in our Batch loaders, i.e. **ImageBatchLoader**, **ImageQueuedLoader**. However, you can use it independently if you want.

- **Create a new ImageLoader:**
  ImageLoader loader = **ImageLoader.Create**(uint:maxCacheFilePerFolder,
  FilePathName.AppPath:cacheDirectory, string:gameObjectName);

- **Start to load/download the image:**
  API 1: loader.**Load**(uint:index, string:url, LoaderManagement:lmgt,
  Action<Texture2D, uint>:onComplete);
  or
  API 2: loader.**Load**(uint:index, string:url, Action<Texture2D, uint>:onComplete,
  uint:retry, float:timeOut);
  or
  API 3: loader.**Load**(uint:index, string:url, string:fileName, string:folderName,
  CacheMode:cacheMode, Action<Texture2D, uint>:onComplete,
  uint:retry, float:timeOut);

- **Cancel download:**
  loader.**Cancel**();

- **Delete all files in the cache folder:**
  loader.**ClearStorageCache**();

- **Cancel a download by image URL:**
  ImageLoader.**CancelLoading**(string:imageUrl);
  (This is a global method, able to cancel any loading from all types of our loaders)

- **Cancel all downloads:**
  ImageLoader.**CancelAllLoading**();
  (This is a global method, able to cancel all loadings from all types of our loaders)

- **File Extension & MIME Type**: Image Loader supports getting actual file extension name and MIME type, even the file extension name is unknown or incorrectly set. You can check the following variables when loading completed in the OnComplete callback.
  string: DetectedFileExtension
  string: DetectedFileMime

(2) Image Batch Loader

Load/Download multiple images from Web or local storage(in-app, Application paths), using Load and Load_Queue methods of ImageBatchLoader:

- **Create a new ImageBatchLoader:**
  ImageBatchLoader loader = **new ImageBatchLoader**(uint:maxCacheFilePerFolder,
  FilePathName.AppPath: cacheDirectory);

- **Start to load/download the images:**

  Non Queued Loader methods (For FPS don't care, static loading screen, etc)**:**
  API 1: loader.**Load**(List<string>:imageUrls, LoaderManagement:lmgt,
  Action<Results>:onComplete, Action<Result>:onProgress);
  or
  API 2: loader.**Load**(List<string>:imageUrls, Action<Results>:onComplete,
  Action<Result>:onProgress, uint:retry, float:timeOut);
  or
  API 3: loader.**Load**(List<string>:imageUrls, string:filenamePrefix, string:folderName,
  ImageLoader.CacheMode:cacheMode, Action<Results>:onComplete,
  Action<Result>:onProgress, uint:retry, float:timeOut);

  Queued Loader methods (Split tasks to a specific num of loaders, until all done)**:**
  API 1: loader.**Load_Queue**(uint:maxLoaderNum, List<string>:imageUrls,
  LoaderManagement:lmgt, Action<Results>:onComplete,
  Action<Result>:onProgress);
  or
  API 2: loader.**Load_Queue**(uint:maxLoaderNum, List<string>:imageUrls,
  Action<Results>:onComplete, Action<Result>:onProgress,
  uint:retry, float:timeOut);
  or
  API 3: loader.**Load_Queue**(uint:maxLoaderNum, List<string>:imageUrls,
  string:filenamePrefix, string:folderName,
  ImageLoader.CacheMode:cacheMode, Action<Results>:onComplete,
  Action<Result>:onProgress, uint:retry, float:timeOut);

- **Delete all files in the cache folder:**
  loader.**ClearStorageCache**();

- **Extra Callbacks** (for manually register)**:**
  OnImageLoaded: to be invoked when an image is loaded.
  loader.**m_OnImageLoaded** = (result) => { /* your code here... */ };

  OnAllImagesLoaded: to be invoked when the last task in the queue is finished.
  loader.**m_OnAllImagesLoaded** = () => { /* your code here... */ };

- **File Extension** and **MIME Type** are included in the **Result/Results** object that returns with the OnProgress and OnComplete callbacks.
  e.g. The **Result** object returns in the OnProgress callback:
  result.m_DetectedFileMime;
  result.m_DetectedFileExtension;

  The **Results** object returns in the OnComplete callback:
  results.GetExtensionName(int:index);
  results.GetMimeType(int:index);

(3) Image Queued Loader

Load/Download multiple images from Web or local storage(in-app, Application paths), in a queued manner. (* T*he Load_Queue methods of ImageBatchLoader wrapped the ImageQueuedLoader to load images*)

- **Create a new ImageQueuedLoader:**
  ImageQueuedLoader loader = **ImageQueuedLoader.Create**(uint:maxLoaderNum,
  uint:maxQueueSize);

- **Load methods are the same as the Load_Queue methods of ImageBatchLoader.**

- **Add new image URLs/paths to load:**
  loader.**Add**(string:imageUrl, string:overrideFilename);
  or
  loader.**Add**(List<string>:imageUrls, List<string>:overrideFilenames);

- **Cancel all the current loading tasks:**
  (Optional not to cancel the specified image URLs)
  loader.**CancelAllLoading**(List<string>:exceptURLs);

- **Cancel all the pending loading tasks:**
  (Optional not to cancel the specified image URLs)
  loader.**CancelAllPending**(List<string>:exceptURLs);

- **Delete all files in the cache folder:**
  loader.**ClearStorageCache**();

- **Extra Callbacks** (for manually register)**:**
  OnImageLoaded: to be invoked when an image is loaded.
  loader.**m_OnImageLoaded** = (result) => { /* your code here... */ };

  OnAllImagesLoaded: to be invoked when the last task in the queue is finished.
  loader.**m_OnAllImagesLoaded** = () => { **/* your code here... */** };

(4) **Memory Cache** (using ImageBatchLoader)

ImageLoader supports caching the loaded images in the memory, or called In-Memory cache, is one of the cache features of ImageBatchLoader, another one is the Storage Cache which you can find in section **(5)**. Each ImageBatchLoader instance can have its own memory cache.

Caching in the memory is a good idea if your app needs to access some of the images frequently, and needs to reduce the computing resources to load and create new textures each time. For example, when show images in the scroll view, less loading and texture creation can remain the scrolling smooth.

To use the memory cache feature, just call the **EnableMemoryCache** method like below:

1. Create a Batch Loader instance:

```
ImageBatchLoader loader = new ImageBatchLoader(...);
```

2. Call the Memory Cache setup method:

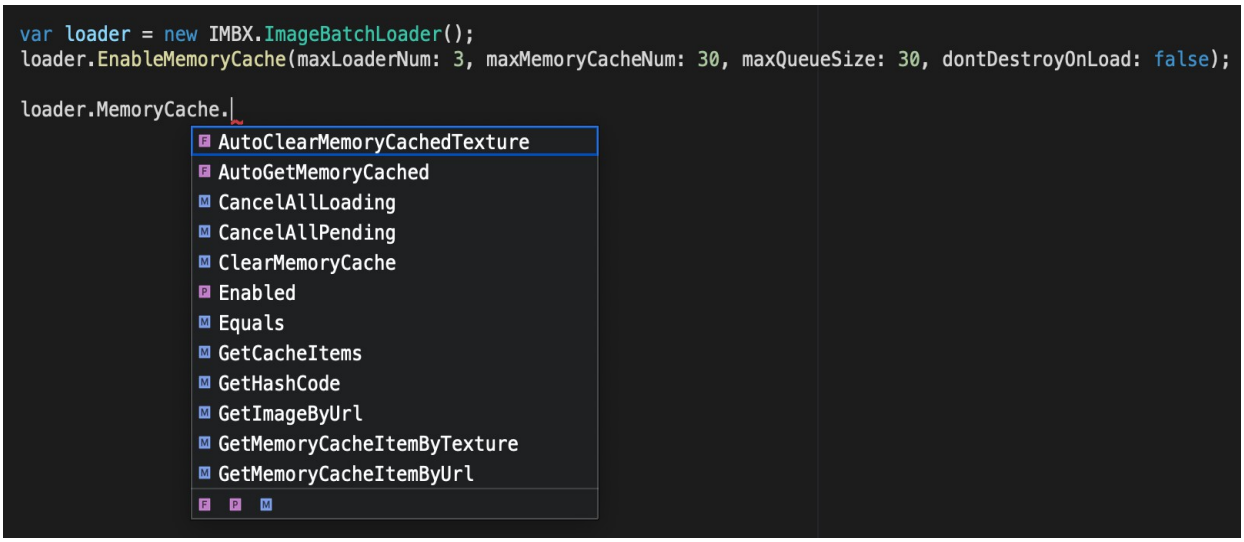
```
loader.EnableMemoryCache(uint:maxLoaderNum, uint:maxMemoryCacheNum,
    uint:maxQueueSize, bool:dontDestroyOnLoad);
```

3. Now you can call the **Load** and **Load_Queue** methods as usual. Or use the dedicated Get image method as shown below (suggested):

GetImageByUrl / GetMemoryCacheItemByUrl : the dedicated methods for getting images when Memory Cache is enabled. Use it to get the existing image(s) in the cache or request to load the image by its path/URL. Calling this method multiple times with different image paths/URLs will queue the tasks up in the list, until all are done.

```
CacheItem image = loader.MemoryCache.GetMemoryCacheItemByUrl(
    string:imageUrl, bool:loadIfNotFound, bool:isLock);
```

* Access memory cache functions through the **MemoryCache** object of ImageBatchLoader:



Memory Cache methods and variables:

- **In-Memory Cache object:**
  loader.**MemoryCache**.MemberFunctionOrVariable;
  (Contains all the methods and variables of the memory cache)

- **Should the loader search the memory cache before loading:**
  bool : AutoGetMemoryCached (default = true)
  (If true, automatically gets the cached items in this loader's memory cache when calling the Load/Load_Queue methods)

- **Should the loader clear the memory cache before loading:**
  bool : AutoClearMemoryCachedTexture (default = true)
  (If true, auto clear the texture in the item when it is being removed from Memory Cache by our internal codes, e.g. when exceeding the cache limit)

- **Total Cached Images:**
  int : MemoryCachedCount
  (Total num of Results/images currently cached in this ImageBatchLoader)

- **Modify the MemoryCache size:**
  loader.MemoryCache.**SetMaxMemoryCacheNum**(uint:maxMemoryCacheNum);

- **Lock specific image URLs to avoid being auto cleared:**
  loader.MemoryCache.**LockMemoryCacheItems**(List<string>:lockImageUrls,
  bool:replace);

- **Check if an image is in this memory cache, by path/URL:**
  bool: loader.MemoryCache.**IsMemoryCacheContainsImage**(string:imageUrl);

- **Check if a texture is in this memory cache, by object reference:**
  bool : loader.MemoryCache.**IsMemoryCacheContainsTexture**(Texture:texture);

- **Get an image(CacheItem) by texture reference:**
  CacheItem : loader.MemoryCache.**GetMemoryCacheItemByTexture**(Texture:texture);

- **Get an image(CacheItem) by path/URL, optional to load the image if not found:**
  CacheItem : loader.MemoryCache.**GetMemoryCacheItemByUrl**(string:imageUrl,
  bool:loadIfNotFound, bool:isLock);

- **Move an image(CacheItem) to the end of the cache list, like new loaded:**
  loader.MemoryCache.**MoveMemoryCacheItemToLast**(CacheItem:item);

- **Cancel all the current loading tasks:**
  (Optional not to cancel the image URLs that are marked as locked)
  loader.MemoryCache.**CancelAllLoading**(bool:exceptLocked);

- **Cancel all the pending loading tasks:**
  (Optional not to cancel the image URLs that are marked as locked)
  loader.MemoryCache.**CancelAllPending**(bool:exceptLocked);

- **Remove an image(CacheItem) by path/URL:**
  loader.MemoryCache.**RemoveMemoryCacheItemByUrl**(string:imageUrl,
  bool:canRemoveLockedItem);

- **Remove an image(CacheItem) by texture reference:**
  loader.MemoryCache.**RemoveMemoryCacheItemByTexture**(Texture:texture,
  bool:canRemoveLockedItem);

- **Remove an image from both Storage Cache folder and Memory Cache list:**
  loader.MemoryCache.**RemoveFromCaches**(string:imageUrl,
  bool:canRemoveLockedItem);

- **Remove all items in the memory cache, optional not to remove locked items:**
  loader.MemoryCache.**ClearMemoryCache**(bool:exceptLocked);

(5) Storage Cache & Loader Management

ImageLoader provides highly customizable settings for loading images and handles cache management per folder to cache images in the device storage. All types of our loaders support Storage Cache, they are ImageLoader, ImageBatchLoader, and ImageQueuedLoader.
Simply set the appropriate parameters in the **LMGT** object (**LoaderManagement**) before loading images. Check out the below example and the LMGT parameters.

Example:

loader.**LMGT**.CacheMode = ImageLoader.CacheMode.UseCached;
(**loader** can be a ImageLoader, ImageBatchLoader or ImageQueuedLoader instance)*

- **Cache Mode**
  ImageLoader.CacheMode : CacheMode
  (The behavior for handling Load and Cache files. ***NoCache***: do not auto save the image; ***UseCached***: use the locally cached file if exist; ***Replace***: download and replace the locally cached file if exist.)

- **Cache Directory**
  string : CacheDirectory (Getter)
  (The root directory for loading and storing/caching image files. Supports the application paths only, e.g. Application.persistentDataPath.)

  FilePathName.AppPath : CacheDirectoryEnum
  (The enum that determines which application path to load and save(cache) file to.)

- **Cache Folder Path**
  string : CacheFolderPath (Getter)
  (The cache folder path that combined by the root CacheDirectory and FolderName.)

- **Sub-Folder**
  string : FolderName
  (The sub-folder under cache directory for loading and storing(caching) files to.)

- **Cache as per URL**
  bool : CacheAsPerUrl
  (If 'true', for URL start with 'http', the loader will cache the image as per URL address.)

- **Batch FileName Prefix (for batch loader)**
  string : FileNamePrefix
  (The filename prefix for storing(caching) the image files. The final filename will be combined by this prefix, separator, and the index together.)

- **File Extension**
  string : FileExtension
  (Suggest use .jpg or .png only)

- **Number of Digits for the Index follow the Filename Prefix (for batch loader)**
  uint : FileIndexFormatDigitsCount
  (e.g. For digits count = 5, and index = 12, the result index string becomes 00012)

- **File Name Starting Index (for batch loader)**
  uint : FileNameStartingIndex
  (Set this value to set an offset for the filename index. The default starting index is 0.)

- **Separator text between the File Name and Index (for batch loader)**
  string : FileNameAndIndexSeparator
  (Please use filename friendly characters only)

- **Max. Files Per Folder**
  uint : MaxCacheFilePerFolder
  (The maximum number of files to cache per folder. Auto deletes the older files if exceed this limit. Zero means no limit.)

- **Min. Keep File Time**
  uint : MinTimeForKeepingFiles
  (e.g. m_MinTimeForKeepingFiles = 86400, 86400 seconds = 1 day
  For images in the target folder that modified within 1 day will not be auto deleted.)

- **File Expire Time**
  uint : MaxTimeForKeepingFiles
  (e.g. m_MaxTimeForKeepingFiles = 864000, 864000 seconds = 10 days
  Delete images modified more than 10 days ago.)

- **Loading Retry**
  uint : LoadingRetry
  (Number of times to retry when a loading failed. Retry per second until the retry value is Zero.)

- **Loading Timeout**
  float : LoadingTimeOut
  (The time duration for waiting for a loading/downloading, stop and kill the loader if time exceeds.)

- **Allow/Avoid Loading the same URL (at the same time)**
  bool : AllowDuplicateDownload
  (If true, allows downloading the same URL using multiple loaders, else it will not start a new download if that URL is being downloaded.)

(6) Extra

All download tasks through any loader methods, no matter you are using the ImageLoader directly, using the ImageBatchLoader or the ImageQueuedLoader, are added to a static loader list in the ImageLoader class, until the download finish or is canceled. This allows us to monitor the tasks during their lifecycle.
Here are some static methods which give you flexible control of the downloading tasks. These methods are optional to use, depending on your app design.

- Total existing loaders/loading processes:
  int : LoadingCount;
  e.g. ImageLoader.LoadingCount;

- Check if a given image URL is being loaded in any existing ImageLoader:
  bool IsUrlLoading(string imageUrl);
  e.g. ImageLoader.IsUrlLoading(imageUrl);

- Cancel and Destroy existing ImageLoader(s) by image URL:
  void CancelLoading(string imageUrl);
  e.g. ImageLoader.CancelLoading(imageUrl);

- Cancel and Destroy all the existing ImageLoaders:
  void CancelAllLoading();
  e.g. ImageLoader.CancelAllLoading();

(7) Manage Cache Files (manually)

ImageLoader only automatically performs cache management(save & delete files) in the specified folder during a Load method is executed with cache mode **UseCached** or **Replace**. It is done by executing the global method **ManageCacheFiles** in the ImageLoader class, inside the Load method. You can also call this method manually as needed (e.g. at the app start/quit).

Example:

```
ImageLoader.ManageCachedFiles(fullFolderPath, maxFilePerFolder,
        minTimeForKeepingFiles, maxTimeForKeepingFiles, fileExtension);
```

THANK YOU

Thank you for using this package!
For any question and bug report please contact us at swan.ob2@gmail.com.
Remember to rate this asset on the Asset Store. Your review is always appreciated, and very important to the development of this asset!

Review And Rating

Visit our asset page to find out more!
https://www.swanob2.com/assets

SWAN DEV