

Game Save Sample

*This sample is compatible with the Microsoft Game Development Kit (June 2020)*

# Description

This sample demonstrates the use of the Game Core **XGameSave** APIs to save and load game save data. It shows how to use both the normal “full sync” method for handling cloud synchronization – where all of the user’s game save containers are synchronized when the user plays the title on another console – and how to use “sync-on-demand” mode, which provides more fine-grain control over when and how containers are synced. This is typically intended for use with larger containers which might take some time to synchronize.

The sample also demonstrates a variety of other techniques related to the new Game Core API surface area. Most of the files related to this may be found in the \Helpers\ folder, and they demonstrate techniques such as wrapping asynchronous operations in C++ classes, RAII wrappers, single-player user management, controller binding and more.

The sample takes the form of a simple single-player word puzzle game. Use the gamepad to select spots on the board and choose which letters to place in those locations. Adjacent letters are chained together, and if you create a valid English word, they will calculate a word score based on letter frequency.

The following game save scenarios are covered by this sample:

**Using full-sync or sync-on-demand mode**

When the sample is launched you have the option of choosing whether the sample uses the full-sync API

(which syncs all game save data between console and the title storage service) or the sync-on-demand API (which syncs game save data only as you need it). A typical real-world title would hard-code this decision, based on the needs of the game.

Very few titles currently shipping today use the sync-on-demand mode; those that do tend to use very large containers **and** large numbers of containers, such that synchronizing all of the game data could be inconvenient for the user to wait to complete.

NOTE: If you’ve already performed a full sync and you want to experiment with sync-on-demand, you should either sign in with a different user or clear the local cache of game save data.

To clear the local cache for the Xbox One, you would run the “xbstorage.exe reset /force” command from the XDK command prompt.

To clear the local cache for a Windows 10 PC, you would run the “gamesaveutil.exe reset” command from an administrator command prompt.

Note: **gamesaveutil.exe** can be found in the following directory once you’ve installed the Windows May 2019 SDK:  
%ProgramFiles(x86)%\Windows Kits\10\Extension SDKs\XboxLive\1.0\Bin\x64

**Load, Save, and Delete game save data**

Use the menu options to load game boards, save them, and delete them. You can save up to 9 different boards.

**List containers and blobs**

Use the menu options to enumerate containers and blobs. The output is displayed in the scrollable debug output region of the game screen.

**View last modified date and remaining quota**

This info is displayed just below the title on the game screen.

**Auto save on user sign out**

If the current game board has not yet been saved, it will be automatically saved if and when the user signs out.

**Auto save on suspend**

In reaction to a suspending event, if the current game board has not yet been saved, it will be automatically saved.

# Building the sample

If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

If using Project Scarlett, set the active solution platform to Gaming.Xbox.Scarlett.x64.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the sample

**Xbox Live Sandbox Requirements**

* Xbox One devkit: set the console’s sandbox to XDKS.1

**Launch Menu**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  | | --- | --- | --- | | Action | Gamepad | Keyboard | | Select between “full sync” mode and “sync-on-demand” mode | Left Stick or D-Pad | Arrow keys | | Select menu item | A button | Enter | |

**Game Board**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  | | --- | --- | --- | | Action | Gamepad | Keyboard | | Move cursor | LS or D-Pad | Arrow keys | | Select menu item | A button | Enter | | Select game save slot | LB/RB button | 1 – 9 keys | | Change letter tile under cursor | RS Left/Right | A – Z keys | | Clear letter under cursor | X button | Delete or Space | | Scroll debug output | RS Up/Down | Page Up/Dn  and  Home/End | |

# Game Menu notes

* **Read Board**Most of the work here is performed by **GameSaveManager::ReadBlocking.** This uses the **XGameSaveCreateContainer** and **XGameSaveReadBlobData** APIs to load the game board for the current game save slot.
* **Save Board**  
  Most of the work here starts in **GameSaveManager::SaveBlocking**. Uses the **XGameSaveCreateContainer, XGameSaveCreateUpdate, XGameSaveSubmitBlobWrite, XGameSaveSubmitBlobDelete,** and the **XGameSaveSubmitUpdate** API, to save the game board for the current game save slot.
* **Reset Board**  
  Clears the board of all letters. If the board has been previously saved, this will mark the board “dirty”. If the board has not yet been saved, the board will not be marked “dirty”.
* **Delete Board**The work is performed inside of **GameSaveManager::DeleteBlocking.** Uses the **XGameSaveDeleteContainer** API to delete the game board for the current game save slot.
* **Delete Board Blob**  
  Work is mostly handled by **GameSaveManager::DeleteBlobsBlocking**. Uses the **XGameSaveCreateContainer, XGameSaveCreateUpdate, XGameSaveSubmitBlobDelete,** and the **XGameSaveSubmitUpdate** API to delete just the blobs for the current game board, leaving an empty container.
* **List Containers**  
  The work here is performed in **GameSaveManager::EnumerateContainersBlocking.** Uses the **XGameSaveEnumerateContainerInfo** API to enumerate all containers and list them in the debug output area of the game screen.
* **List Containers & Blobs**The work here is performed in **GameSaveManager::EnumerateContainersBlocking.** Uses the **XGameSaveEnumerateContainerInfo, XGameSaveCreateContainer** and **XGameSaveEnumerateBlobInfo** APIs to enumerate all containers and blobs and list them in the debug output area of the game screen.

Game Play notes

**Game Play**

The game is played on a 5 x 5 grid. You can place letters anywhere on the grid. Consecutive letters that form a recognized English word, either across or down the board, will score points based on the total of the point values for each letter in the word. The objective is to maximize your score. You have a limited number of each letter to place on each board. The remaining count is tracked just above the game board.

**Game Board Loading**

For demonstration purposes, game boards **do not load automatically** when the game board screen appears, or when you switch to a new game save slot. This gives you full control of load and save operations while on a particular game save slot.

**Changed Game Boards & Auto-Save**

When a letter has been changed on the game board, or when you use the Reset menu command, the game board will be marked “dirty” (indicated by an asterisk after the board name at the top of the screen). Dirty game boards will be auto-saved under the following conditions:

* Switching to a different game board (gamepad LB/RB)
* User signout
* Game suspending

# Implementation notes

The **GameSaveManager** class manages game save operations for the game. The **InitializeForUser()** method sets up a Connected Storage save context for a player. There are also methods for loading, saving, enumerating, and deleting save data. See the comments in the header file for usage notes on each method in the class.

While this sample has some user management and gamepad management features, *it is not intended to demonstrate those features*. This sample is for understanding the usage of game saves in different circumstances.

There are 2 types of game data structures used by the game: an index and a game board. The templatized **GameSave** class provides methods for use by the **GameSaveManager** for loading and saving data generically for any type of game data. The index, defined by the **GameBoardIndex** struct in GameSaveManager.h, is used primarily to keep track of the last save slot used by the player (the “active board”). The game board data is represented by the **GameBoard** struct in GameBoard.h.

# Additional Sample Code

This sample ships with a series of additional helper classes which you may use in your own code to assist in working with the Game Core APIs. Most of these files can be found in the \Helpers\ folder.

|  |  |  |
| --- | --- | --- |
| Folder | Filename | Description |
| \Helpers\ | HandleWrapperBase.h  XGameSaveHandleWrappers.h XTaskQueueHandleWrapper.h XTaskQueueHandleWrapper.cpp XUserHandleWrapper.h | RAII class wrappers around the Game Core handle types defined by key APIs used by the sample. |
| \Helpers | User.h  User.cpp | A user object which can be held onto by your title and queried. At its heart is an XUserHandle. It is the object-type returned by the single-signed in user manager. |
| \Helpers | UserManager.h  UserManager.cpp | A single-signed in user manager which tracks the one user playing the game, and allows a different user to be selected. |
| \Helpers | UTF8Helper.h  UTF8Helper.cpp | Utf-8 text conversion functions. |
| \Helpers | Buffer.h  Buffer.cpp | Blob-handling classes; these allocate memory and track the start location and size of the allocated buffer, releasing the memory when the object is destroyed. They also support move semantics. |
| \Helpers | StateMachine.h | A lock-free implementation of a state machine. (A version is also provided which is both lock-free and allows a consumer to wait until the state machine changes to a specific state). |
| \Helpers | AsyncOp.h  AsyncOp.cpp | A base class which you can extend from to track the progress of an asynchronous call in the API. An AsyncTask implementation is also provided which allows you to perform operations on an async task queue. |
| \Helpers | AsyncAction.h | A template based class which performs a std::function or lambda on an async task queue, and allows you to obtain a produced result (if any). This works similarly to ppltasks. |
| \Helpers | ScopedLockWrappers.h | Scoped wrappers around a Slim Reader/Writer lock, including support for try-enter and for only providing access to payload data within the scope of a lock. |
| \Helpers | TaskQueue.h TaskQueue.cpp | On initialization, creates one worker task dispatch queue and one work-completion queue per core, and allows you to query for that core’s task queue.  **Note:** This code shows how to properly create a thread that is *suspended* on startup, and then affinitize it to a specific core before resuming that thread. This allows it to be spun up without disturbing work on existing cores, or core hopping. |
| \Common\ | InputDeviceManager.h  InputDeviceManager.cpp  ScopedGameInputDeviceInfo.h GamePad.h GamePad.cpp | Gamepad manager which tracks game pad connection/disconnection events, and user-device association, as well as plumbing GameInput readings through to the traditional DirectXTK Gamepad implementation. |
| \ | Assets.h Assets.cpp SampleSpecificAssets.inl | The foundation for a simple streaming asset management system, which could also be used to simplify lost device handling on PC by centralizing asset management for the sample compared to previous versions. |

## Asynchronous Queue Usage

The sample creates a task queue which submits tasks and notifications on the default system thread pool. All asynchronous workloads created by the sample code are queued on this queue. Those tasks will be run the next available thread on the thread pool.

All user-device association, user events and GameInput-related callback work are queued to the **DEFAULT\_INPUT\_WORK\_AND\_CALLBACK\_CORE** defined in Common\InputDeviceManager.h. The intent is to make handling these events simpler by forcing them to all be serialized onto a single thread.

## Thread Safety

The user manager classes and gamepad-related classes use traditional heavyweight locking mechanisms for cross-thread synchronization, and should be thread-safe. However, they may not be reentrancy-safe.

# Known issues

* **GameInput** Controllers entering power-down idle state crash the GameInput library.

# Update history

**Initial Release: August 2019**

New version, mostly rewritten to utilize new task system, Game Core APIs, use synchronous (blocking) GameSave calls, new asset loader code, new user management work, remove all WinRT related code. This version also removes PC support, which will be added back into a future release.

# Privacy statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, remove ATG\_ENABLE\_TELEMETRY from the C/C++ / Preprocessor / Preprocessor Definitions list in the project’s settings.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).