# UniLWP.Droid

## Table of contents

# Introduction

UniLWP.Droid is a live wallpaper (LWP) implementation of Unity on Android. It is used in most live wallpapers made by FinGameWorks, including Metropolis and Vortex.

## Features

UniLWP.Droid uses Unity 2019.3+ as the base Unity version, which integrates the ‘Unity as a library’ feature into the package, meaning that you can do your own Android development aside from Unity development at the same time without the need to export the project again.
Features include:

- Support Unity 2019.3+
- Android Callbacks
    - Window Insets
    - Wallpaper Offsets
    - Screen AOD / Lock / Unlock State
    - Dark Mode Events
- Flexible Android customization
    - Bring your own preview activity
    - Bring your own wallpaper service
    - Bring your own wallpaper.xml
    - Bring your own UI (native or unity)
    - Extend native callbacks
    - Support Always On Display (Android 9 only)

You can download the sample app from Google Play. Notice that some features flags are set in the compile stage, so the sample app may lack certain features, but it should give you a pretty good impression of what it is capable of.

## Technical Structure

In order to better understand how UniLWP.Droid works, we have to first comprehend how does live wallpaper work.

### WallpaperService

In Android, live wallpaper is provided as a service: WallpaperService. The WallpaperService works as normal Android services, except that it exposes a method:
`public abstract WallpaperService.Engine onCreateEngine ()`
This method is called every time an engine is needed, and there might be multiple engines existing at the same time. For example: when you have set your app as system wallpaper and is trying to open the wallpaper picker, there would be two engines, one for the actual system wallpaper and one for the preview in the picker. Thus, the `WallpaperService.Engine` class should be treated as the atomic unit of the whole system.

So how does the engine work, then?

## WallpaperService.Engine

The WallpaperService.Engine subclass, as previously said, is the base operation point of live wallpapers, including lifecycle callbacks, touch handling, and drawing target switches. The first and foremost issue would be: how does Unity draw to the Engine subclass and consequently draw to the wallpaper?
In the Engine subclass, you will find these callbacks:

```
public void onSurfaceCreated(SurfaceHolder holder)
public void onSurfaceChanged(SurfaceHolder holder, int format, int width, int height)
public void onSurfaceDestroyed(SurfaceHolder holder)
```

And these are where we pass the surface to the Unity instance. The `UnityPlayer` class found in unity-classes.jar has such a method:

```
public boolean displayChanged(int var1, Surface var2)
```

The first parameter (int var1) means the Display index, normally you just pass 0 to it as the main display. The second parameter (Surface var2), is the surface you are about to assign to that specific display. Notice in this context, Display is a class in Unity3D, and it does not necessarily relate to a real display panel. Instead, a Unity Display is mapped with an Android Surface, and you can assign different surfaces with different indexes to construct a virtual multi-screen setup. Back to the topic of live wallpapers, now if you put those API together, it is obvious that we can call `UnityPlayer.displayChanged(int, Surface)` when `WallpaperService.Engine.onSurfaceCreated(SurfaceHolder)` is triggered, and after that, Unity would just render the scene (with the camera on that display index) to the surface provided by Android.

## UnityPlayer

Now, as we learned how to pass the Surface to Unity, we can have a look at the UnityPlayer java class and see if there are other notable methods. For example:

```
public static void UnitySendMessage(String var0, String var1, String var2)
```

This one should be the most familiar method if you tried to write native plugins in the past. the var0 is the GameObject name, var1 is the method name in a component on that GameObject, and var2 is the parameter you want to pass to that method.

```
public UnityPlayer(Context var1)
```

This is the constructor of the UnityPlayer class. Now notice it only requires a Context instance, not an Activity instance. And this is where we

would inject our ApplicationContext to let Unity run even if there is no activity running. However, as UnityPlayer occasionally checks if the context is an instance of Activity to ensure certain functionalities like running on the UI thread and deciding screen orientation, we have to do some workaround in later stages.

### Application Startup

Normally, the UnityPlayer instance would be constructed within the default Activity implementation. But in live wallpapers, the activity isn't the only entry point. Imagine if the user reboots with your live wallpaper set as system wallpaper, the activity wouldn't open until you manually click the icon, but the wallpaper service is automatically called by the Android system. In that sense, we need to init UnityPlayer as soon as the Application instance creates, or even earlier.
Therefore UniLWP.Droid uses ContentProvider class to initialize Unity. ContentProvider is an early-init technique used by Firebase and AndroidX libraries, more info can be found on the firebase blog, in which Android developers from Google detailed why ContentProvider can be inited way early than normal onCreate() calls in the Application class.
By making Unity to init early, the app is always ready when a service or activity needs to access the Unity instance, and thus lower the possibility of exceptions. However, this also means the app loads the Unity instance whenever your app is alive even if your app was just awakened by other apps. This is intended behavior, but UniLWP.Droid does provide such an option for you to manually init Unity instead automatically if you want.

### Switch Surface

What we need to do next is to switch the surface as soon as a new surface is created. (We assume that the newest surface is usually also the visible one). It is easy to register such callbacks on WallpaperService, but how do we do it on Activity, though?
UniLWP.Droid uses a customized activity instance to replace the originally exported activity. The magic happens at the compile stage: As multiple AndroidManifest.xml files are combined, the remove tag on in the UniLWP.Droid aar file is merged into the main manifest, and consequently, removes the declaration of the original activity. Learning this is very important, as in later stages when we want to customize or replace the default activity provided in UniLWP.Droid, we will be using the same technique.

## Comparison

So what makes UniLWP.Droid different from other assets? I used to work on Skyline with 'uLiveWallpaper', a popular live wallpaper asset, and it helped Skyline to be once the most paid app on Google Play. However, that great asset stopped updating which forced me to write my own framework. With the experience in both frameworks, I can now outline what UniLWP.Droid is and isn't to let you make a better decision on whether or not to purchase or use this asset.

**UniLWP.Droid is:**

- It is a highly customizable framework that if done right, makes people barely know this is a Unity-based app.
- It is completely up to you to decide how to present your wallpaper and related settings, may it be a preview activity with a surface view, a preview activity with transparent background, or no activity at all, UniLWP.Droid can handle all that.

**UniLWP.Droid isn't:**

- It is not a framework that you can use out-of-the-box. Although you can just import the project and build a live wallpaper out of it, it requires you to have certain Android programming experience if you want to customize its behavior.
- It does not provide any way for you to integrate Ads / IAP for now. You need to handle that yourself, primarily through Android programming using your own Activity implementation.

## Still have questions?

Please write an email to justzht+unilwp@gmail.com, or contact me using this page: fincher.im. Both English and Chinese tech supports are available.

# Installation

## Import

Download UniLWP.Droid from the asset store and import it. Normally it would be extracted to the /FinGameWorks/UniLWP/Droid folder, with an optional demo at /FinGameWorks/UniLWP/Droid.Demo.
The UniLWP folder contains files explained as below:

- Editor: editor scripts for building players and post-build executions
- Plugins: contains the UniLWP jar file
- Scripts: contains some Singletons for you to register callbacks and call methods from the C# side.
- Settings: contains some ScriptableObjects that records certain settings like estimated build path.

## Menus

After importing UniLWP, your Unity window will have one additional menu item at 'FinGameWorks/UniLWP/Droid'. It contains these sub-items:

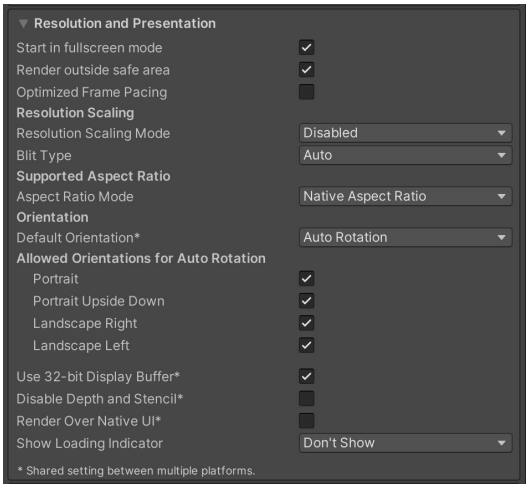
- Build to temp and copy to main
- Build debug to temp and copy to main
- Copy to main [release/debug]
- Init main project
- Delete module.xml
- Delete iml files

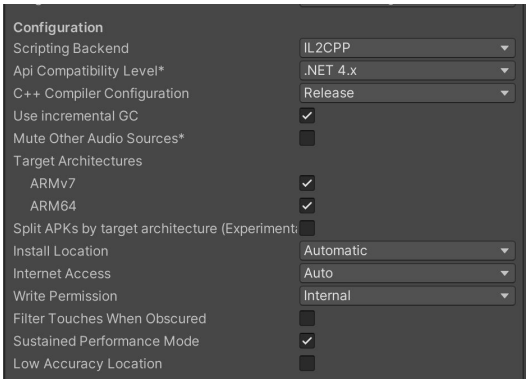
To understand what these items do, we need to introduce a certain structure of organizing folders. Open the path info file at FinGameWorks/UniLWP/Droid/Editor/Settings/BuildPathInfo, you will see three text fields, they are Temp(Release), Temp(Debug), and Main project path. The workflow is that you firstly init the main project, which creates an Android project at the path specified in the path info file, and then you update the Unity contents by building to temp paths and copy contents back to the main path. Thus you can continue your Android developments on the main project without the fear that it might be overwritten by your next Unity build, as the copy process only updates files in a sub-project called ‘unityLibrary’, not ‘launcher’.
Now it would be obvious what will these menu items do, except the ‘delete’ item. Why delete module.xml? It is because sometimes when you copy the newly built Unity data back to the main project, the Android Studio might not recognize the structure and treat the ‘unityLibrary’ sub-project as a non-Gradle project. If you delete the xml file Android Studio would treat it as a fresh project and reimport the structure.

## Build Settings

For UniLWP to work, you need to adjust several build setting items.

- In the ‘Player / Resolution and Presentation’ section
  - uncheck the optimized frame pacing option
  - uncheck render over native UI option
- In the ‘Player / Other Settings’ section
  - Uncheck mute other audio sources option
  - Set minimum API level to Android 7.0
  - Turn off filter touches when obscured, or you won’t receive touch events
- And lastly, go to audio settings and check ‘disable Unity audio’




# Usage

## First-time export

Instead of building your player through the Unity build panel, you need to click the menu item ‘FinGameWorks/UniLWP/Droid/Init Main Project’ to trigger a build for the base project. Before this, you might want to modify

the export path by editing the field in FinGameWorks/UniLWP/Droid/Editor/Settings/BuildPathInfo.
This project would be used as the main Android project that you can develop your own native features. Later patches would only replace the 'unityLibrary' sub-project without affecting your own work.
It is also recommended to add a gitignore file so that you only stage files in the main project, not exported temp projects.

# Export Workflow

After your first export, you only need to use item 'FinGameWorks/UniLWP/Droid/Build Temp And Copy To Main' to update the exported project. Unity would build the project to a temp path also set in FinGameWorks/UniLWP/Droid/Editor/Settings/BuildPathInfo, and copy the 'unityLibrary' folder to replace the one in the main path.

# Configurations

## Callbacks

UniLWP.Droid provides those callbacks to the C# side, they can all be accessed within LiveWallpaperManagerDroid.Instance:

### Window Insets

```
OnInsetsUpdatedDelegate(int left, int top, int right, int bottom)
```

This is a callback for window insets. Normally you won't have the issue if you are using Unity in a WallpaperService, but if you are in a preview Activity, certain phones have notches that can block your Unity UI elements and this callback would be useful, you can draw a safe area using the padding value provided by this callback.

### Wallpaper Offset

```
OnWallpaperOffsetsUpdatedDelegate(float xOffset, float yOffset, float xOffsetStep, float yOffsetStep, bool simulated)
```

This is a callback for launcher pages. It is similar to the onOffsetsChanged callback in WallpaperService.Engine, but with an additional bool value: simulated.
Certain Android launchers, like OneUI launcher, would not report real values when the user is swiping across pages. In order to provide the same experience, the java side of UniLWP.Droid introduces a swipe gesture recognizer within the wallpaper engine that works only if it detects such abnormal behavior. Then the simulated field would be true, meaning the value is simulated by the gesture recognizer, not the real launcher.

### Dark Mode

`OnDarkModeEnableUpdatedDelegate(bool darkMode)`

This is the callback for Dar Mode. If you live wallpaper needs different appearance like what Google's Pixel LWP does, then you can listen to this callback and reacts when the value is updated.

### Phone Unlock State

```
OnScreenDisplayStatusUpdatedDelegate(Enums.ScreenStatus screenStatus)
public enum ScreenStatus {
            LockedAndOff = 0,
            LockedAndAOD = 1,
            LockedAndOn = 2,
            Unlocked = 3
}
```

This is the callback for screen status changes. To draw different contents before and after phone unlock, you will need to listen to this callback. Notice that `LockedAndAOD` works only when your Android version is 9 and declares `'androidprv:supportAmbientMode'` in your wallpaper.xml. On Android 10 this feature is protected by a permission that can only be acquired if you compile your own ROM with the same signature.

### Is In Settings Activity

`OnEnterActivityUpdatedDelegate(bool inActivity)`

If you use Unity UI as your settings UI, then it would require you to hide the UI when the user is going back to the desktop. This callback is called when your settings UI is becoming visible or returning to the background so you can hide certain elements.

## Concept of wrapper

To construct a flexible structure that allows people to customize, UniLWP.Droid has a wrapper pattern that if followed, can help you to easily replace the default implementation of activity or service with your own subclasses.
A wrapper in UniLWP.Droid consists of these optional interfaces (you can find a detailed list in the LiveWallpaperPresentationEventWrapper.java file):

```
void setupSurfaceViewInActivityOnCreate(SurfaceView surfaceView)
void setupSurfaceViewInActivityOnDestroy(SurfaceView surfaceView)
void setupSurfaceHolderInServiceOnCreate(SurfaceHolder surfaceHolder)
void setupSurfaceHolderInServiceOnDestroy(SurfaceHolder surfaceHolder)
void configurationChanged(Configuration newConfig)
```

```
void onLowMemory()
boolean touchEvent(MotionEvent event, int[] additionalOffset)
void windowInsets(WindowInsets insets)
void offsets(float xOffset, float yOffset, float xOffsetStep, float
yOffsetStep, boolean simulated)
void serviceVisibility(boolean visible, SurfaceHolder surfaceHolder)
void activityVisibility(boolean visible, SurfaceHolder surfaceHolder)
void intent(Intent intent)
```

The naming of these interface methods is pretty self-explanatory. You call those methods in your own activity or service subclasses, and it would be forwarded to the Unity side. For example, the default LiveWallpaperPresentationActivity.java contains such calls:

```
public class LiveWallpaperPresentationActivity extends Activity
{
    protected SurfaceView surfaceView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        surfaceView = new SurfaceView(this);
        setContentView(surfaceView);
LiveWallpaperPresentationEventWrapper.getInstance().setupSurfaceViewInActivityOnCreate(surface
View);
    }

    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        LiveWallpaperPresentationEventWrapper.getInstance().intent(intent);
    }

    @Override
    protected void onResume() {
        super.onResume();
        LiveWallpaperPresentationEventWrapper.getInstance().activityVisibility(true,
surfaceView.getHolder());
    }

    @Override
    public void onLowMemory() {
        super.onLowMemory();
        LiveWallpaperPresentationEventWrapper.getInstance().onLowMemory();
    }

    @Override
    public void onTrimMemory(int level) {
        super.onTrimMemory(level);
        LiveWallpaperPresentationEventWrapper.getInstance().onLowMemory();
    }

    @Override
    protected void onPause() {
        LiveWallpaperPresentationEventWrapper.getInstance().activityVisibility(false,
surfaceView.getHolder());
        super.onPause();
    }
```

```
    @Override
    protected void onDestroy() {
        super.onDestroy();

LiveWallpaperPresentationEventWrapper.getInstance().setupSurfaceViewInActivityOnDestroy(surfac
eView);
    }

    @Override public void onConfigurationChanged(Configuration newConfig)
    {
        super.onConfigurationChanged(newConfig);
        LiveWallpaperPresentationEventWrapper.getInstance().configurationChanged(newConfig);
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        int[] windowPos = new int[2];
        surfaceView.getLocationInWindow(windowPos);
        return LiveWallpaperPresentationEventWrapper.getInstance().touchEvent(event, new
int[]{-windowPos[0],-windowPos[1]});
    }
}
```

### How to make your own wrapper (and register it)

Now, if you have created your own wrapper in your activity or services, how do you register it and replace the default ones? You just need to remove the old ones in AndroidManifest.xml by adding a tool:remove tag on the old ones.

## Behavior config

In this section, we will discuss several behavior configurations.

### Live Wallpaper Only

This one is pretty simple: just add a tool:remove tag to remove the default activity.

### Live Wallpaper with a Settings Activity (Default)

This is the default setup. There is a standard activity with a preview surface view in it.

### Live Wallpaper with a Settings Activity (Variant 1)

Now if you want to have an activity that has no surface view and is transparent to see through the desktop background, you can create your own activity class and change the layout file. Then your activity wrapper don’t need to call surface view related interfaces, as there isn’t a surface view in your layout. To achieve transparent effect, you need to declare a theme with a translucent background.

### Live Wallpaper with a Settings Activity (Variant 2)

Still, you can just use your activity as a settings activity that has a solid color background and no surface view. This one is similar to variant 1, just without the theme.

### Live Wallpaper with a Settings Preview

If you want your app opens instantly with the preview UI, then you can just create an activity that calls openPreview() in the onCreate() callback. Then on the Unity side, you can listen to the preview state changes, and display settings UI if it is in preview.