http://dx.doi.org/10.5392/IJoC.2012.8.4.042

An Efficient Interruption Handling for 3D Games based on Android Platform

Doan Quang Viet, Chang Yeol Choi
Department of Computer and Communications Engineering
Kangwon National University, Chuncheon, 200-701, Korea

ABSTRACT

Recently the unprecedented progress of Android platform and Google Play has brought more opportunities for individual mobile game developers. When playing an Android 3D game, users can accidentally hit HOME Key or BACK Key or receive an incoming message. Subsequently, another screen will appear on top and make the game application lose focus, or the operating system pause that game. These interruption events may also lead to the loss of game information or the game runs out of user control if that game cannot catch interruption events itself and resume at the exact time. As same as other mobile platforms, Android platform also provides Activity Lifecycle methods to help the game application control interruption problems. However, these methods are not adequate to solve all the interruption events. By implementing ANDCube game, we examine most of the possible interruption cases and propose some solutions to help Android game developers avoid some common interruption cases. Concurrently, we show the ways how a game application can catch all unavoidable interruption events and effectively resume from interruption to obtain a high quality game.

Keywords*: Android, interruption, 3D game, life cycle.*

1. INTRODUCTION

Imagine when someone is playing a popular mobile game with tons of entertaining and terrific sounds, he suddenly receives a phone call. Those mentioned sounds are still playing and he finds no way to turn back to that game to stop it in order to listen to the caller at best. On the other hand, while playing a fighting game, he is trying to hide from the enemy; a system dialog such as the alarm clock or the message notification appears on top. They act like a hindrance, and prevent him from controlling his character. However, the game still embarrassingly continues and his character dies consequently. Especially, consider a high configuration game with the long-term character's accumulation, that sudden out-of-control state may lead to the game achievements' loss. It would add a serious impact to the game quality.

While playing a mobile game application, game users always want the game to run continuously from the beginning to the end without any break. While many applications in personal computer can run at the same time, in Android platform, other applications or events can interrupt current application [1], [2]. An unavoidable interruption event makes the game pause or stop and game users cannot interact with Game User Interface (Game UI) although it is visible, partially visible or not. The requirement is after resuming, the "game status" remains the same as before [3].

Unlike traditional game applications, in mobile game applications, interruption handling is a crucial testing part [4], [5]. Complete solving interruption problems is critical for such a high quality Android 3D game.

The same as Java Micro Edition platform, Android platform also provides us with a set of application lifecycle methods to help the game system control itself instead of waiting for the operating system to handle [6]. They help the game system catch and control interruption events by triggering them when interruption events happen. However, Android lifecycle methods are not always triggered correctly in all cases [2].

In this paper, we present the experience of all common interruption events in Android environment and propose solutions to avoid some interruption cases subsequently. Going through the implementation of ANDCube, we also show remedies for unavoidable interruptions and ways of effective resumption in order to get a higher quality game.

2. BACKGROUND

2.1 Can We Avoid Interruption?

The game runs without any moment of interruption is the best for any Android applications. In fact, it depends on whether the platform allows the game system to catch and ignore an interruption event and return to the game immediately or not. On the other hand, it also depends on game user's expectation. For example, game users may want to

* *Corresponding author, Email: cychoi@kangwon.ac.kr*
Manuscript received Jun. 29, 2012; revised Sep 19, 2012; accepted Sep 29, 2012

receive a notification from alarm dialog or an incoming call. It will cause an interruption and vice versa.

2.2 What to handle before and after Interruption Events?

Avoiding interruption events is better but in case the game system must face unavoidable one, it has to catch and control that interruption event so that the game runs the same "game status" as before interruption. The "game status" includes following elements.

Game progress includes all the game elements which users create when playing the game such as game story, game scene, game setting, time and score. Before interruption events happen, game system should stop the game thread and save the last "game progress". Afterwards, when the game resumes, game system restarts game thread and reloads game data. Then, game users can continue to play the game with the same status as before. The game system can save them on application's memory. However, it is the best to save them on the persistent storage to prevent the operating system from deleting them for some reasons.

Graphics rendering is the extremely crucial part of a game. In an Android 3D game, when the game changes to PAUSED or STOPPED state, graphics configuration such as OPEN GL ES context or graphic resource such as textures can be lost. Stop rendering before interruption events happens to save battery. Game system should recreate or reload graphics configuration and resource, and restart rendering after game resumes.

Input events can cause the game run as unexpected state when the system still enables the inputs, while an interruption event happens concurrently. In some cases, game users cannot interact with the game interface by touching but sensor events still affect the game play. We should stop receiving all input events in interrupting time and start to receive them when the game resumes.

Sound or all audio elements should also be stopped when interruption events occur and resumed when the game resumes. A typical game uses two kinds of sound, i.e. music sound with long duration, and effect sound with its duration around 1 second. While the role of music is a background sound, game system triggers sound effect in situations. With the short time and no loop playing, the stopping, pausing or resuming functions have no meaning to the effect sound. Thus, we do not need to control sound effect in interruption cases. Unlike the effect sound, music can play a long time and loop infinitely. Therefore, when interruption events occur, the game system needs to pause or stop music and resume it when the game resumes.

2.3 Android 3D Game Lifecycle and Interruption

There are four different types of application components namely Activity, Services, Content provider and Broadcast receiver. Activity, which has user interface and can be interacted with users, [8] is the core part of a game application. While some Android applications can use several activities in one application, most of the Android 3D game applications use unique activity. On the other hand, if an Android 3D game uses several activities, then the INGAME phase, which users start the game story to the end, usually uses one single activity. Therefore, we assume that an Android 3D game application uses unique Android Activity. Each Activity specifies the lifecycle as the Android Activity Lifecycle [2], [5]. The concepts of Android 3D game lifecycle and Activity lifecycle can be considered as identical.

An activity can transit among four states: RUNNING, PAUSED, STOPPED and SHUTDOWN. Each of these states can be changed by seven methods which can be triggered when an interruption event happens. Figure 1 shows a diagram of four states and seven life cycle methods.

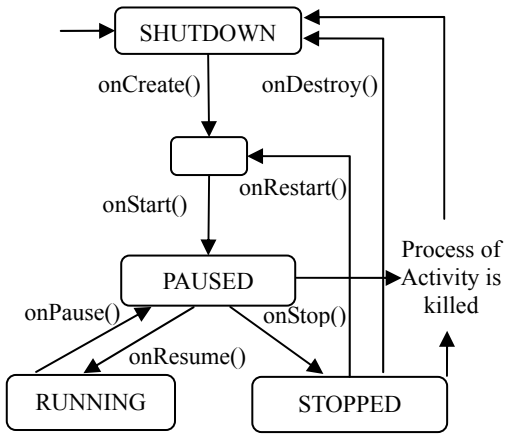


Fig. 1. Android Activity Lifecycle Model [2]

An activity is changed to RUNNING state after returning from calling onResume(). Game users can interact with it as foreground screen. If an activity is in RUNNING state and it loses user focus but still partially visible, the state is changed to PAUSED state. Method onPause() is triggered. In PAUSED state, after returning from calling onStop(), the activity is changed to STOPPED state. It still exists in the background but no longer be visible. Activity is changed to SHUTDOWN state after returning from calling onDestroy(). The activity is fully stopped and no longer exists. Moreover, in PAUSED or STOPPED state, the activity may be killed by operating system if foreground activity needs more memory resources. It is called KILLABLE state. From HoneyComb version, the activity is not KILLABLE after calling onPause(). Because more than 90% of Android devices run on older versions than HoneyComb [7], we assume that the activity is KILLABLE after calling onPause(). In KILLABLE state, the game system should save game data for resuming stage.

3. ANALYSIS OF ACTIVITY LIFECYCLE AND INTERRUPTION EVENTS

This section discusses the disadvantages of Activity lifecycle in determining interrupting and resuming time, interruption types and resuming stage in some special screens.

3.1 Android Activity Lifecycle Problems

As a general solution to solve interruption events, the game application should override lifecycle methods and manually handle instead of expecting operating system to catch and

control them. In Figure 1, before the game is changed to RUNNING state, onResume() method is always called even though it launches from SHUTDOWN state or resumes from STOPPED or PAUSED state. Therefore, in interruption context, game system does not need to override onCreate(), onRestart() and onStart() methods. Similarly, onPause() will always be called before onStop() and onStop() will always be called before onDestroy(). After all of the three calling methods, the activity is in the same KILLABLE state. Game system can also safely ignore the onStop() and onDestroy() methods. In other contexts, the game system should override lifecycle methods to set up, load or release some application components. It means PAUSED, STOPPED and SHUTDOWN can be grouped in a single state. Figure 2 shows the state transition after grouping among lifecycle methods and the standard way of interruption handling [6]. The RUNNING state is between onResume() and onPause().

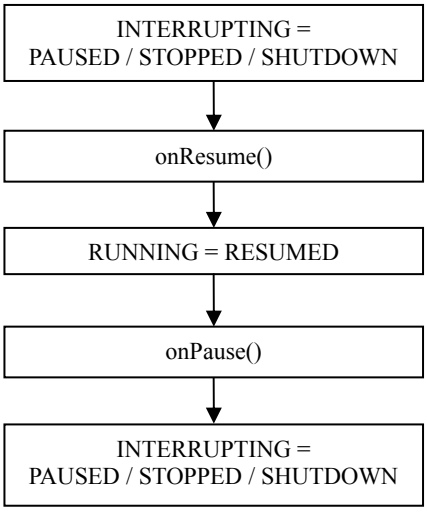


Fig. 2. Standard interruption handling [6]

We experienced some interruption events in the two Samsung devices, i.e. Galaxy Tab using Android version 2.2 - Froyo and Galaxy S using Android version 2.3.3 - Gingerbread. These two Android versions are currently the most popular ones in the world and cover approximately 74% of all running Android devices [7].

When the alarm dialog appears as top screen, or users receive an incoming call, incoming message, or press the HOME KEY, then the operating system will stop that game. If users press BACK KEY, the operating system will destroy that game. The sequence of lifecycle methods for these 2 devices of all the above cases plus the case when the game launches at the first time, and device configuration changed is normal and as follows:

-Activity:onPause()
-Activity:onStop ()
And when the game resumes:
-Activity:onRestart()/Activity:onCreate()
-Activity:onStart()
-Activity:onResume()

When game users scroll down the top bar, press and keep POWER Key or press and keep HOME KEY, a dialog appears as a top view and cover partially device screen as Figure 3. Although Game UI is partially visible and game user cannot interact with Game UI, the method onPause() is not called here. It means the game still remains RUNNING state and users cannot control the game flow. After the game resumes, no any lifecycle methods is triggered. In this case, after a break time, the game returns to user with another story because the game system cannot catch interruption events exactly and pause game story. This case happens on both Galaxy Tab and Galaxy S device.

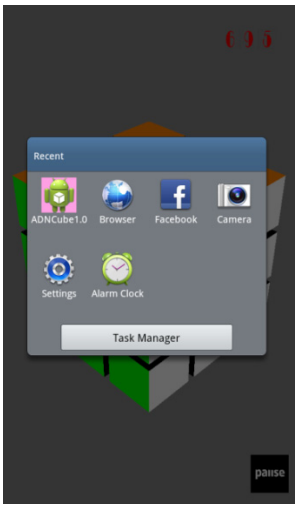


Fig. 3. Game UI is partially visible when users press and keep the HOME Key

When the screen is time out, users press POWER Key one time or users turn off the screen, game users cannot interact with game interface. Method onPaused() is called here as normal process. After game users press POWER Key again, the screen turns on and method onResume() is called. It means the game application is changed to RUNNING state. However, the screen is still locked and game users, therefore, cannot interact with game interface. If game application resumes at this time, that game will run out of user control. Thus, after game users unlock the screen, Game UI will be totally visible without any methods called and users can interact with the game application. This problem only occurs on Galaxy Tab device. On Galaxy S, no method is called when the screen is turned on and onResume() is called exactly when the screen is unlocked.

If the RUNNING state of an activity is always between two methods onResume() and onPause(), game application can catch interruption events exactly and has a way to control them effectively. Two above cases prove that when running on a real device, this rule does not always keep exactly.

3.2 Interruption Types

Interruption events can be understood under two meanings – making the game stop temporarily at the PAUSED state, or the game is totally stopped at SHUTDOWN state [7]. Both situations can be considered as interruption since they make the

game application stop, game users cannot continue to interact with Game UI and play the game. In PAUSED state, the game can back to RESUMED state in the previous screen. In SHUTDOWN state, we can only return to game after it restarts itself or game users manually restart the game application then go through some screens to reach the previous screen.

An Android 3D game can meet many interruptions from inside to outside of mobile devices. We classify them into 3 categories.

- Internal events are from operating system,, happening without users' actions. These events can make the device suspend or shutdown. Furthermore, they also can pause or stop the game. The alarm dialog, low battery dialog, the system shutdown due to lack of battery and the screen being time out are examples for this kind of event.
- External events are from outside of operating system and users such as incoming call or message.
- User's action: When playing the game, user unintentionally or intentionally makes the game pause or end if pressing a function key like HOME Key, BACK Key, POWER Key or VOLUME Key, using a system function like choosing a notification item from the top bar or changing the device system's locale or orientation.

3.3 Resuming in some Special Screens

A typical game may have many screens with their transitions and at least one menu screen and INGAME screen. Although game users want the games remain the last "game status" as much as possible, but in some special screens, the last one is not the best choice and we should return to user a better state which is decided by themselves. For example, if the game returns in "Exit confirmation screen" to ask users whether to exit the game or not, even the last choice is YES or NO option, the game system should highlight NO option to suggest that the user had better return to the game.

4. DESIGN AND IMPLEMENTATION: ANDCube

4.1 Game Screen Transition Diagram

In order to fully experience interruption problems, we implement ANDCube game as an Android version of a classic game Rubik's Cube [9]. Game users can touch directly on the screen to select and rotate a layer to play the game.

ANDCube game includes some screens and their transition as Figure 4. We can experience interruption events in each screen. Each screen involves multiple elements including a button, a control and the rendering of the game world. Each transition between two screens can be triggered by a user's interaction or the game system.

In ANDCube screen transition diagram, "INGAME menu screen" is a subset of the main menu screen. It helps game users conveniently change the game configuration instead of backing to the main menu while they are still playing the game.

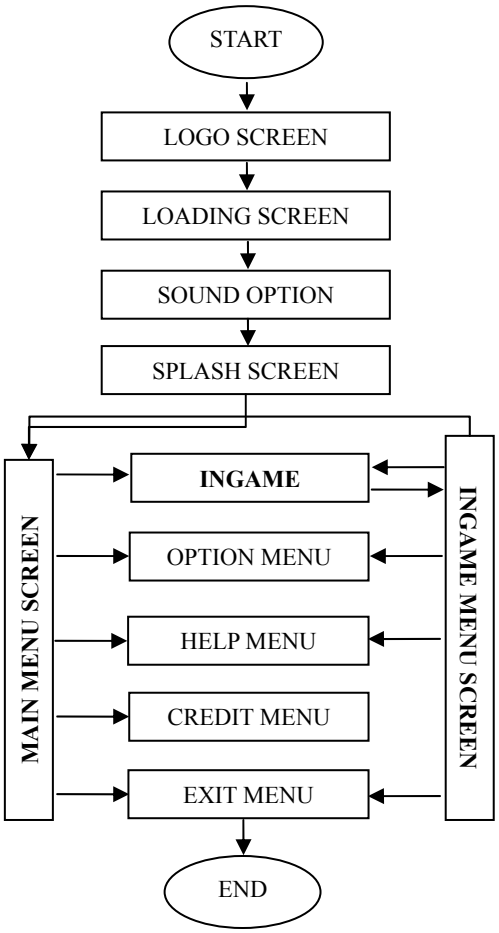


Fig. 4. ANDCube screen transition diagram

4.2 System Design

In order to render 3D graphics on the surface, ANDCube uses GLSurfaceView which is a type of view and allows the game system draw OpenGL ES [10]. GLSurfaceView is a subclass of View class. Thus, it has three calling methods surfaceCreated(), surfaceChanged() and surfaceDestroyed() to indicate when GLSurfaceView is created, changed or destroyed respectively.

GLSurfaceView clients implement the interface Renderer and make OpenGL calls render a frame. The Renderder interface has three methods onSurfaceCreated(), onSurface Changed() and drawFrame(). When GLSurfaceView is created, method onSurfaceCreated() is triggered. The same for onSurfaceChanged(), it is triggered when GLSurfaceView is resized. The Renderer will run on a separate thread and method drawFrame() run continuously from the GLSurfaceView is created to its destruction. Figure 5 shows their relationship to cover graphic part of ANCube game.

GLSurfaceView is created after Activity starts and destroyed before Activity stops. Therefore, method drawFrame() also runs in PAUSED state. In order to prevent GLSurfaceView rendering when Activity is not RUNNING, ANDCube pauses GLSurfaceView at PAUSED state and resumes them at RUNNING state by calling two methods of GLSurfaceView class onPause() and onResume() respectively. In this case, the OpenGL ES rendering context is typically lost when the Activity is in PAUSED state and all OpenGL associated with

that context will be automatically deleted. Therefore, after the game resumes from an interruption, OpenGL resources must be re-created. ANDCube does it on onSurfaceCreated() method of Renderer.

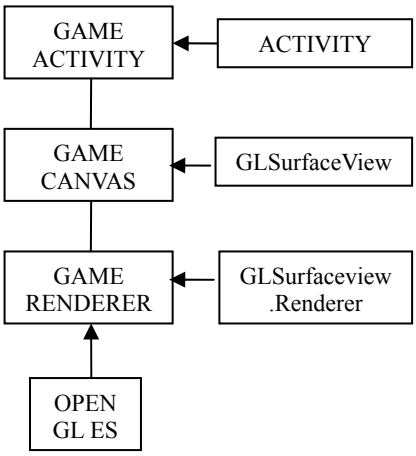


Fig. 5. Graphic part of ANDCube game.

4.3 Windows Focus Method and Interruption Handling

Two functions related to window focus and not contained in lifecycle methods but still triggered when activity is gained or lost focus are Activity:onWindowFocusChanged() and GLSurfaceView:onWindowFocusChanged(). Each of these functions is passed a Boolean argument that indicates the window is gained or lost. Windows focus methods are independently called from activity lifecycle methods. It means when the activity is RUNNING state, activity does not always gain focus. In case Activity is lost focus, game users though cannot interact with the game.

4.3.1 Interruption catching and controlling: The previous part mentioned some interruption events which cannot be caught by Activity lifecycle methods. To fix this, ANDCube uses both Activity lifecycle method and Windows Focus method to determine interrupting and resuming time of interruption events.

In case users scroll the top bar or press and keep the POWER Key/HOME Key, the sequence of methods is:
-Activity:onWindowFocusChanged(FALSE)
-GLSurfaceView:onWindowFocusChanged(FALSE)

Method onPause() is not called but onWindowFocus Changed() is triggered with FALSE argument. It also means Game UI is lost user focus. The game system can use windows focus method to catch this interruption event exactly. Then, when resuming, the sequence of methods is:
-Activity:onWindowFocusChanged(TRUE)
-GLSurfaceView:onWindowFocusChanged(TRUE)

Method onWindowFocusChanged() is called with TRUE argument while onResume() is not triggered. The game system can also use windows focus method to determine resuming time exactly.

When the screen is time out or users press POWER Key one time to turn off device screen, the sequence of methods is as follows:
-Activity:onPause()
-GLSurfaceView:surfaceChanged()
-Activity:onWindowFocusChanged(FALSE)
-GLSurfaceView:onWindowFocusChanged(FALSE)

There is no problem here because onPaused() is triggered. In addition, onWindowFocusChanged() is also triggered exactly with TRUE argument. And when turning on the screen:
-Activity:onResume()

Although method onResume() is called but the current game state is not really RUNNING because users cannot control the game. Therefore, using method onResume() to catch the resuming time is wrong in this case.

After unlocking screen, the sequence of the calling method:
-Activity:onWindowFocusChanged(TRUE)
-GLSurfaceView:onWindowFocusChanged(TRUE)

Game UI is visible and game application is gained focus. It is really in RUNNING state: the game is RESUMED state and users can control it. In this case, method onWindowFocusChanged() determines the resuming time exactly.

Using both lifecycle method and Windows Focus method helps game application determine interruption events and resuming time exactly on both Galaxy tab and Galaxy S device. ANDCube implements new interruption handling as Figure 6.

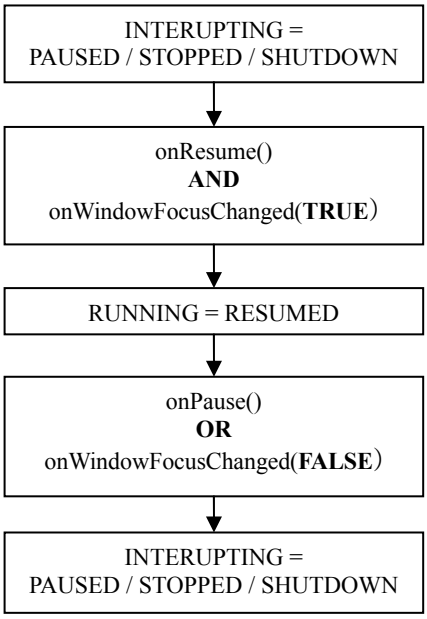


Fig. 6. Interruption controlling in ANDCube.

With new interruption handling, lifecycle model in Figure 1 can be extended as in Figure 7.

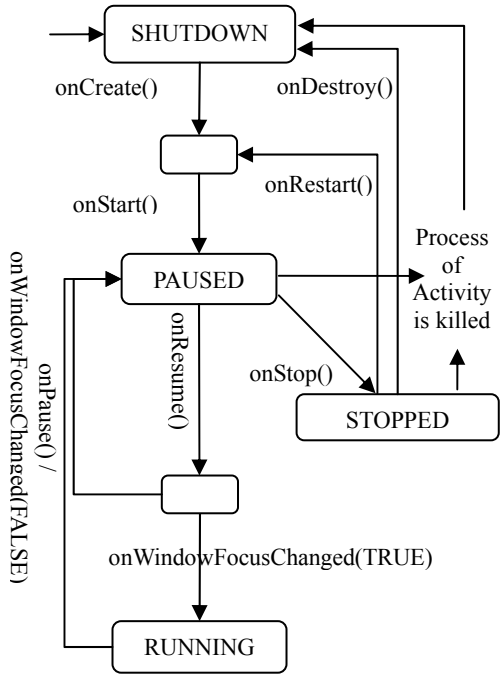


Fig. 7. ANDCube game state transition diagram

While onResume() is called before onWindowFocuss Changed(TRUE), the game system can prepare all game data in onResume() method for resuming stage. The following pseudocode presents ANDCube interruption handling.

```
void function onResume
{
        Set GameResume to TRUE;
        If   WindowFocus is equal to TRUE
        {
                RESUME GAME;
        }
        Else
        {
                PREPARE data for resuming;
        }
        Call the super function;
}

void function onPause
{
        Set GameResume to FALSE;
        PAUSE GAME;
        Save data;
        Call the super function;
}

void function onWindowFocusChanged(boolean isFocus)
{
        Set WindowFocus to isFocus;
        If GameResume is equal to TRUE and
WindowFocus is equal to TRUE
        {
                RESUME GAME;
        }
        If GameResume is equal to TRUE and
        WindowsFocus is equal to FALSE
        {
                PAUSE GAME;
        }
}
```

4.3.2 Interruption avoiding: When an interruption event occurs, one or more methods will be triggered. It seems to be possible to avoid interruption by overriding the calling method and choosing not to return supper method. In fact, it does not help the game avoid killable states.

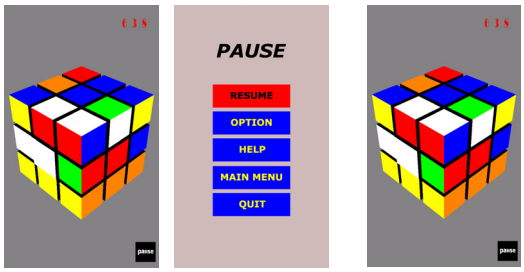
Because external events such as incoming call or incoming message are high priority with most users, we do not avoid them. The same answer happens with internal events. Thus, game system should avoid users' action interruption as much as possible. Firstly, setting game application in full screen mode prevents users from seeing and interacting with top bar. There are many interruption events caused by key pressing. However, the game can avoid Key interruptions by overriding Key Input Events methods. Except POWER Key and HOME Key which cannot be manually handled, others cannot make any interruptions. In order to prevent configuration changing interruption, which can make the game application restart, we should specify the android:configChanges attribute in the manifest and handle onConfigurationChanged() method to change the game application if it needs to adapt to the new configuration.

4.3.3 Resuming in some special screens: In INGAME phase, when the game resumes, the INGAME menu screen must be displayed. It takes time for users to be ready for returning to the game. The confirmation and option screen have two or more options for users. When the game resumes, the best option, which is assessed by the system, will be marked although the other options have already marked before interruption. In the "Exit confirmation screen", "Game reset screen", the NO option is highlighted to tell game user come back to the game. In the menu screen or INGAME menu screen, when the game resumes, the menu item marked must be the same as the before. If there is no marked menu item before interrupting, the RESUME item will be in high priority.

5. RESULT AND DISCUSSION

ANDCube game is implemented and tested on Samsung Galaxy Tab and Samsung Galaxy S device with Android OS version 2.2 Froyo and 2.3.3 Gingerbread respectively. This ANDCube can avoid some interruption cases and catch exactly the other ones.

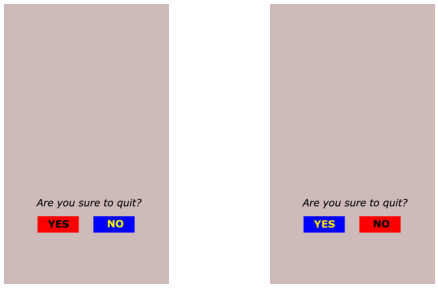
When interruption events occur in INGAME state, ANDCube saves all necessary game information and stops playing sound and receiving input. At resuming stage, the INGAME menu firstly displays and allows game user to have time to be ready to return the game as Figure 8. After the game returns, the game status remains the same as before interruption.



a b c

Fig. 8. Interruption and resuming in INGAME phase (a) The game status before interruption (b) After the game resumes, INGAME menu appears and allows game users to be ready to return the game (c) When game returns, the game status is the same as before.

Figure 9 shows interruption events and resuming in "Exit confirmation screen". Although before interruption, YES option menu is highlighted, when resuming NO option is marked to suggest game user returning the game. It is also a good design for any type of applications.



a b

Fig. 9. Interruption and resuming in "Exit confirmation screen" (a) Before interruption, YES option is selected (b) After the game resumes, NO option is highlighted to suggest coming back to the game.

Table 1. Comparison of interruption handling between standard method and ANDCube

	Standard [2, 6]	ANDCube
Method	Use Activity lifecycle methods	Use Activity lifecycle methods and window focus methods
Interruption catching and resuming	Cannot catch some interruption events and resume in wrong time.	Catching and resuming exactly all common interruption events
Result in game	-The game still runs while user cannot interact with Game UI. -The resuming game status is different from before interrupting.	-The game pauses immediately when interruption events happen. -The resuming game status is the same as before interrupting.

Table 1 shows the comparison between standard interruption handling and ANDCube handling. With standard interruption handling, the game may not catch some interruption events. Game users cannot interact with the game interface but the game still runs and causes some unexpected situations. ANDCube implementation catches all interruption events and game users can return to the game safely with the same game situation as before being interrupted. In standard handling, for some cases which do not catch the interruption events, the time component of that game may still be counted while users cannot control the game. After the game resumes, the game status will be different from before.

6. CONCLUSION

Differing from personal computers where many applications can run simultaneously, an application starts can interrupt another running application in Android platform. With the hope that Android 3D games can run continuously, users expect the game applications themselves can have the ability to avoid interruptions. Otherwise, after resuming from an interruption event, the game application must preserve previous game status.

In this paper, we experienced all common interruption events and proposed solution to avoid interruption events. With unavoidable ones, the implementation of ANDCube showed how to determine interrupting and resuming time exactly and proposed solutions for resuming in some special screens. It helps game system solve interruption completely and achieve a high quality game.

ANDCube interruption handling works well on Android 2.0 and 2.3.3. However, with the particular specification of each real device, few special mobile devices may process differently under the same interruption circumstances such as they do not call a lifecycle method or make it happen at variable orders.

Besides solving interruption events, we can also reuse the game screen transition diagram as a template when designing and implementing an Android 3D game. ANDCube interruption handling technique can be applied to Android based 3D game development and testing.

REFERENCES

[1] R. Meier, *Professional Android Application Development*, Wiley Publishing, Indiana, 2009.

[2] D. Franke, C. Elsemann, and S. Kowalewski, "Reverse Engineering of Mobile Application Lifecycles," *Proc. The 18th Working Conference on Reverse Engineering*, 2011, pp. 283-292.

[3] D. Franke, S. Kowalewski, C. Weise, and N. Prakobkosol, "Testing Conformance of Life Cycle Dependent Properties of Mobile Applications," *Proc. IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 241-250.

[4] T. Dumaresq, M. Villeneuve, "Test Strategies for Smartphones and Mobile Devices", Macadamian, 2010.

[5] S. Srirama, R. Kakumani, A. Aggarwal, and P. Pawar, "Effective Testing Principles for the Mobile Data

Services Applications," *IEEE Workshop on Software for Wireless Communications and Applications*, 2006, pp. 1-5.
[6] Android Developer Reference, *Activity class*. Available at http://developer.android.com/reference/android/app/Activity.html.
[7] Android Developer Resources, *Platform Versions*. Available at http://developer.android.com/resources/dashboard/platform-versions.html.
[8] E. Burnette, *Hello, Android: Introducing Google's Mobile Development Platform*, The Pragmatic Bookshelf, Indiana, 2008.
[9] Wikipedia, *Rubik's Cube*. Available at http://en.wikipedia.org/wiki/Rubik's_Cube.
[10] M. Zechner, *Beginning Android Games*, Apress, 2011.

Doan Quang Viet
He received his B.Eng in Computer Science from Ho Chi Minh City University of Technology, Vietnam in 2009 and M.Eng in Computer and Communications Engineering from Kangwon National University, Republic of Korea in 2012. He has 2 years of work experience in Gameloft Studio in Vietnam, as a mobile game programmer and producer. His main research interests include mobile game applications and geographic information systems.

Chang Yeol Choi
He received the B.S, M.S in electronics engineering from Kyungpook National University, in 1979, 1981 respectively and also received Ph.D. in computer engineering from Seoul National University in 1995. Before joining the School of Computer Science and Engineering, Kangwon National University in 1996, he has been with Electronics and Telecommunications Research Institute (ETRI) during 1984-1996, where he served as a principal researcher of Computer Research Division. His research interest lies in the field of computer architecture, embedded system and ubiquitous services.