Embedded System Software

Mini Game Project

(과제 수행 결과 보고서)

과목명: [CSE4116] 임베디드시스템소프트웨어

담당교수: 서강대학교 컴퓨터공학과 박 성 용

학번 및 이름: 20171664, 이상윤

개발기간: 2024. 06. 14. -2024. 06. 24.

최 종 보 고 서

I. 개발 목표

모듈 프로그래밍, 안드로이드 프로그래밍 및 JNI, Top Half(TH), Bottom Half(BH), timer, fpga control 등 강의 내 배운 부분을 활용하고, 배우지 않은 부분인 OpenGL ES 2.0을 활용하여 미니 게임들을 플레이 할 수 있는 프로그램을 완성한다.

II. 개발 범위, 동기 및 내용

가. 개발 범위

1. 2D 게임을 개발하기 위한 OpenGL ES 2.0의 렌더링을 구현한다.
2. 안드로이드 프로그래밍을 통해 최상단 프로그램의 틀을 완성한다.
3. 모듈에서 안드로이드로 인터럽트 정보를 전달할 수 있도록 한다.
4. 각 게임에 따라 fpga가 적절히 동작할 수 있도록 한다.
5. 각 게임의 로직을 구현한다.
6. JNI를 통해 위 사항을 적절히 연결한다.

나. 개발 동기

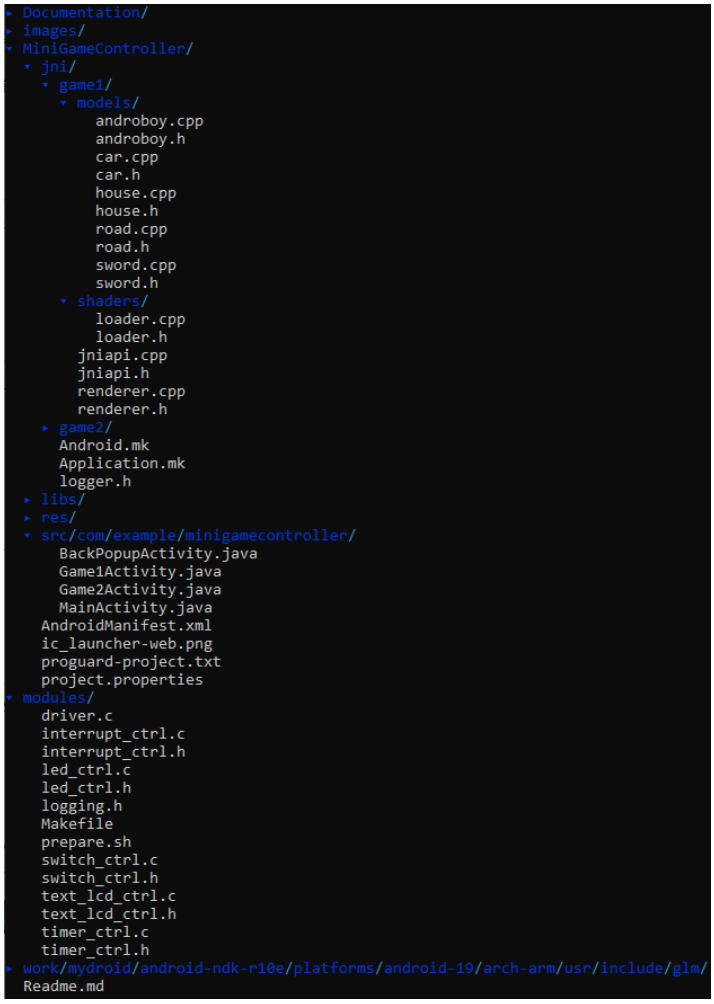
<그림 1: 개발 동기>

보드를 처음 접하였을 때 입력 디바이스 형태는 게임 패드와 유사하다는 생각을 하였다. 자유 주제 프로젝트의 진행을 모듈 프로그래밍을 통해 디바이스와 상호작용하는 게임을 만들면 좋겠다고 판단하였다.

<그림2: 개발 동기>

Android Kitkat은 OpenGL ES 2.0을 지원하며, 이 라이브러리는 그래픽스 라이브러리로서 NDK와 친화적이다. Java단에서 매번 OpenGL ES 2.0 함수를 호출 시 매번 JNI를 통해 네이티브 레이어까지 내려가기 때문이다. NDK를 통해 개발하는 프로젝트 특성 상 이 점을 활용하면 좋겠다고 판단하였다.

다. 개발 내용



<그림1: 프로젝트 개발 디렉토리 구조>

1. Documentation와 images 폴더는 단순히 프로젝트를 설명하는 문서와 사진들이 저장되어 있는 폴더이다. work/mydroid/android-ndk-r10e/…/include/ 디렉토리는 외부 라이브러리를 명시적으로 나타내기 위해 따로 첨부하였다. glm을 사용하며, 버전은 0.9.5.4(2014-06-21, Header-Only library) 이다.
2. modules/ 디렉토리는 모듈 프로그래밍을 위한 코드들이 들어있다. driver.c에 file operations들이 구현되어 있으며 나머지 기능들은 파일 이름과 같이 일반적인 기능을 담당한다. timer_ctrl.c는 FND와 DOT에 타이머 인터럽트를 이용해 시간을 표시해주는 역할을 한다.
3. MiniGameController/src/com/example/minigamecontroller 디렉토리는 안드로이드 프로그래밍을 위한 자바 파일을 포함한다. 각각 서로 다른 Activity를 나타낸다. 또

한 MiniGameController/res/ 도 안드로이드 프로그래밍을 위한 여러 파일들을 포함한다.

4. MiniGameController/jni/gameX/models 디렉토리는 화면에 렌더링할 물체의 모델링을 담당한다.
5. MiniGameController/jni/gameX/shaders 디렉토리는 OpenGL ES 2.0에서 2D 게임을 위한 Vertex Shader, Fragment Shader 들을 구현한다.
6. MiniGameController/jni/gameX/jniapi.cpp 파일은 JNI 함수를 통해 Java단과 C/C++단을 연결하는 역할을 한다.
7. MiniGameController/jni/gameX/renderer.cpp 파일은 모델들을 적절히 계산하고, 디바이스 파일을 이용하는 등 게임의 핵심 로직들을 구현한다. 즉, Java 레이어부터 MiniGameController/src/com/example/minigamecontroller -> jniapi.cpp -> renderer.cpp 순서로 플로우가 진행된다.

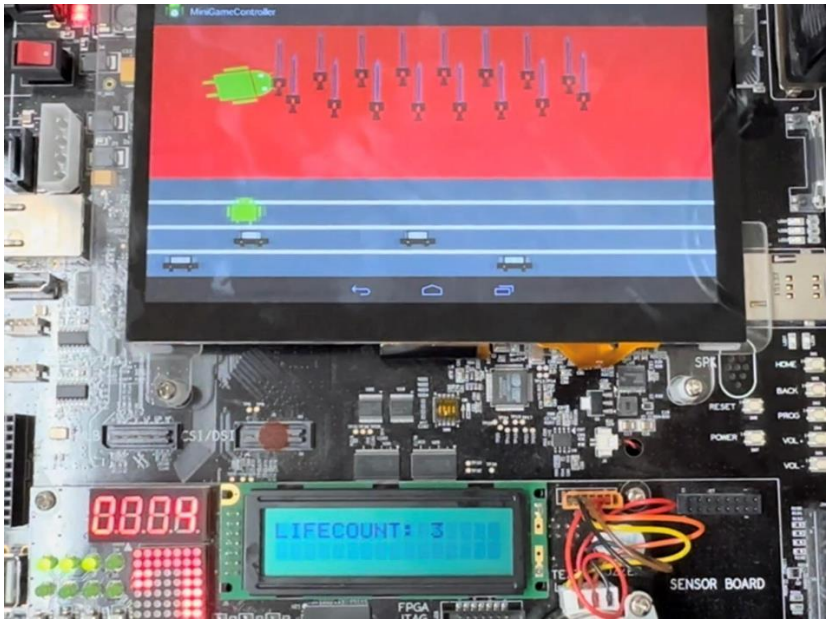
III. 추진 일정 및 개발 방법

가. 추진 일정

- 2024/06/14: 개발 환경 세팅 및 분석
- 2024/06/15 ~ 2024/06/16: OpenGL ES 2.0 환경 세팅
- 2024/06/17: OpenGL ES 2.0 환경 세팅 및 외부 라이브러리(glm) 세팅
- 2024/06/18: 디바이스 드라이버 구현
- 2024/06/19 ~ 2024/06/21: Game1(Car Avoidance Game) 구현
- 2024/06/22 ~ 2024/06/23: Game2(Maze Game) 구현
- 2024/06/24: 로직 수정 및 버그 수정

나. 개발 방법

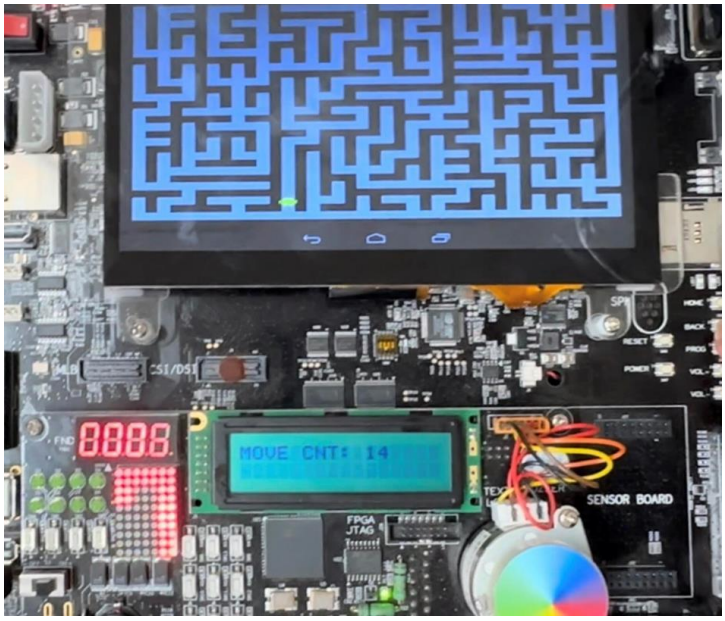
1. Car Avoidance Game 설명



<그림1: game1 시작화면>

사용자는 dip switch(1~9버튼) 중 상하좌우 네개의 버튼으로 화면상의 androboy를 이동시킬 수 있다. Back Button(interrupt)을 눌려 게임을 정지할 수 있으며, 이 상태에서 게임을 Resume, Restart, Exit 할 것인지의 여부를 결정하는 작은 팝업창이 뜬다. 매 0.1초마다 Timer가 작동되며 DOT의 숫자를 1씩 올린다. 또한 1초가 되면 FND상의 숫자를 1 씩 올린다. 그리고 LED와 TEXT LCD는 목숨을 나타낸다. <그림1>처럼 시작 상태는 3개의 목숨으로 시작하며 androboy가 3개의 차를 모은 경우 엔딩모션이 발생하며 게임은 종료된다.

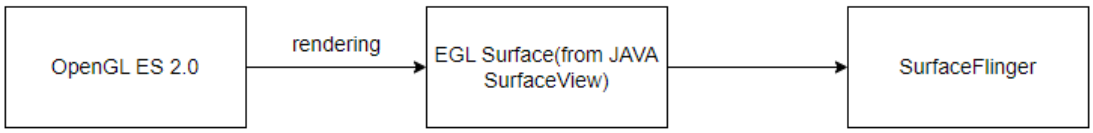
2. Maze Game 설명



<그림2: game2 사진>

게임1과 마찬가지로 dip switch를 통해 andromam을 상하좌우로 조작할 수 있고 Back interrupt로 게임을 정지하고 팝업창을 켤 수 있으며 Timer가 같은 원리로 동작한다. goal point까지 도착하기 위해 이때까지 움직인 횟수가 TEXT LCD에 표시된다. 만약 goal point에 도달한다면 엔딩모션과 함께 게임이 종료된다.

3. OpenGL ES 2.0



<그림5: architecture-rendering>

기본적으로 OpenGL ES 2.0을 통해 EGL Surface에 렌더링을 하면, 자동으로 SurfaceFlinger가 이를 화면에 나타내준다. 즉, 렌더링을 통해 적절히 EGL Surface에 표현하기만 하면 된다.

```
struct egl_status{
        ANativeWindow* window;
        EGLDisplay display;
        EGLSurface surface;
        EGLContext context;
        EGLConfig config;
        EGLint numConfigs;
        EGLint format;
        EGLint width;
        EGLint height;

        bool exists_window;
};
static struct egl_status egl;
```

<그림6: EGL context, renderer.cpp>

하지만 OpenGL ES context는 그것 자체만의 스레드를 요구한다. 즉, 스레드마다 위 구조체 안의 내용을 따로 가지고 있어야 한다. 뿐만 아니라 성능상, nonblock 등의 여러 이유로 rendering thread를 직접 따로 만들 필요가 있다.

```
GLuint g1_shader_program;
GLint g1_loc_position, g1_loc_mvp_matrix, g1_loc_primitive_color;
void game1_shaders_init(void) {
        g1_shader_program = load_shaders();

        glUseProgram(g1_shader_program);

        g1_loc_mvp_matrix = glGetUniformLocation(g1_shader_program, "u_ModelViewProjectionMatrix");
        g1_loc_primitive_color = glGetUniformLocation(g1_shader_program, "u_primitive_color");
        g1_loc_position = glGetAttribLocation(g1_shader_program, "a_position");

        glUseProgram(g1_shader_program);
        LOG_INFO("Init shader finish %d %d %d %d", g1_shader_program, g1_loc_mvp_matrix, g1_loc_primitive_color, g1_loc_position);
}

void game1_shaders_del(void){
        if (g1_shader_program != 0) {
                glDeleteProgram(g1_shader_program);
                g1_shader_program = 0;
        }
        struct shader_info* entry = shaders;
        while(entry->type != GL_NONE){
                glDeleteShader(entry->shader);
                entry->shader = 0;
                ++entry;
        }
}
```

<그림7: shaders/loader.cpp>

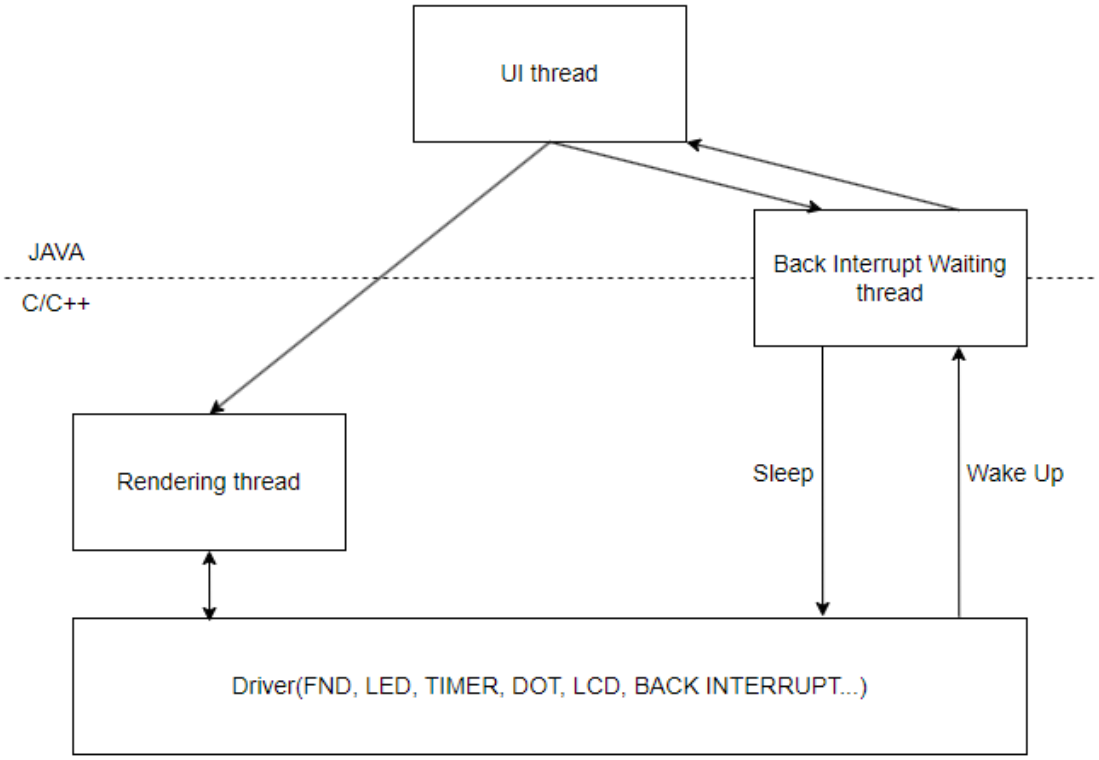
모델들의 좌표와 색을 계산하기 위한 Vertex Shader, Fragment Shader 또한 <그림7>처럼 rendering thread가 생성될 때 마다 로드 되어야 한다. 디바이스를 통해 게임을 조작할 수 있을 때 마다(onResume() 때 마다) 위에 언급한 shader, egl들을 다시 초기화하고 화면을 움직이게 하는 rendering thread가 생성되어야 한다.

```
m_matrix = glm::translate(glm::mat4(1.0f), glm::vec3(androboy_x, androboy_y, 0.0f));
m_matrix = glm::rotate(m_matrix, gameover_androboy_cur_time * TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
m_matrix = glm::scale(m_matrix, glm::vec3(0.9f, 0.9f, 1.0f));
mvp_matrix = vp_matrix * m_matrix;
glUniformMatrix4fv(g1_loc_mvp_matrix, 1, GL_FALSE, &mvp_matrix[0][0]);
androboy_draw();
 void androboy_draw(void){
        glBindBuffer(GL_ARRAY_BUFFER, androboy_vb);
        glEnableVertexAttribArray(g1_loc_position);
        glVertexAttribPointer(g1_loc_position, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_HEAD]);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 180);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_LEFT_EYE]);
        glDrawArrays(GL_TRIANGLE_FAN, 180, 360);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_RIGHT_EYE]);
        glDrawArrays(GL_TRIANGLE_FAN, 540, 360);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_LEFT_ANTENNA]);
        glDrawArrays(GL_TRIANGLE_FAN, 900, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_RIGHT_ANTENNA]);
        glDrawArrays(GL_TRIANGLE_FAN, 904, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_BODY]);
        glDrawArrays(GL_TRIANGLE_FAN, 908, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_LEFT_HAND]);
        glDrawArrays(GL_TRIANGLE_FAN, 912, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_RIGHT_HAND]);
        glDrawArrays(GL_TRIANGLE_FAN, 916, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_LEFT_FOOT]);
        glDrawArrays(GL_TRIANGLE_FAN, 920, 4);
        glUniform3fv(g1_loc_primitive_color, 1, COLORS[ANDROBOY_RIGHT_FOOT]);
        glDrawArrays(GL_TRIANGLE_FAN, 924, 4);

        glDisableVertexAttribArray(g1_loc_position);
}
```

<그림8, 9: androboy draw>

이후 실제로 화면에 나타내기 위해서 각 모델마다 버퍼를 준비한다. 그 후 위 <그림 6,7> 과 같이 Vertex shader에 사용되는 mvp matrix를 구하고 모델의 좌표를 바인딩한 후 fragment shader에 사용되는 색깔을 입히면 화면에 적절히 나타낼 수 있다.

4. Back Button Interrupt



<그림10: architecture-Interrupt>

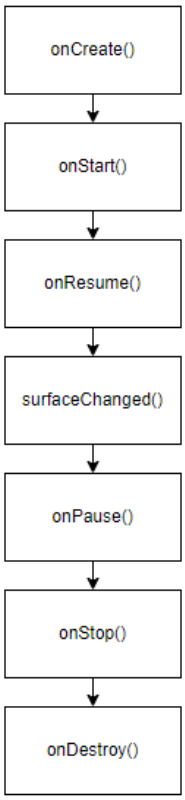
디바이스 파일을 이용하며 디스플레이 화면을 출력하는 Rendering thread 뿐만 아니라 BACK Button을 통해 인터럽트를 감지하기 위한 Back Interrupt Waiting thread도 필요하다. 이 스레드는 JAVA단에서 생성되어 네이티브 레이어로 내려와 디바이스 파일을 통해 인터럽트를 감지하기 전까지 sleep상태에 빠져든다. 그리고 인터럽트가 감지된다면 UI thread로 돌아가 이를 알리고, 새로운 게임 상태(Resume, Restart, Exit) 컨트롤을 위한 팝업 엑티비티를 생성한다. 즉, Back Button을 누르면 게임이 중지되며 게임을 지속할것인지, 재시작할것인지, 종료할 것인지의 여부를 결정할 수 있는 작은 팝업창이 떠야한다. <그림 10>은 이를 나타낸다.

5. Lifecycle Consideration

C/C++ rendering thread, JAVA Back Interrupt Waiting thread, JAVA SurfaceView lifecycle, onDestroy()는 항상 호출되지 않는다는 점, screen off 상태(PROG 버튼), 강제종료 상황 등 프로그램이 정상적으로 돌아가기 위해 고려해야할 상황이 많다.

이에 주요 요소들을 나열하면 아래와 같다.

- thread가 남아있지 않는 한 close()는 강제 종료 상황에서 명시적으로 호출하지 않아도 호출된다(아마 Garbage Collector의 영향때문).
- onCreate()는 activity가 생성될 때 항상 호출됨이 보장된다.
- surfaceChanged()는 surfaceCreate()될 때 최소 한번은 호출된다.
- onResume()->surfaceChanged()->onPause()->surfaceDestroyed()는 이 프로그램에서 보장된다. 가로화면을 세로화면을 바꾸거나 할 일 등이 없기 때문이다. 또한 만들어진 surface가 destroy되지 않았다면 onResume()다음의 surfaceChanged()는 무시된다.



<그림11: lifecycle>

이를 바탕으로 각 단계를 나열하면 <그림9>과 같고, 각 단계에서 구현해야할 주요 내용을 간단히 나타내면 아래와 같다.

- onCreate(): open device file and initialize fpga and game attributes
- onStart(): pthread_mutex_init(), 힙과 같은 자원할당
- onResume(): 중지되었다 실행될 때 rendering thread 생성, back interrupt waiting thread 생성, egl 속성과 fpga 세팅
- surfaceChanged(): 처음 실행될 때 rendering thread 생성, egl 속성 세팅

- onPause(): rendering thread와 back interrupt waiting thread 삭제
- onStop(): deallocation
- onDestroy(): close()

각 단계는 나름 자명하지만 onPause()단계에서 모든 스레드를 없애주는건 조금 애매한 부분이다. 이에 대해 설명하자면 우선, onPause()단계에서 렌더링은 더 이상 진행되지 않아도 된다. 게다가 화면이 정지된 상태일 때 rendering thread가 계속 cpu자원을 차지하게 두는 것은 낭비이다. 다음으로, 더이상 Back interrupt로 팝업이 뜨기 또한 기다리지 않아도 된다. 애초에 정지 상태를 위해 Back interrupt가 존재하기 때문이다. 따라서 강제종료든, 화면 off 상태이든, 정상적으로 back button을 클릭하였든 간에 onPause()에서 rendering thread와 back interrupt waiting thread를 제거하는 것이 적절하다. 즉, 추가적인 스레드들을 onResume() 단계에서 생성하고, onPause() 단계에서 삭제하는 것을 반복한다.

```
        back_prev_pressed = cur;

        waked_by_back_handler = 1;
        complete(&over);
        spin_unlock(&sl);
        return IRQ_HANDLED;
    }

void interrupt_wake_back_waiting_thread(void){
        spin_lock(&sl);
        waked_by_back_handler = 0;
        complete(&over);
        spin_unlock(&sl);
}
```

<그림12, 13: 인터럽트에 의해 스레드 깨우기와 강제로 스레드 깨우기, /modules/>

이렇듯 강제종료와 같은 경우 onPause()단계에서 명시적으로 남아있는 스레드를 제거해줄 필요가 있다. spinlock은 multi process 환경에서 interrupt context가 공유데이터를 접근하므로 필요하고, kernel context 또한 이를 접근하므로 필요하다. 또한 back interrupt에 의해 깨어났는지, 아니면 강제로 깨어났는지의 여부 또한 중요하므로 반환한다. 강제 종료와 같이 back interrupt로 인해 깨어난 경우에만 게임을 지속할것인지, 재시작할것인지, 종료할것인지의 여부를 결정할 수 있는 작은 팝업창이 떠야하기 때문이다. JAVA단에서 이 여부를 전달받아 새로운 팝업창을 띄우는 결정을 한다.

6. TH, BH

버튼이 빠르게 눌러지는 상황에서 back interrupt가 직관적으로 오지 않음을 관찰하였다. 여기서 직관적으로 오지 않는다는 뜻은 falling edge, rising edge를 완벽하게 구분할 수 없다는 점뿐만 아니라 사용자가 의도하지 않은 interrupt또한 종종 발생함을 의미한다. 즉, 대부분의 상황에서 사용자가 의도한 인터럽트와 더불어 다른 인터럽트가 추가로 발생한다. 이를 적절히 판별하는 로직을 TH에 구현한다. 게다가 back interrupt는 단순히 waiting thread를 깨우는 것 외에는 아무런 작업을 하지 않으므로 TH만으로 해결하도록 한다.

Timer interrupt는 TH, BH로 나누어져 있다. BH는 system workqueue를 이용한다. TH에선 메모리 상의 데이터를 수정만 하도록 하고 BH에선 실제로 fpga에 출력하는 다소 무거운 행위를 담당한다.

Ⅳ. 연구 결과

1. rendering thread

```
void game1_resume(void){
        pthread_mutex_lock(&gstate.mutex);
        gstate.is_paused = false;
        pthread_mutex_unlock(&gstate.mutex);

        // At most onResume() -> surfaceChanged() -> onPause() -> surfaceDestroyed().
        // However, if the created surface is not destroyed, from the second onResume onwards,
        // surfaceCreated and surfaceChanged will be ignored.
        // In other words, the rendering thread is initially started from surfaceChanged
        // and subsequently from onResume.
        if(egl.exists_window)
                pthread_create(&gstate.tid, NULL, render_loop, NULL);
        else
                LOG_INFO("First onResume()");

        ioctl(gpad.fd, IOCTL_RUN_TIMER_NONBLOCK);
}
```

<그림14: onResume(), renderer.cpp>

onResume() 시 JNI를 통해 renderer.cpp 해당 부분으로 들어오게 된다.

```
static void* render_loop(void* nouse){
        acquire_context();

        // change screeen vertical to horizontal and normalize coords.
        vp_matrix = glm::ortho(-egl.width / 2.0, egl.width / 2.0,
                -egl.height / 1.35, egl.height / 1.35, -1000.0, 1000.0);

        while(true){
                // consider onPause()
                pthread_mutex_lock(&gstate.mutex);
                if(gstate.is_paused){
                        release_context();
                        pthread_mutex_unlock(&gstate.mutex);
                        pthread_exit(0);
                }
                read_gpad();
                process_gameover();
                draw_frame();
                if (!eglSwapBuffers(egl.display, egl.surface)) {
                        LOG_ERROR("eglSwapBuffers() returned error %d", eglGetError());
                }
                ++gstate.cur_time;
                pthread_mutex_unlock(&gstate.mutex);
                usleep(10000); // 10ms
        }
}
```

<그림15: onResume(), renderer.cpp>

여기서 생성된 renderer thread는 <그림15>의 함수를 호출하고, 반복문을 돌며 렌더링하며 디스플레이에 화면을 표시한다. acquire_context(), release_context()를 통해 현재 스레드에서 egl을 설정한다. 렌더링 과정을 onPause()를 통해 정지상태가 될 때까지 지속된다. read_gpad()(=read game pad)는 디바이스를 통해 상하좌우를 움직이는 역할을 하며 process_gameover()는 게임오버 상태인지 확인하는 역할을 한다. 또 gstate.cur_time(game state의 current time)은 long long 타입으로서, 10ms 마다 1씩 올라간다. 이는 대략 십만년 이상의 긴 시간을 커버하므로 오버플로우는 걱정하지 않아도 된다. UI thread와 rendering thread가 공유 데이터를 서로 수정할 수 있으므로 mutex를 사용하였다.

2. back interrupt waiting thread

```
class BackInterruptDetector extends Thread{
        BackInterruptDetector(){}

        public void run(){
                Log.i(TAG, "Interrupt Dectector started");
                // Blocking manner
                if(nativeWaitBackInterrupt()){
                        Log.i(TAG, "Waked up by interrupt");
                        // wake up
                        Intent intent = new Intent(Game1Activity.this, BackPopupActivity.class);
                        intent.putExtra("CALLING_ACTIVITY", Game1Activity.class);
                        startActivity(intent);
                        overridePendingTransition(0, 0);
                }
                else{
                        Log.i(TAG, "Waked up by pause");
                }
        }
}
```

<그림16: back interrupt waiting thread, .java>

onResume()단계에서 JAVA단에서 위 스레드를 생성한다.

```
JNIEXPORT jboolean JNICALL Java_com_example_minigamecontroller_Game1Activity_nativeWaitBackInterrupt(JNIEnv* env, jobject obj){
        LOG_INFO("nativeWaitBackInterrupt");
        return game1_wait_back_interrupt();
}
```

```
/*
 * In a blocking manner, the back interrupt detector thread, which is passed in Java,
 * detects the back button interrupt.
 */
bool game1_wait_back_interrupt(void){
        int waked_by_intr;
        ioctl(gpad.fd, IOCTL_WAIT_BACK_INTERRUPT, &waked_by_intr);
        if(waked_by_intr) {
                LOG_INFO("waked up by interrupt");
                return true;
        }
        LOG_INFO("waked up by pause");
        return false;
}
```

<그림17, 18: nativewaitBackInterrupt()>

해당 스레드는 JNI를 통해 ioctl(IOCTL_WAIT_BACK_INTERRUPT)를 호출한다. 이는 <그림12, 13> 부분으로 들어가게 되며 깨어나게 된다. 깨어난 후 <그림16>으로 돌아가며, back interrupt에 의해 깨어난 경우에만 새로운 엑티비티를 시작한다. 이후 rendering thread도 종료되며 생성된 스레드들은 종료된다.

3. Popup Activity

```
// closeButton click listener
exitButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
                Log.i(TAG, "closeButton clicked");
                Intent callerIntent = getIntent();
                Class<?> callingActivity = (Class<?>) callerIntent.getSerializableExtra("CALLING_ACTIVITY");
                if(callingActivity != null){
                        Intent intent = new Intent(BackPopupActivity.this, callingActivity);
                        // reuse current Game1Activity
                        intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
                        intent.putExtra("EXIT", true);
                        startActivity(intent);
                        finish();
                }
        }
});
```

<그림19: Popup Activity-close button>

Back interrupt에 의해 중지된 경우 popup Acitivy가 뜨게되고, <그림19>는 해당 popup activity는 close button의 경우만을 나타낸다. intent에 불러진 activity를 그대로 설정하고 메시지를 설정한 후 다시 돌아간다. 현재 팝업창은 종료한다.

```
@Override
protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        setIntent(intent);
        if (getIntent().getBooleanExtra("EXIT", false)) {
                Log.i(TAG, "onNewIntent(EXIT)");
                finish();
        }
        else if(getIntent().getBooleanExtra("RESTART", false)){
                Log.i(TAG, "onNewIntent(RESTART)");
                nativeRestartGame();
        }
        else if (getIntent().getBooleanExtra("RESUME", false)) {
                Log.i(TAG, "onNewIntent(RESUME)");
        }
}
```

<그림20: onNewIntent()>

다시 원래의 게임 엑티비티로 돌아오고 나서 intent를 보고 행위를 결정한다.

4. Maze game

Game1에서 JNI를 이용하는데 있어 JAVA로 부터 객체를 인자로 받아오거나 반환값을 설정하는 등 여러 함수들을 활용하였다(game1/jniapi.cpp). 하지만 JAVA의 메서드를 따로 네이티브 레이어에서 호출하는 부분이 없었다. 따라서 이를 추가적으로

구현할 목적으로 미로 생성 함수를 JAVA에서 구현하고, 이를 네이티브 레이어에서 호출하는 식으로 하였다.

```
/**
 * Initialize maze using Eller's Algorithm.
 * It is proper to initialize maze in C(or C++), because I use OPENGL ES 2.0 and NDK,
 * but I declare maze in Java for educational purpose.
 */
private int[][] generateMaze() {
        // initialize
        int[][] ret = new int[2*ROW + 1][2*COL + 1];
        int[][] profile = new int[ROW][COL];
        for(int i = 0; i < 2*ROW + 1; ++i){
                for(int j = 0; j < 2*COL + 1; ++j){
                        if((i & 1) != 0 && (j & 1) != 0)
                                ret[i][j] = 0;
                        else
                                ret[i][j] = 1;
                }
        }
        for(int i = 0; i < ROW; ++i)
                for(int j = 0; j < COL; ++j)
                        profile[i][j] = i*COL + j;

        // Eller's Algorithm
```

<그림 21: Game2Activity>

<그림 21>의 함수는 Game2Activity에 존재하는 함수로서 미로를 생성하여 배열을 반환한다. 1으로 표시된 부분은 벽이 있는 것이며 렌더링 시 검은 타일로 표현할 것이고 0으로 표시된 부분은 벽이 없는 것이며 렌더링 시 회색 화면으로 표현할 것이다. 플레이어는 0으로 표시된 부분만 지나갈 수 있다. 해당 함수에서 미로를 생성하기 위해 엘러의 알고리즘을 사용하였다. 랜덤 완전탐색이랑 크게 다른 부분은 없고, 모든 빈 칸의 집합(profile)을 최종적으로 하나의 집합으로 표현할 수 있으면 모든 정점에서 모든 곳으로 갈 수 있으므로 승리할 수 있는 게임을 만들수 있다는 점을 이용한 간단한 알고리즘이다.

```
static std::vector<std::vector<int> > generate_maze(JNIEnv* env, jobject obj){
        jclass cls = env->GetObjectClass(obj);
        jmethodID mid = env->GetMethodID(cls, "generateMaze","()[[I");
        if(mid == NULL)
                LOG_ERROR("method id not found");

        std::vector<std::vector<int> > ret;
        jobjectArray res = (jobjectArray)env->CallObjectMethod(obj, mid);
        jsize row = env->GetArrayLength(res);
        ret.resize(row);
        for(int i = 0; i < row; ++i){
                jintArray jarr = (jintArray)env->GetObjectArrayElement(res, i);
                jsize col = env->GetArrayLength(jarr);
                ret[i].resize(col);
                jint* carr = env->GetIntArrayElements(jarr, NULL);
                for(int j = 0; j < col; ++j)
                        ret[i][j] = carr[j];
                env->ReleaseIntArrayElements(jarr, carr, 0);
        }
        return ret;
}
```

<그림 22: game2/jniapi.cpp>

JNI를 통해 해당 함수를 호출해 값을 받아온 다음, c++자료구조 값으로 바꾸고 이후 이 값을 사용한다.

V. 기타

임베디드시스템소프트웨어 강의에서 배운 모든 요소들을 사용할 수 있는 뜻깊은 경험이었습니다. 또한 저에게 새로운 기술이었던 안드로이드 프로그래밍을 익힐 수 있었고 네이티브 단의 라이브러리(OpenGL ES 2.0)을 익히며 성능을 고려하는 경험은 저에게 새로웠습니다. 마지막 학기 좋은 경험을 하게 해주어 감사합니다.