گزارش پروژه پایانی برنامه نویسی اسمبلی

Tennissembly

ساختار زبان و کامپیوتر - دکتر جهانگیر (زمستان ۱۴۰۳)
نرگس کاری

دانشکده مهندسی کامپیوتر

فهرست مطالب

مقدمه

بازی طراحی‌شده در این پروژه Tennissembly نام دارد که ترکیبی از دو واژه Tennis (ورزش تنیس) و Assembly (زبان اسمبلی) است. پروژه‌ی اولیه‌ی ما یک بازی تنیس گرافیکی است که با استفاده از زبان سطح بالای Java و کتابخانه‌ی گرافیکی JavaFX توسعه داده شده است. پس از پیاده‌سازی کامل پروژه در Java، بخش‌های محاسباتی زمان‌بر را به‌صورت توابع `static` در کلاسی به نام `MyNativeLibrary` جمع‌آوری کردم. سپس این توابع را در یک فایل C پیاده‌سازی کرده و آن را به یک کتابخانه‌ی بومی تبدیل کردم (`.dll` در ویندوز و `.so` در لینوکس).

پس از ساخت کتابخانه، آن را به مسیر کتابخانه‌های Java نصب‌شده روی سیستم اضافه کردم. در ادامه، با استفاده از **JNI** کتابخانه‌ی ساخته‌شده را در `MyNativeLibrary` بارگذاری کرده و به جای پیاده‌سازی توابع در Java، مستقیماً از نسخه‌ی بومی آن‌ها استفاده کردم. در نهایت، کد C موجود در این توابع را با **زبان اسمبلی** به‌صورت **assembly inline** بازنویسی کرده و مجدداً به Java متصل کردم.

کدهای پروژه و روند پیشرفت آن در GitHub قابل مشاهده است.

شکل ۱: صفحه اول بازی

توضیحات کلی بازی

۱.۲ قوانین و مقررات

در این بازی، هر بازیکن باید تلاش کند تا توپ را هنگام ورود به زمین خود، به زمین حریف بازگرداند. اگر توپ از بالای یا پایین صفحه عبور کند، یک امتیاز به حریف داده می‌شود.

در منوی بازی، قبل از شروع، بازیکنان می‌توانند تعداد امتیاز موردنیاز برای پایان بازی را تعیین کنند. بازی ادامه پیدا می‌کند تا زمانی که یکی از بازیکنان به امتیاز تعیین‌شده برسد.

لازم به ذکر است که در حالت **تمرین**، قوانین کمی متفاوت خواهد بود که در ادامه به آن خواهیم پرداخت.

۲.۲ روش اجرای بازی

کنترل بازی از طریق **موس** و **کیبورد** انجام می‌شود. بازیکنان می‌توانند راکت خود را با کشیدن (drag) روی صفحه حرکت دهند و با قسمت **توری** راکت به توپ ضربه بزنند. راکت بالای صفحه با کلیدهای **W**، **S**، **A** و **D** کنترل می‌شود، در حالی که راکت پایین صفحه با کلیدهای **I**، **K**، **J** و **L** هدایت می‌گردد. محل برخورد توپ با تور اهمیت زیادی دارد؛ جهت حرکت توپ پس از برخورد بستگی به نسبت فاصله نقطه برخورد با مرکز تور دارد. همچنین، اگر بازیکنی توپ را با بخش پایینی تور ضربه بزند، توپ از طرف خود او خارج خواهد شد. از طریق دکمه‌های موجود در سمت چپ صفحه، می‌توان نحوه حرکت توپ را تنظیم کرد که در ادامه توضیح داده خواهد شد. نکته مهم این است که توپ در صورتی که هنوز در هوا باشد، قابل ضربه زدن با راکت نخواهد بود.

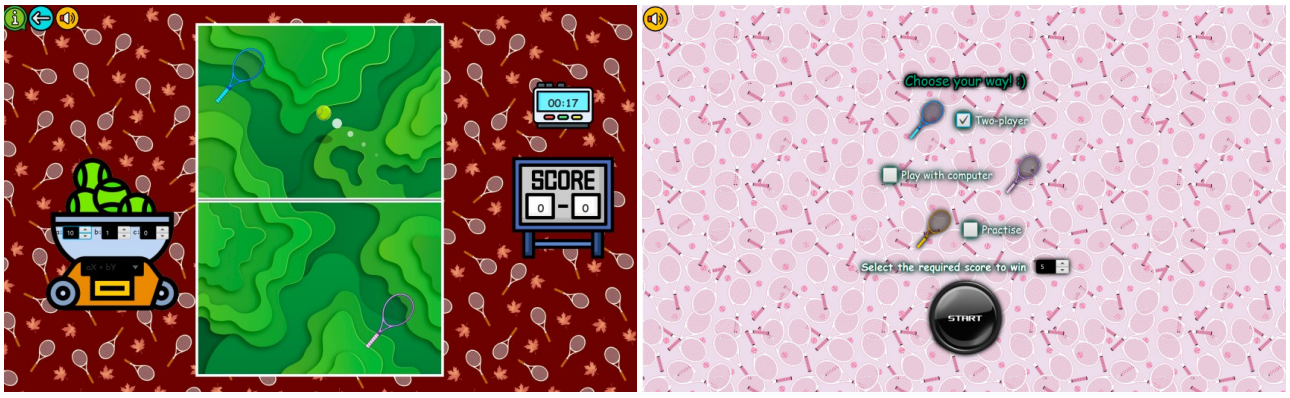
۳.۲ حالت‌های مختلف بازی

بازی دارای سه حالت مختلف است که می‌توان آن‌ها را انتخاب کرد: حالت دو نفره، حالت تک‌نفره (بازی با کامپیوتر) و حالت تمرین.

در **حالت دو نفره** ، هر دو بازیکن می‌توانند راکت‌های خود را کنترل کنند و بازی از دو طرف ادامه می‌یابد. هر بازیکن باید سعی کند توپ را به زمین حریف بازگرداند و از دریافت امتیاز منفی جلوگیری کند.

در **حالت بازی با کامپیوتر** ، یکی از راکت‌ها در اختیار شما خواهد بود و راکت دیگر تحت کنترل هوش مصنوعی است. شما همچنان می‌توانید نحوه‌ی حرکت توپ را مطابق با تنظیمات مورد نظر خود انتخاب کنید.

در **حالت تمرین** ، یک دیوار فرضی در یک سمت زمین قرار دارد که توپ در صورت برخورد با آن بازتاب می‌شود. در این حالت، تمرین تا زمانی که امتیاز از دست‌رفته شما به مقدار مشخص‌شده برسد (یا اگر به تعداد مشخصی امتیاز مثبت برسید)، ادامه خواهد داشت.



شکل ۲: صفحه منو و صفحه ی اصلی بازی

۴.۲ مکانیک حرکت توپ

در این بازی، سه نوع حرکت برای توپ در نظر گرفته شده است که بازیکنان می‌توانند از طریق گزینه‌های سمت چپ صفحه آن‌ها را انتخاب کنند. تغییرات حرکت توپ با زدن **دکمه نارنجی** اعمال می‌شود. این سه نوع حرکت عبارت‌اند از:

- **حرکت سینوسی:** در این حالت، حرکت توپ بر اساس معادله‌ی سینوسی $a\sin(bt+c)$ تنظیم می‌شود. این حرکت باعث ایجاد نوسانات عمودی در مسیر حرکت توپ میشود.
- **حرکت سهمی:** در این نوع حرکت، مسیر توپ از معادله‌ی درجه دوم at^2+bt+c پیروی می‌کند. محور t این معادله روی خط وسط زمین تنظیم شده است، در حالی که محور عمودی آن در راستای خارج از صفحه قرار دارد. اعمال این نوع حرکت به وسیله‌ی تغییر اندازه‌ی توپ و سایه‌ی آن صورت می‌گیرد، به طوری که توپ بین یک مقدار حداقلی و حداکثری از نظر ابعاد نوسان می‌کند. در صورتی که توپ در ناحیه‌ی بالایی صفحه قرار بگیرد (و اندازه‌ی آن بیشتر از مقدار معمول شود)، راکت‌ها قادر به ضربه زدن به آن نخواهند بود و توپ از بالای آن‌ها عبور می‌کند.
- **حرکت خطی:** این حرکت به صورت خطی با معادله‌ی $aX+bY$ تعریف می‌شود که در آن ضرایب a و b نسبت سرعت تغییرات را مشخص می‌کنند. در این حالت، مسیر حرکت توپ بدون نوسان و به صورت مستقیم خواهد بود. در این حالت میتوان با اعمال عدد c به سرعت عمودی توپ ضریبی دائم داد.

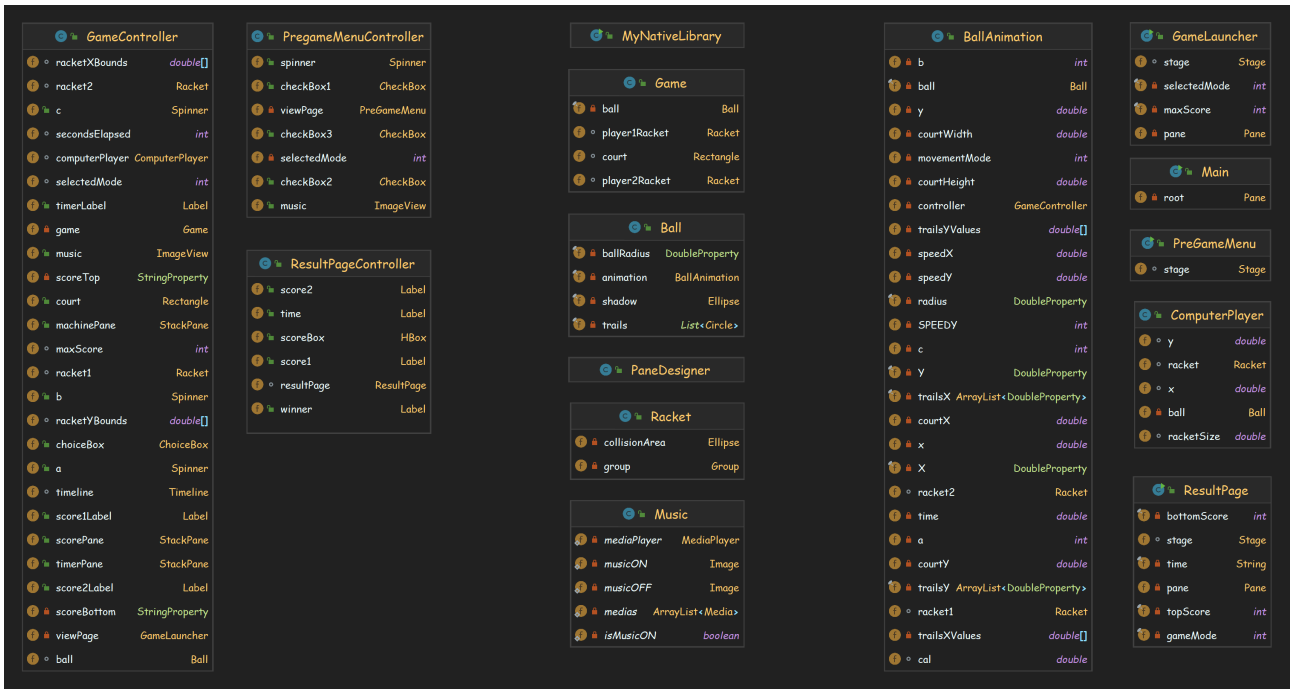
در هر یک از این حالات، بازیکنان می‌توانند مقادیر a, b و c را به صورت دستی وارد کرده و با فشردن دکمه نارنجی، تنظیمات موردنظر خود را روی توپ اعمال کنند. همچنین، بازیکنان می‌توانند با کلیک روی icon information که در گوشه‌ی بالای سمت چپ صفحه قرار دارد، مقادیر پیشنهادی برای ضرایب a, b و c را برای هر نوع حرکت مشاهده کنند.

پیاده‌سازی بازی با جاوا

این پروژه یک **بازی جاوا** با معماری **MVC** است که از JavaFX برای طراحی رابط گرافیکی استفاده می‌کند. کلاس‌های موجود در پروژه به سه بخش کلی تقسیم می‌شوند:

۱.۳ مدل $(Model)$ - مدیریت داده‌ها و منطق بازی

- **Ball**: مدیریت توپ، شامل شعاع، سایه و افکت‌های حرکتی.
- **Racket**: کنترل راکت‌ها و مدیریت برخوردهای آن‌ها.
- **Game**: شامل زمین بازی، بازیکنان و توپ.
- **Music**: مدیریت صدا و افکت‌های صوتی.
- **BallAnimation**: کنترل حرکت توپ و ثبت تاریخچه‌ی موقعیت‌ها.
- **ComputerPlayer**: کنترل هوش مصنوعی در حالت تک‌نفره.
- **PaneDesigner**: اعمال تغییرات گرافیکی بر روی صفحات بازی.
- **MyNativeLibrary**: استفاده از توابع **Native (مانند C/C++)** برای بهینه‌سازی اجرا.



شکل ۳: تصویر uml پروژه بدون فایل های پوشه تست

۲.۳ کنترلر $(Controller)$ - مدیریت تعاملات کاربر

- **GameController**: کنترل اصلی بازی، شامل زمان‌بندی، دریافت ورودی‌های کاربر و اعمال قوانین بازی.
- **PregameMenuController**: مدیریت منوی تنظیمات پیش از شروع بازی.
- **ResultPageController**: نمایش صفحه‌ی نتایج بازی.

۳.۳ ویو $(View)$ - نمایش رابط کاربری

- **PreGameMenu** و **ResultPage**: نمایش منوهای بازی.
- **GameLauncher**: مدیریت صحنه‌های بازی و تنظیمات گرافیکی.
- **Main**: نقطه‌ی ورود برنامه و مقداردهی اولیه‌ی رابط گرافیکی.

۴.۳ تست $(Test)$ - آزمودن میزان بهروی

- **PreviousLibrary**: شامل توابع جاوا که معادل توابع متصل‌شده با سی و اسمبلی هستند.
- **TestTime**: مقایسه‌ی عملکرد توابع محاسباتی پیاده‌سازی‌شده در سی و اسمبلی با معادل‌های جاوایی آن‌ها.
- **KeyPressTest**: مقایسه‌ی کارایی توابعی که با ورودی‌های صفحه‌کلید کار می‌کنند.

بهینه‌سازی با زبان‌های سطح پایین

۱.۴ کدهای زبان C و Assembly

این کتابخانه با استفاده از اسمبلی درون‌خطی ($InlineAssembly$) بهینه‌سازی‌هایی را برای افزایش کارایی برخی محاسبات اعمال می‌کند.

در توابعی که نیاز به پردازش سریع دارند، مانند calculateParabola و divRoundAwayFromZero ،از دستورالعمل‌های SIMD و پردازش برداری AVX/SSE برای افزایش سرعت محاسبات بهره گرفته شده است.

تابع calculateParabola به‌گونه‌ای طراحی شده که بتواند مقادیر را در چندین نقطه به‌طور هم‌زمان و بهینه محاسبه کند.

همچنین، در سایر توابع مانند myHypot و sin از دستورات اسمبلی برای کاهش سربار پردازش و بهینه‌سازی محاسبات ریاضی استفاده شده است.

```
#include <jni.h>
#include "model_library_MyNativeLibrary.h"
#include <math.h>
#include <stdlib.h>
#include <windows.h>
JNIEXPORT jdoubleArray JNICALL
    Java_model_library_MyNativeLibrary_calculateParabola(
    JNIEnv *env, jobject obj, jdouble time, jdouble a, jdouble b, jdouble
    c, jdouble speed) {
    jdouble result[4] __attribute__((aligned(32)));
    const double indices[4] __attribute__((aligned(32))) = {0.0, 1.0, 2.0
    , 3.0};  //
    const double scale = 0.001; //       0.001
    __asm__ volatile (
        // Load initial time into all 4 lanes of YMM0
        "vbroadcastsd %1, %%ymm0\n\t"
        // Create time increments: time, time+speed, time+2*speed, time+3
    *speed
        "vbroadcastsd %5, %%ymm1\n\t"   // Load speed
        "vmovupd %7, %%ymm2\n\t"        // Load {0,1,2,3} from memory
        "vmulpd %%ymm1, %%ymm2, %%ymm2\n\t"  // Multiply speed with {0,1,
    2,3}
        "vaddpd %%ymm2, %%ymm0, %%ymm0\n\t"  // Add to initial time
        // Multiply each time value by a
        "vbroadcastsd %2, %%ymm3\n\t"   // Load 'a'
        "vmulpd %%ymm0, %%ymm3, %%ymm3\n\t"
        // Add b to each lane
        "vbroadcastsd %3, %%ymm4\n\t"   // Load 'b'
        "vaddpd %%ymm4, %%ymm3, %%ymm3\n\t"
        // Multiply time by (time * a + b)
        "vmulpd %%ymm0, %%ymm3, %%ymm3\n\t"
        // Add c
        "vbroadcastsd %4, %%ymm5\n\t"   // Load 'c'
        "vaddpd %%ymm5, %%ymm3, %%ymm3\n\t"
        // Final multiplication by 0.001
        "vbroadcastsd %6, %%ymm6\n\t"   // Load 0.001 from memory
        "vmulpd %%ymm6, %%ymm3, %%ymm3\n\t"
        // Store results
        "vmovupd %%ymm3, (%0)\n\t"
        :
```

```
        : "r" (result),         // Output array
          "m" (time),           // Initial time
          "m" (a),              // Coefficient a
          "m" (b),              // Coefficient b
          "m" (c),              // Coefficient c
          "m" (speed),          // Speed increment
          "m" (scale),          // Constant 0.001
          "m" (indices)         // Indices {0,1,2,3}
        : "ymm0", "ymm1", "ymm2", "ymm3", "ymm4", "ymm5", "ymm6", "memory"
    );
    // Create Java double array
    jdoubleArray jResult = (*env)->NewDoubleArray(env, 4);
    (*env)->SetDoubleArrayRegion(env, jResult, 0, 4, result);
    return jResult;
}


JNIEXPORT jint JNICALL Java_model_library_MyNativeLibrary_divRoundAwayFromZero(JNIEnv *env, jobject obj, jdouble x, jdouble bound) {
    jint result;
    if (bound == 0) {
        // Handle division by zero error
        return 0; // Alternatively, throw an exception (e.g., jniThrowException)
    }
    __asm__ __volatile__ (
        "movsd %1, %%xmm0\n\t"   // Load x into xmm0
        "movsd %2, %%xmm1\n\t"   // Load bound into xmm1
        "divsd %%xmm1, %%xmm0\n\t" // Divide x by bound, result in xmm0
        "movapd %%xmm0, %%xmm2\n\t" // Copy xmm0 to xmm2 for later use
        "movapd %%xmm0, %%xmm3\n\t" // Copy xmm0 to xmm3 for reuse

        "xorpd %%xmm1, %%xmm1\n\t"  // Zero out xmm1 (sets it to 0)
        "comisd %%xmm0, %%xmm1\n\t" // Compare xmm0 (quotient) with zero
        "jb negative_case\n\t"      // Jump to negative_case if the quotient is negative

        // Positive case: rounding away from zero (rounding up)
        "addsd %3, %%xmm0\n\t"      // Add 0.5 to the quotient for rounding
        "cvttsd2siq %%xmm0, %%rax\n\t" // Convert rounded double to integer (truncating)
        "movl %%eax, %0\n\t"        // Move the 32-bit integer result into output variable
        "jmp done\n\t"              // Jump to done (skip negative case)

        "negative_case:\n\t"
        // Negative case: rounding away from zero (rounding down)
        "subsd %3, %%xmm2\n\t"      // Subtract 0.5 from the quotient for rounding
        "cvttsd2siq %%xmm2, %%rax\n\t" // Convert rounded double to integer (truncating)
        "movl %%eax, %0\n\t"        // Move the 32-bit integer result into output variable

        "done:\n\t" // Label marking the end of the assembly block

```

```
84          : "=r" (result)                 // Output: result variable
85          : "x" (x), "x" (bound), "x" (0.5) // Inputs: x, bound, and 0.5
        for rounding
86          : "xmm0", "xmm1", "xmm2", "xmm3", "rax" // Clobbered registers
87      );
88
89      return result;
90  }
91
92
93
94  JNIEXPORT jint JNICALL Java_model_library_MyNativeLibrary_asmAbs(JNIEnv *
        env, jobject obj, jint x) {
95      jint result;
96      __asm__ volatile (
97          "movq %1, %%rax\n\t"        // Move x into the 64-bit RAX register
98          "testq %%rax, %%rax\n\t"    // Check if the number is negative (if
        negative, the sign bit is 1)
99          "jns positive\n\t"          // Jump to 'positive' label if the
        number is already positive
100         "negq %%rax\n"              // If negative, negate the value in
        place
101         "positive:\n\t"             // Label for positive or modified
        numbers
102         "movl %%eax, %0\n\t"        // Store the 32-bit value (eax) into
        the result variable
103         : "=r" (result)              // Output: result variable
104         : "r" ((long long)x)         // Input: x, cast to 64-bit
105         : "%rax"                     // Clobbered register: RAX
106     );
107     return result;
108 }
109
110
111 JNIEXPORT jdouble JNICALL Java_model_library_MyNativeLibrary_sin(JNIEnv *
        env, jobject obj, jdouble time, jdouble a, jdouble b, jdouble c) {
112     jdouble result;
113     __asm__ volatile (
114         "fldl %4\n\t"           // st0 = c
115         "fldl %3\n\t"           // st0 = b, st1 = c
116         "fldl %2\n\t"           // st0 = time, st1 = b, st2 = c
117         "fmulp\n\t"             // st0 = b * time, st1 = c
118         "faddp\n\t"             // st0 = b * time + c
119         "fsin\n\t"              // st0 = sin(b * time + c)
120         "fldl %1\n\t"           // st0 = a, st1 = sin(b * time + c)
121         "fmulp\n\t"             // st0 = a * sin(b * time + c)
122         "fstpl %0\n\t"          //       result
123         : "=m" (result)
124         : "m" (a), "m" (time), "m" (b), "m" (c)
125         : "%st"
126     );
127     return result;
128 }
129
130 JNIEXPORT jdoubleArray JNICALL
        Java_model_library_MyNativeLibrary_updateTrailValues(JNIEnv *env,
        jobject obj, jdoubleArray trailsValues, jdouble value) {
131     jdouble *values = (*env)->GetDoubleArrayElements(env, trailsValues, 0
        );
```

```
    for (int i = 3; i > 0; i--) {
        values[i] += 0.1 * (values[i - 1] - values[i]);
    }
    values[0] += 0.1 * (value - values[0]);

    (*env)->ReleaseDoubleArrayElements(env, trailsValues, values, 0);
    return trailsValues;
}

JNIEXPORT jboolean JNICALL Java_model_library_MyNativeLibrary_isBetween(
    JNIEnv *env, jobject obj, jdouble p, jdouble q, jdouble r) {
    jboolean result = JNI_FALSE;
    __asm__ volatile (
        "movsd %1, %%xmm0\n\t"    // Move p to xmm0
        "movsd %2, %%xmm1\n\t"    // Move q to xmm1
        "movsd %3, %%xmm2\n\t"    // Move r to xmm2
        "ucomisd %%xmm1, %%xmm0\n\t"  // Compare p with q
        "setae %%al\n\t"          // Set AL if p >= q
        "ucomisd %%xmm2, %%xmm0\n\t"  // Compare p with r
        "setbe %%cl\n\t"          // Set CL if p <= r
        "andb %%al, %%cl\n\t"     // Logical AND of conditions
        "movb %%cl, %0\n\t"       // Move result to output
        : "+r" (result)
        : "x" (p), "x" (q), "x" (r)
        : "%xmm0", "%xmm1", "%xmm2", "%al", "%cl"
    );
    return result;
}

JNIEXPORT jdouble JNICALL Java_model_library_MyNativeLibrary_makeInBound(
    JNIEnv *env, jobject obj, jdouble m, jdouble min, jdouble max) {
    jdouble result;
    __asm__ volatile (
        "movsd %1, %%xmm0\n\t"      // Load `m` into XMM0
        "movsd %2, %%xmm1\n\t"      // Load `min` into XMM1
        "movsd %3, %%xmm2\n\t"      // Load `max` into XMM2
        // Compare and set minimum bound
        "maxsd %%xmm1, %%xmm0\n\t"  // Use MAXSD for m = max(m, min)
        // Compare and set maximum bound
        "minsd %%xmm2, %%xmm0\n\t"  // Use MINSD for m = min(m, max)
        "movsd %%xmm0, %0\n\t"      // Store final result
        : "=x" (result)             // Output operand
        : "x" (m), "x" (min), "x" (max)  // Input operands
        : "xmm0", "xmm1", "xmm2"    // Clobbered registers
    );
    return result;
}



JNIEXPORT jdouble JNICALL Java_model_library_MyNativeLibrary_myHypot(
    JNIEnv *env, jobject obj, jdouble x, jdouble y) {
    jdouble result;
    __asm__ volatile (
        // Load x and y into XMM registers
        "movsd %1, %%xmm0\n\t"     // Load x into xmm0
        "movsd %2, %%xmm1\n\t"     // Load y into xmm1
        // Square x
        "mulsd %%xmm0, %%xmm0\n\t" // x * x
```

```
        // Square y
        "mulsd %%xmm1, %%xmm1\n\t" // y * y
        // Add squared values
        "addsd %%xmm1, %%xmm0\n\t" // x^2 + y^2
        // Take square root
        "sqrtsd %%xmm0, %%xmm0\n\t" // sqrt(x^2 + y^2)
        // Multiply by 3.0
        "movsd %3, %%xmm1\n\t"     // Load 3.0
        "mulsd %%xmm1, %%xmm0\n\t" // result * 3.0
        // Store result
        "movsd %%xmm0, %0\n\t"
        : "=x" (result)            // Output operand
        : "x" (x), "x" (y), "x" (3.0)  // Input operands
        : "xmm0", "xmm1"           // Clobbered registers
    );
    return result;
}

JNIEXPORT jdouble JNICALL
    Java_model_library_MyNativeLibrary_makeRandomMovement(JNIEnv *env,
    jobject obj, jdouble direction, jdouble frac) {
    jdouble result = 0.0; // Set initial result value to zero

    __asm__ volatile (
        "call rand\n\t"             // Generate the first random number
        "cvtsi2sdq %%rax, %%xmm0\n\t"  // Convert 64-bit integer in RAX
    to double in register
        "divsd %1, %%xmm0\n\t"      // Normalize the first random number
    (between 0 and 1)
        "mulsd %2, %%xmm0\n\t"      // Multiply the random value by
    direction
        "mulsd %3, %%xmm0\n\t"      // Multiply the new value by frac

        "call rand\n\t"             // Generate the second random number
        "cvtsi2sdq %%rax, %%xmm1\n\t"
        "divsd %1, %%xmm1\n\t"      // Normalize the second random number
     (between 0 and 1)
        "mulsd %%xmm1, %%xmm1\n\t"  // Square the second random value
        "mulsd %%xmm1, %%xmm0\n\t"  // Multiply the previous random value
     by the squared new value

        "movsd %%xmm0, %0\n\t"      // Store the final result in result
        : "+x" (result)
        : "x" ((double)RAND_MAX), "x" (direction), "x" (frac)
        : "%xmm0", "%xmm1"
    );

    return result;
}


JNIEXPORT jboolean JNICALL
    Java_model_library_MyNativeLibrary_isKeyPressed(JNIEnv *env, jobject
    obj, jint keyCode) {
    return (GetAsyncKeyState(keyCode) & 0x8000) != 0;
}
```

اتصال کدهای جاوا به کدهای سطح پایین

۱.۵ روش‌های اتصال C و اسمبلی به جاوا

برای برقراری ارتباط بین کدهای C / C++ و جاوا روش‌های متعددی وجود دارد. در اینجا برخی از این روش‌ها را بررسی می‌کنیم. توجه کنید که ما اگر بتوانیم زبان سی را به جاوا متصل کنیم، میتوانیم برنامه سی ای که اسمبلی به آن متصل شده زا بهه جاوا متصل کنیم:

۱.۱.۵ **JNA** $(JavaNativeAccess)$

JNI امکان ارتباط مستقیم بین جاوا و کدهای C / C++ را فراهم می‌کند. نیازی به نوشتن کد JNI ندارد و از Libffi برای مدیریت فراخوانی‌های بومی استفاده می‌کند. این روش ساده‌تر از JNI است اما ممکن است از نظر عملکرد کمی کندتر باشد.

۲.۱.۵ **SWIG** $(SimplifiedWrapperandInterfaceGenerator)$

یک ابزار برای تولید خودکار کدهای wrapper است که به کمک آن می‌توان کدهای C را به زبان‌های مختلف مانند Python، Java، Perl PHP، JavaScript و ... متصل کرد.

۳.۱.۵ **GIWS** $(GenerateInterfaceWithSWIG)$

این روش مبتنی بر XML است و برای تولید خودکار کدهای JNI و C به کار می‌رود. همچنین باعث می‌شود که تعامل با JVM آسان‌تر شود.

۴.۱.۵ **Libffi** $(ForeignFunctionInterfaceLibrary)$

یک bridge عمومی برای ارتباط بین زبان‌های برنامه‌نویسی است. JNA وCPython از Libffi برای برقراری ارتباط با C استفاده می‌کنند.

۵.۱.۵ روش اتصال اسمبلی با `Runtime.exec()`

علاوه بر روش‌های بالا، می‌توان از Runtime.exec() برای اجرای کد اسمبلی در یک پردازش جداگانه استفاده کرد. در این روش:

- از Runtime.getRuntime().exec(path) برای اجرای یک برنامه خارجی (مثلاً فایل اسمبلی کامپایل‌شده) استفاده می‌شود.
- با استفاده از BufferedWriter داده‌ها به این برنامه ارسال می‌شوند.
- خروجی پردازش از طریق BufferedReader خوانده می‌شود.

این روش برای اجرای برنامه‌های اسمبلی مستقل مفید است اما نمی‌تواند مانند JNI توابع اسمبلی را مستقیماً داخل جاوا اجرا کند.

۲.۵ چرا JNI برای این پروژه بهتر است؟

در پروژه‌ی من، هدف **بهینه‌سازی عملکرد** در پردازش‌های سنگین گرافیکی است. JNI نسبت به روش‌های دیگر مانند JNA کارایی بیشتری دارد زیرا:

- **دسترسی مستقیم به حافظه بومی** دارد و سربار JVM را کاهش می‌دهد.
- **عملکرد بهینه‌تر** نسبت به JNA دارد، زیرا نیازی به Libffi یا API Reflection ندارد.

- می‌توان **کد اسمبلی را مستقیماً داخل توابع JNI** پیاده‌سازی کرد، که باعث بهینه‌سازی سطح پایین می‌شود.
- بر خلاف `Runtime.exec()`، نیازی به ایجاد پردازش‌های خارجی ندارد و **overhead سیستم‌عامل** را کاهش می‌دهد.
- بهبود عملکرد و کاهش سربار پردازشی نسبت به روش‌های دیگر مانند JNA.
- امکان اجرای مستقیم کدهای C و Assembly برای بهینه‌سازی محاسبات.
- کنترل بیشتر روی نحوه‌ی ارتباط و مدیریت حافظه نسبت به دیگر روش‌ها.

به همین دلیل، JNI را انتخاب کردم تا بتوانم عملکرد را بهبود ببخشم و مستقیماً از اسمبلی در پردازش‌های حساس به زمان استفاده کنم.

۳.۵ دستورالعمل اتصال با JNI

JNI ($JavaNativeInterface$) یک روش استاندارد برای اتصال کد جاوا به کتابخانه‌های نوشته‌شده به زبان‌های سطح پایین مانند C و Assembly است. در اینجا مراحل ایجاد یک کتابخانه‌ی بومی و استفاده از آن در جاوا را بررسی می‌کنیم.

۱.۳.۵ تعریف متدهای native

برای استفاده از متدهای بومی، ابتدا در کلاس جاوا توابع موردنظر را به‌صورت prototype تعریف می‌کنیم و از کلیدواژه‌ی native استفاده می‌کنیم تا جاوا بداند این متدها در یک کتابخانه‌ی خارجی تعریف شده‌اند. همچنین، باید از System.load(”yourLibraryName”) برای بارگذاری کتابخانه‌ی بومی استفاده کنیم.

```
1    public class MyNativeLibrary {
2        static {
3        System.load("MyNative.dll");
4        }
5        public native int addNumbers(int a, int b);
6    }
```

Listing 1: native methods example in java

۲.۳.۵ ایجاد فایل header مناسب

پس از تعریف متدهای بومی، باید یک فایل header ایجاد کنیم که ساختار متدهای native را مشخص کند. این کار را با استفاده از کامپایلر جاوا و انجام می‌دهیم.:

```
1    javac MyNativeLibrary.java
2    javac -h . MyNativeLibrary.java
```

Listing 2: Generating the `.h` file

این دستورات یک فایل $h.$ تولید می‌کنند که شامل اعلان توابعی است که در کد C پیاده‌سازی خواهند شد.

۳.۳.۵ نوشتن کد C

حالا فایل **h.** و کتابخانه jni.h را در برنامه‌ی C خود include می‌کنیم و توابع موردنیاز را پیاده‌سازی می‌کنیم. برای بهینه‌سازی بیشتر، می‌توان از assembly inline یا فراخوانی فایل اسمبلی درون توابع استفاده کرد.

```c
#include <jni.h>
#include "MyNativeLibrary.h"

JNIEXPORT jint JNICALL Java_MyNativeLibrary_addNumbers(JNIEnv *env, jobject obj, jint a, jint b) {
    return a + b;
}
```

Listing 3: Implementing native methods in C

۴.۳.۵ ایجاد کتابخانه‌ی باینری از کد C

بعد از نوشتن کد C ،باید آن را به یک کتابخانه‌ی بومی کامپایل کنیم. برای این کار، از دستور GCC استفاده می‌شود. از این بخش به بعد، مثال‌ها و توضیحات مربوط به ویندوز ارائه شده‌اند، اما روند کار در لینوکس و سایر سیستم‌عامل‌ها نیز مشابه خواهد بود. در پوشه‌ای که فایل C قرار دارد، کافی است دستور زیر را در cmd اجرا کنیم:

```
gcc -shared -o MyNative.dll MyNative.c -Wl,--out-implib,libMyNative.a -m64 -I"C:\Program Files\Java\jdk-21\include" -I"C:\Program Files\Java\jdk-21\include\win32"
```

Listing 4: Creating the `.dll` file on Windows

توجه کنید که در دستور بالا :"C:\Program Files\Java\jdk-۲۱" به عنوان آدرس محل نصب JDK در ویندوز در نظر گرفته شده است. حالا فایل *dll*. ساخته شده و کافی است آن را در پوشه bin محل نصب JDK کپی کنیم.

برای راحتی بیشتر، می‌توان این مراحل را مستقیماً در cmd با دسترسی Administrator انجام داد:

```
gcc -shared -o "C:\Program Files\Java\jdk-21\bin\MyNative.dll" MyNative.c -Wl,--out-implib,libMyNative.a -m64 \
 -I"C:\Program Files\Java\jdk-21\include" \
 -I"C:\Program Files\Java\jdk-21\include\win32"
```

Listing 5: Creating and copying the library to the correct location

تحلیل عملکرد و بهینه‌سازی

بهینه‌سازی و مقایسه عملکرد توابع جاوا و کتابخانه بومی

همان‌طور که در بخش کدهای جاوا اشاره کردم، یک پوشه اختصاصی برای تست تفاوت بین توابعی که از طریق کتابخانه به جاوا متصل شده‌اند و معادل‌های پیاده‌سازی‌شده آن‌ها در جاوا ایجاد کرده‌ام.

در این پوشه:

- کلاس PreviousLibrary شامل نسخه‌های جاوایی توابع کتابخانه‌ای است که با همان نام اما با پسوند ۰ پیاده‌سازی شده‌اند.
- کلاس TestTime شامل ۹ تابع است که هر یک، یک تابع کتابخانه‌ای را با معادل جاوایی آن مقایسه می‌کند.
- کلاس KeyPressTest برای بررسی زمان پاسخ‌گویی به فشردن کلیدهای کیبورد و مقایسه‌ی دو تابع مرتبط طراحی شده است.

۱.۷ نحوه‌ی اجرای تست عملکرد

در کلاس TestTime، ابتدا **Compilation) (Just-In-Time JIT** را گرم می‌کنیم تا اثر تأخیر اولیه‌ی کامپایلر حذف شود. این کار به این صورت انجام می‌شود:

۱. هر تابع ۱۰۰ بار اجرا می‌شود تا JIT بهینه‌سازی‌های لازم را انجام دهد.

۲. سپس تابع ۱۰۰۰ **بار** با ورودی‌های تصادفی (تولیدشده توسط کلاس Random در جاوا) اجرا شده و میانگین زمان اجرای آن‌ها مقایسه می‌شود.

۲.۷ تأثیر JIT بر عملکرد اجرای کد

جاوا از **Compilation) (Just-In-Time JIT** در **JVM** $(JavaVirtualMachine)$ استفاده می‌کند که روی سرعت اجرای کد تأثیر مستقیم دارد.

۱.۲.۷ چرا JIT اولین اجرای کد را کندتر می‌کند؟

وقتی یک متد جاوا اجرا می‌شود:

۱. در ابتدا بایت‌کد توسط **مفسر** $(Interpreter)$ اجرا می‌شود.

۲. اگر متد چندین بار اجرا شود، **JIT آن را به کد ماشین بهینه‌شده کامپایل می‌کند**.

۳. در اجرای‌های بعدی، نسخه‌ی بهینه‌شده اجرا شده و عملکرد به‌شدت بهبود می‌یابد.

نتیجه:

- اولین اجرای کد کندتر است، چون هنوز بهینه‌سازی انجام نشده است.
- بعد از چندین بار اجرا، متد **سریع‌تر و کارآمدتر** اجرا می‌شود.
- اگر یک متد فقط یک بار اجرا شود و زمان آن اندازه‌گیری شود، ممکن است نتایج غیرواقعی به دست آید.

۳.۷ چگونه تأثیر JIT را کاهش دهیم؟

برای به‌دست آوردن **نتایج دقیق‌تر**، قبل از اندازه‌گیری زمان اجرا، چندین بار متد را اجرا می‌کنیم .(warm-up) مثلا:

```
for (int i = 0; i < 100; i++) {
    method();
}
```

سپس زمان اجرای واقعی را محاسبه می‌کنیم. این کار باعث می‌شود که JIT **کد را بهینه کند** و **عملکرد واقعی تابع را مشاهده کنیم**.

۴.۷ نتیجه‌ی نهایی

پس از اجرای این فرآیند، خروجی نهایی **میزان بهره‌وری و تفاوت عملکردی توابع کتابخانه‌ای و معادل‌های جاوایی آن‌ها** را نشان خواهد داد:

```
----------Check ABS----------
JAVA Avg: 246 ns
C+ASM Avg: 122 ns
Relative improvement: 0.50
----------Check IS Between----------
JAVA Avg: 234 ns
C+ASM Avg: 191 ns
Relative improvement: 0.18
----------Check Make Random Movement----------
JAVA Avg: 720 ns
C+ASM Avg: 153 ns
Relative improvement: 0.79
----------Check Make In Bound----------
JAVA Avg: 227 ns
C+ASM Avg: 123 ns
Relative improvement: 0.46
----------Check My Hypot----------
JAVA Avg: 467 ns
C+ASM Avg: 117 ns
Relative improvement: 0.75
----------Check Calculate Parabola----------
JAVA Avg: 482 ns
C+ASM Avg: 336 ns
Relative improvement: 0.30
----------Check Trail Values----------
JAVA Avg: 188 ns
C+ASM Avg: 165 ns
Relative improvement: 0.14
----------Check Sin----------
JAVA Avg: 252 ns
C+ASM Avg: 205 ns
Relative improvement: 0.19
----------Check Div Round Away From Zero----------
JAVA Avg: 221 ns
C+ASM Avg: 127 ns
Relative improvement: 0.42
----------Check Key Pressed----------
JAVA Avg: 30839011 ns
C+ASM Avg: 15973933 ns
Relative improvement: 0.48
```

این توابع در کلاس‌های اصلی کد ما قرار دارند و در هر فریم تصویر چندین بار فراخوانی می‌شوند. بنابراین، بهبودی که در زمان اجرا مشاهده می‌کنید، در مقیاس کلی تأثیری شگفت‌انگیز بر عملکرد پروژه‌ی جاوا خواهد داشت. حتی در برخی موارد، این تفاوت آن‌قدر محسوس است که با

چشم غیرمسلح هم کاملاً قابل مشاهده است!

به‌طور کلی، ترکیب زبان‌های C، اسمبلی و جاوا نتیجه‌ای رق‌العاده به همراه داشته است.