digital whisper Technologic Papers

המעשה המופלא בקבוע המסתורי 0x5f3759df

מאת דר' גדי אלכסנדרוביץ'

הקדמה

למתמטיקאים יש את סיפורי המסתורין שלהם. המפורסם מביניהם הוא ככל הנראה הערה ששרבט פייר דה פרמה בשולי ספר ה"אריתמטיקה" של דיופנטוס שלו, שבה העיר שהכללה של טענה שהופיעה בספר היא שגויה תמיד וש"בידי הוכחה מופלאה למשפט אך שולי ספר זה צרים מלהכילה". הערת השוליים הזו לא פורסמה על ידי פרמה בימי חייו והיא התגלתה רק כשנקראו הספרים שבעזבונו, ואז היה קצת מאוחר מדי לשאול את פרמה לאיזו הוכחה הוא התכוון. שום הוכחה דומה לא נמצאה בכתביו או התכתבויותיו, ובמשך למעלה מ-350 שנה המתמטיקאים ניסו להוכיח את המשפט שלו ללא הצלחה. גם כשנמצאה הוכחה, היא הייתה מודרנית ומורכבת ובוודאי לא "ההוכחה הנפלאה" של פרמה. מה הייתה ההוכחה המקורית? איך פרמה הגיע אליה? מתי ואיך הבין שאינה נכונה, אם בכלל? תעלומה.

במדעי המחשב אין לנו תעלומות בנות מאות שנים - מדעי המחשב הם תחום צעיר יחסית. אבל היום אני רוצה לספר על תעלומה בת למעלה מעשור, שגם היא כנראה שלא תיפתר לעולם אבל היא מעניינת מספיק גם ככה - תעלומת המספר 0x5f3759df וקטע הקוד שבו הוא מופיע. קטע הקוד הזה נמצא, מכל המקומות בעולם, בקוד של משחק היריות מגוף ראשון Quake 3. הוא נתגלה בשנת 2005, כשקוד המשחק שוחרר לציבור הרחב. אפשר למצוא אותו כאן, והוא נראה ככה:

```
float Q_rsqrt( float number )
{
        long i;
        float x2, y;
        const float threehalfs = 1.5F;

        x2 = number * 0.5F;
        y  = number;
        i  = * ( long * ) &y;                       // evil floating point bit level hacking
        i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
        y  = * ( float * ) &i;
        y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

אך לפני שעבור על הקוד עצמו, נתחיל עם קצת היסטוריה.

פרק ראשון (שבו המספר נזכר בערגה בראשית דרכו כגיימר ואנחנו לומדים איך לוחם בנאצים שינה את עולם משחקי המחשב לנצח)

בואו נעבור לרגע לתחילת שנות התשעים. עולם המחשבים האישיים קיים כבר עשור או שניים, אבל עדיין מגשש את דרכו בזהירות. אז כמו עכשיו, היבט חשוב ביותר של משחקי המחשב הוא **הגרפיקה** שלהם - כמה טוב הם נראים. גרפיקה זה עניין מסובך. לא מספיק לדעת לצייר יפה, צריך גם לוודא שהמחשב יודע להציג את הציורים היפים **מהר** .כשמדובר על משחקי פעולה, זה קריטי לחלוטין שהמשחק ירוץ חלק ורציף תוך כדי שהוא נראה טוב. העבודה האמיתית כרגע נעשית מאחורי הקלעים: המתכנתים שצריכים לכתוב את המנוע של המשחק - הקוד שגורם למשחק לפעול, ובפרט הקוד שמאפשר את הצגת הגרפיקה - משתמשים בכל תעלול תכנות אפשרי כדי לגרד עוד קצת מהירות. הכל יחסית חלוצי. עדיין אין יותר מדי קוד קיים להתבסס עליו; אין נסיון מצטבר של עשרות שנים; אין מנועים קיימים בשוק שאפשר פשוט להשתמש בהם. בשנת 1991 מצטרפת לעולם הזה חברה חדשה - id Software. סדרת המשחקים הראשונה שהם מוציאים נקראת Commander Keen ועוסקת בהרפתקאותיו של ילד בן שמונה עם קסדת פוטבול ומקל פוגו ומלחמתו בחייזרים שמנסים להשמיד את כדור הארץ. ככה בערך זה נראה:

קין מתרחש בעולם דו-ממדי שבו אפשר לנוע ימינה, שמאלה, למעלה ולמטה, כשאנחנו מסתכלים על העניינים מהצד. למשטחים שעליהם הדמויות במשחק עומדים קוראים **פלטפורמות** ועל שמם משחקים כאלו נקראים **משחקי פלטפורמות**. אל תזלזלו במה שאתם רואים כאן. לזמנו הגרפיקה של המשחקים הללו הייתה טובה למדי (הסגנון הקרטוני הוא מכוון) והם אפילו היו חדשניים בתור משחקי פלטפורמות בכך שהתנועה בהם הייתה "חלקה" - דהיינו, במקום שהדמות תצא מהמסך שבו היא נמצאת ויעלה מסך אחר, המסך זז באופן רציף יחד עם הדמות של קין. לעשות את זה בזמנו על מחשב אישי (להבדיל

מקונסולה כמו נינטנדו) לא היה טריוויאלי, והאחראי לתעלולי התכנות שאיפשרו את זה היה המתכנת הראשי של id software, ג'ון קרמק.

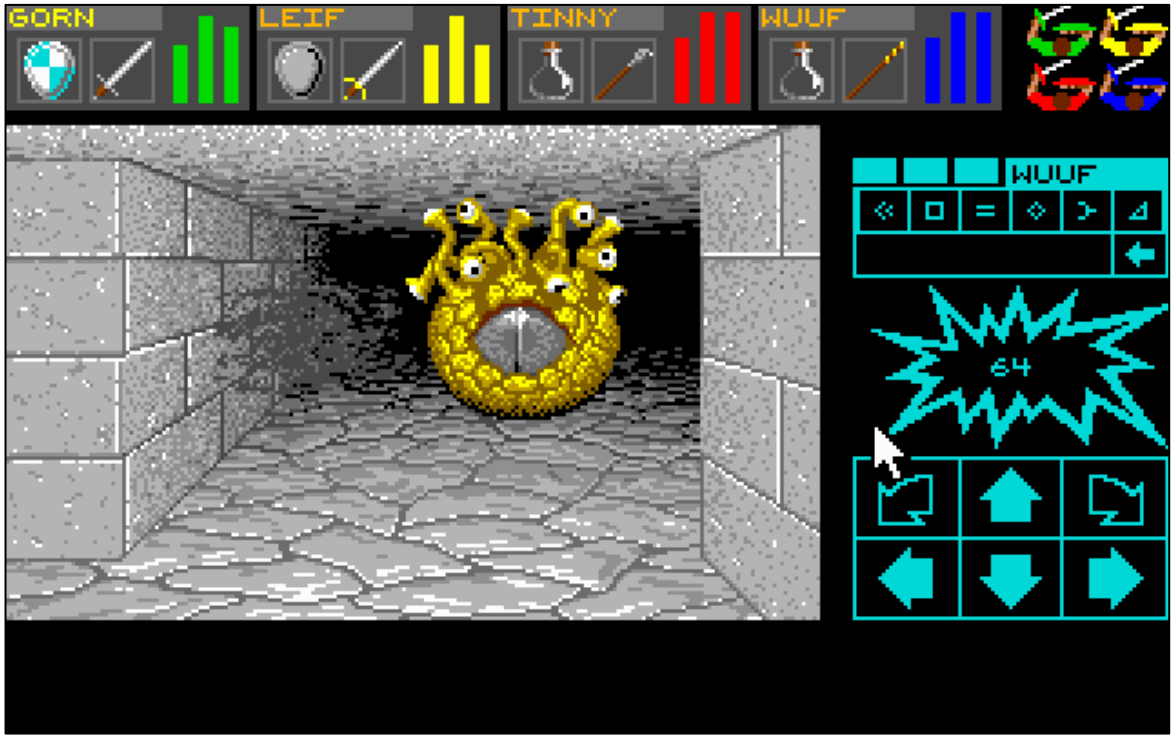
אחרי המנוע של קין קרמק עבר להתעסק עם אתגר אחר - מנוע גרפי תלת ממדי. במקום שהעולם יוצג מהצד, הוא מוצג מנקודת המבט של הדמות שאותה משחקים. המשחק המפורסם ביותר שהוציאה החברה עם המנוע הראשון שיצר קרמק נקרא Wolfenstein 3D. ככה זה נראה:

הגרפיקה פה שונה מהותית מזו של קומנדר קין. אצל קומנדר קין, הכל מצוייר ביד והמשחק פשוט מציג את הציורים הללו. לעומת זאת בוולפנשטיין התמונה שהשחקן רואה נוצרת בידי המחשב תוך כדי משחק: מישהו צייר את הטקסטורה של קיר ואפשר לראות שחוזרים עליה שוב ושוב, אבל אותה הטקסטורה מצויירת באופן **קצת שונה** בהתאם לקיר שרואים. אם הקיר רחוק יותר, רואים אותו קטן יותר; אם רואים אותו מהצד, הקיר מוצג בצורה אלכסונית. יש תאורה ויש הצללה (לכל הפחות, הקירות לפעמים בהירים ולפעמים כהים), וכדומה. במילים אחרות, המחשב לוקח תמונה של "איך קיר נראה" ומחשב איך בדיוק הקירות אמורים להיות מוצגים בהתבסס על המיקום הנוכחי של השחקן ושאר הפרטים שבזירה.

שימו לב שהמשחק עצמו הוא **דו-ממדי**: הדמות של השחקן יכולה לנוע רק ימינה, שמאלה, קדימה ואחורה (וכמובן, באלכסון שהוא שילוב של שניים מאלו). אין במשחק הזה אפשרות ללכת "למעלה" ו"למטה" בכלל. החשיבות היא בנקודת המבט של השחקן, לא במספר כיווני התנועה שלו. למשחק מסוג זה קוראים "משחק פעולה מגוף ראשון". ומה עושים עם כל הגרפיקה הזו? ובכן, חסכתי את זה מכם בצילום המסך,

אבל הרעיון במשחק (שמבוסס על משחקים משנות השמונים, נטולי גרפיקה תלת מימדית) הוא לשחק חייל אמריקאי יהודי שפולש לכל מני מעוזים נאציים במלחמת העולם השניה ומחסל את יושביהם, כולל היטלר עצמו מתישהו.

כמו עם קין, כך גם עם Wolf3D, המנוע שקרמק יצר בשביל המשחק חולל מהפכה זוטא. הגרפיקה שלו נראתה **טוב** מצד אחד, אבל מצד שני היא נוצרה מספיק מהר כדי שהמשחק ירוץ חלק, כפי שנדרש ממשחק יריות מהיר שכזה. השילוב של שני אלו היה מהפכה של ממש. בואו נראה דוגמאות למשחקים ישנים יותר כדי להבין מה השתנה. ראשית, הנה צילום מסך ממשחק מבוכים בשם Dungeon Master מ-1987:



כאן התמונה נראית תלת ממדית, אבל זו "רמאות" - מישהו צייר ביד את הכל - גם ציור של "קיר קרוב" וגם ציור של "קיר רחוק" וגם ציור של "קיר מהצד" וכדומה. המחיר של זה הוא שאי אפשר לנוע באופן חופשי - הדמות שאותה משחקים יכולה לבצע סיבובים של 90 מעלות ולנוע קדימה ואחורה משבצות שלמות בכל פעם וזהו. ב-Wolf3D התנועה היא חופשית וההרגשה של המשחק היא שונה לגמרי (הרבה יותר מתאימה למשחק יריות).

והנה צילום מסך ממשחק הרפתקאות בשם Castle Master מ-1990:

כאן הגרפיקה היא תלת ממדית לגמרי. יש גם "למעלה" ו"למטה" ואפשר להסתכל אליהם ואפילו סוג של ללכת אליהם (למשל, אפשר ליפול). המחיר הוא שהגרפיקה הזו נראית **ממש לא משהו** והקצב של המשחק איטי (ה"קרבות" כוללים יצורים שעומדים או זזים בצורה לא רציפה ומנסים לפגוע בהם בלי שיש לתזוזה של השחקן שום ערך מוסף). המשחק עצמו די מהנה ומבוסס בעיקרו על פאזלים ועל שיטוט וחיפוש של דברים, אבל **זה לא משחק פעולה**.

מה שאני רוצה לומר לכם בסיפור הארוך הזה הוא כמה דברים שלטעמי הם קריטיים כדי להעריך את קטע הקוד המוזר שלעיל:

- גרפיקה היא דבר חשוב ביותר במשחקי מחשב.
- כשמדובר על משחקי פעולה תלת ממדיים אי אפשר להתפשר לא על איכות הגרפיקה ולא על מהירות המשחק. חייבים להיות יצירתיים ולהשיג את שניהם.
- בזמנו הדרך להשיג את הדברים הללו הייתה על ידי התחכמויות ברמת הקוד.
- ג'ון קרמק היה חתיכת פורץ דרך רציני למרות שבקושי מכירים את השם שלו מחוץ לחוגים הרלוונטיים.

במקרה הספציפי של Wolf3D ההשקעה השתלמה. היה כאן שילוב של המנוע הגרפי, העיצוב הסגנוני של המשחק והאופן החכם שבו הוא הופץ (הפצה חינמית של החלק הראשון שלו, מודל שעבד לא רע גם עם קומנדר קין) והמשחק היה הצלחה גדולה. id software ראתה כי טוב והמשיכה בכיוון של משחקי יריות מגוף ראשון. ג'ון קרמק יצר מנוע תלת ממדי חדש ומתוחכם בהרבה מזה של Wolf3D, ועל בסיסו עוצב אחד ממשחקי המחשב החשובים ביותר בהיסטוריה .Doom - בבסיסו, דום הוא כמו Wolf3Dרק עם שדים

במקום נאצים והגיהנום במקום טירה. לאחר ההצלחה הגדולה של דום (וההמשך שלו) עברה החברה לסדרה חדשה של משחקי יריות מתלת מימד .Quake - שבהם העלילה היא... אה... טוב, למי אכפת בכלל. ב-1999 יצא Quake 3 שבו כל הקונספט הזה של עלילה די נזנח לטובת קרבות מרובי משתתפים. בשלב הזה הגרפיקה כבר נראתה הרבה, הרבה יותר טוב והייתה תלת מימדית באופן מלא:

מה השתנה בשנים שחלפו שאיפשר גרפיקה יותר טובה? ראשית, המחשבים היו יותר חזקים. שנית, הם התחילו להשתמש ברכיבי חומרה ייעודיים להצגת גרפיקה (מה שנקרא בשעתו "מאיץ גרפי"). אבל תכנות חכם של המנוע עדיין היה אספקט קריטי, והמנוע שמאחורי Quake 3 היה מוצלח מאוד. לכן כשג'ון קרמק הודיע בשנת 2005 שקוד המקור המלא של המנוע ישוחרר לרשת לטובת כל מי שבא לו לקרוא אותו (בינתיים כבר פותח המנוע הבא בתור) הייתה שמחה גדולה. ואנשים רצו לקרוא את הקוד. ואז התגלה בו קטע הקוד הקצרצר שבו אנחנו עוסקים כאן.

עוד מעט אסביר מה בדיוק הקוד הזה עושה, אבל בקצרה: הוא עוזר, בצורה חכמה מאוד, לעשות גרפיקה יפה ומהירה וכבר הסכמנו שזה חשוב. השאלה המעניינת יותר היא **מי** כתב אותו, ומתי. מן הסתם החשוד המיידי היה ג'ון קרמק עצמו, אבל כששאלו אותו הוא אמר בפשטות שלא, זה לא הוא, אולי זה הברנש האחר ההוא... אבל גם הברנש האחר ההוא הכחיש כל קשר. אפשר לקרוא עוד על החיפוש כאן. השורה התחתונה - לאף אחד אין מושג מי כתב את הקוד הזה. כנראה שהוא עתיק בהרבה מאשר המנוע של Quake 3 ולא ברור איך בדיוק התגלגל לשם. פשוט תעלומה. זה לא לגמרי מפתיע - ככה זה עם קוד מחשב רציני, יש דברים ש"מתגלגלים" פנימה בלי שלאף אחד יהיה מושג אחר כך מה הולך כאן. אבל הקוד הזה הוא דוגמה יפה במיוחד לכך. בעיקר כי הוא עושה את מה שהוא אמור לעשות בצורה יעילה עד להפתיע. זה קצת מזכיר את הסיפור של הסנדלר הכושל שגמדים באו בלילה ועשו את העבודה שלו בשבילו, ובצורה טובה בהרבה.

פרק שני (שבו אנחנו לומדים לקרוא קוד שנראה כמו ג'יבריש ומבינים הכל אבל לא מבינים שום דבר)

לפני שנתחיל לצלול לקוד, בואו נבהיר מה הוא עושה: זו פונקציה שלוקחת מספר x ומחשבת את $\frac{1}{\sqrt{x}}$, כלומר את ההופכי של השורש של x. זה הכל. למה זה חשוב לגרפיקה? אסביר זאת בהמשך, אבל בשורת מחץ אחת: כי ככה מנרמלים וקטורים. שאלה אחרת היא למה לעשות את זה ככה ולא לבנות כמו בני אדם שפויים פונקציה שלוקחת את x ומחשבת את $\sqrt{x}$ ואחר כך אפשר לעשות פעולת חילוק רגילה ולחשב את $\frac{1}{\sqrt{x}}$ כמו בני תרבות. התשובה היא **יעילות**. יעילות היא מילת המפתח בכל מה שאנחנו עושים פה. פעולת חילוק היא בדרך כלל פעולה **יקרה לביצוע** יחסית; אם אפשר להימנע ממנה, למה לא. להבדיל, פעולת כפל היא פחות יקרה, אז אם יש לנו פונקציה יעילה מאוד שמחשבת את $\frac{1}{\sqrt{x}}$ יחסית קל לחשב את $\sqrt{x}$: פשוט מחשבים את המכפלה $x \cdot \frac{1}{\sqrt{x}}$. המחיר של קודם כל לחשב ביעילות את $\frac{1}{\sqrt{x}}$, ואז לבצע את ההכפלה יהיה זול יותר מאשר המחיר של קודם לחשב את $\sqrt{x}$ ואז לחשב את המנה $\frac{1}{\sqrt{x}}$.

קחו מבט נוסף על הקוד, עכשיו כשאתם יודעים מה הוא אמור לעשות. האם אתם מרגישים קצת מוזר? אני מרגיש **מאוד** מוזר. חישוב שורש... זה משהו שאמור להיות מסובך, לא? איך אפשר שקוד יבצע גם חישוב שורש וגם הופכי שלו ביחד בכל כך מעט שורות קוד, ויעשה את זה מהיר ומדויק? משהו פה מרגיש כאילו הוא לא מסתדר. אבל הכל מסתדר - זה עובד, וזה עובד מאוד יפה.

```
552  float Q_rsqrt( float number )
553  {
554          long i;
555          float x2, y;
556          const float threehalfs = 1.5F;
557
558          x2 = number * 0.5F;
559          y  = number;
560          i  = * ( long * ) &y;                                    // evil floating point bit level hacking
561          i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
562          y  = * ( float * ) &i;
563          y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
564  //      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
```

בואו נסביר את הקוד שורה שורה, עבור מי שלא מכיר שפות תכנות. אין כאן שום דבר שמעבר ליכולת ההבנה שלכם - זה קוד מאוד פשוט. רק טיפה טרמינולוגיה קודם: כשאני מדבר על "מספר ממשי" אני מתכוון לכל מספר שאנחנו יודעים לכתוב עם ייצוג עשרוני, למשל 3 או 3.141 או 0.333 וכדומה. ליתר דיוק, אני מתכוון רק לאלו מתוכם שאנחנו יודעים לייצג במחשב, אבל מי אלו בדיוק נראה רק בהמשך. באופן דומה, "מספר שלם" הוא מספר שאין לו כלום אחרי הנקודה העשרונית. 3 הוא שלם ו-3.1 או 0.3 הם לא שלמים. גם על השלמים יש הגבלה, שלא אתאר כרגע, לגבי מי מהם יכול להיות מיוצג במחשב.

```
float Q_rsqrt( float number )
```

השורה הראשונה הזו אומרת "שלום בוקר טוב אני פונקציה ושמי הוא Q_rsqrt (אני מנחש ש-rsqrt זה קיצור של reciprocal square root - ההופכי של שורש ריבועי), אני מקבלת קלט בשם number שהוא מספר ממשי ומחזירה פלט שגם הוא מספר ממשי". מה שאולי לא ברור לכם הוא למה משתמשים במילה float כדי לתאר מספר ממשי; הסיבה לכך היא שבשפת C, מספרים ממשיים מיוצגים על ידי שיטת ייצוג שנקראת **נקודה צפה** ואתאר בהמשך המאמר. אתם לא באמת צריכים להבין אותה בשלב הזה.

שלוש השורות הבאות מגדירות משתנים וקבועים שבהם ישתמשו בהמשך הפונקציה:

```
long i;
float x2, y;
const float threehalfs = 1.5F;
```

המשתנים x2 ו-y שניהם מספרים ממשיים. לעומת זאת i הוא **מספר שלם**. זה חשוב כי מספרים שלמים מיוצגים בצורה שונה מאשר מספרים ממשיים כלליים. המילה long נובעת מכך שיש שיטות שונות לייצג מספרים שלמים ב-C שנבדלות בגודל המקסימלי של המספרים שאפשר לייצג. שם מקובל למספר שלם הוא int, קיצור של Integer; השם long בא לומר שהמספר השלם הולך להיות גדול יחסית - לכל הפחות בתחום מספרים סביב 0 שגודלו 2^{32}, ואולי גם יותר (לא ניכנס פה לדקויות של הגדרות טיפוסים ב-C, זו זוועה שאין כמוה).

השורה האחרונה מגדירה **קבוע**: משתנה שערכו נקבע מראש ולא ישתנה אחר כך. במקרה הנוכחי, threehalfs מוגדר להיות בדיוק מה ששמו מרמז: המספר 1.5 כאשר הייצוג שלו הוא על ידי float (זה ה-F שבסוף). למה צריך את הקבוע הזה? בהמשך, כשנראה את החישובים שעומדים מאחורי הפונקציה הזו, נראה שהוא אכן צץ מעצמו.

שתי השורות הבאות מאתחלות את המשתנים שהוגדרו קודם:

```
x2 = number * 0.5F;
y = number;
```

כלומר y, הוא כרגע בדיוק המספר שקיבלנו בתור קלט, ו-$x2$ הוא חצי ממנו. למה צריך את זה? נראה אחר כך.

שלוש השורות הבאות הן ללא ספק החלק הכי לא ברור בכל הקוד:

```
i = * ( long * ) &y; // evil floating point bit level hacking
y = * ( float * ) &i;
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

ראשית, הטקסט האנגלי שמופיע אחרי זוג הלוכסנים בסוף שתי השורות הראשונות הוא **הערה**, כלומר משהו שלא רץ בפועל אלא קיים שם למען הדורות הבאים שיקראו את הקוד. אני מנחש שמי שהוסיף את ההערות הללו לא היה המתכנת המקורי אלא מישהו שניסה להבין מה בעצם הוא עשה שם, וכפי שניתן לראות, השורה האמצעית די בלבלה אותו... כל שלוש השורות הללו הן לחלוטין בלתי קריאות למי שלא

מכיר C, אבל קל להסביר את ה"בערך" של מה שהן עושות: השורה הראשונה אומרת "קח את המספר הממשי y ותתייחס אליו לרגע בתור מספר שלם, ואת זה תציב ב-i". השורה האחרונה אומרת "קח את המספר השלם i ותתייחס אליו לרגע בתור מספר ממשי ואת זה תציב ב-y". מפתה לומר שמתבצעת פה **המרה** ממספר ממשי למספר שלם, וההפך. אבל זה **ממש לא** מה שקורה פה. המרה היא תהליך מתוחכם שבו מתבצעת מניפולציה על המספר, למשל 3.73.7 יומר ל- 33 כאשר מבצעים המרה. לא. מה שקורה פה הוא יותר מוזר: אנחנו לוקחים את **האופן שבו המספר הממשי מיוצג במחשב** ומתייחסים לדבר הזה בתור **ייצוג במחשב של מספר שלם**. זה תעלול מוזר מאוד כי שיטות הייצוג של שני סוגי המספרים הללו הן **שונות בתכלית**. אפרט על זה בהמשך.

ואז מגיעה השורה האמצעית. דווקא אותה די קל להבין, אבל צריך להכיר את הסימונים. ראשית, הקבוע המסתורי 0x5f3759df. הקבוע הזה הוא בסך הכל דרך ייצוג מקובלת למספר השלם 1597463007, כאשר כותבים אותו ב**בסיס הקסדצימלי**, כלומר בסיס ספירה שבו יש לנו 16 ספרות. ה-0x בהתחלה הוא האופן הסטנדרטי שבו מודיעים לשפת C "הנה עכשיו אני מביא לך מספר בבסיס 16 ולא בבסיס 10 כמו בדרך כלל" וה-d,f הללו שנמצאים שם הם פשוט הספרות עבור 13 ו-15.

קצת יותר מסתורי ה-i << 1 הזה. אני אסביר בהמשך למה בדיוק משתמשים בסימון הזה, אבל המשמעות שלו פשוטה - זו חלוקה ב-2. אם כן, כל מה שהשורה הקסומה הזו עושה הוא לקחת את הקבוע 0x5f3759df ולהפחית ממנו את "הקלט של הפונקציה שלנו כאשר הוא מתפרש איכשהו בתור מספר שלם ומחולק ב-2".

למה? למה עושים דבר מוזר כזה? בשביל מה?

התשובה היא שהשורות הללו נותנות לנו **קירוב** לערך של $\frac{1}{\sqrt{x}}$. הקירוב הזה רחוק מלהיות מושלם, אבל הוא טוב בצורה מפתיעה. כדי לשפר את הקירוב הזה עוד יותר מגיעות השורות האחרונות בקוד:

```
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can
be removed
```

השורות הללו מבצעות שתיהן בדיוק את אותו חישוב: $y \leftarrow \frac{3}{2} - x_2 y^2$. החישוב הזה הוא מימוש למקרה הספציפי שלנו של שיטת קירוב שנקראת **שיטת ניוטון-רפסון** ואתאר בהמשך. הרעיון בשיטת ניוטון-רפסון הוא שזו שיטה איטרטיבית: כשרוצים לחשב איתה משהו, מתחילים עם קירוב כלשהו שלו, ואז מפעילים על הקירוב הזה חישוב שמשפר אותו, שוב, ושוב, ושוב. אחרי כל הפעלה של ניוטון-רפסון הקירוב שלנו משתפר עד שבסוף הוא "קרוב מספיק לצרכים שלנו" ואפשר להפסיק. השיטה הזו פועלת די מהר - על פי רוב לא צריך יותר משלוש-ארבע איטרציות שלה כדי להגיע לקירוב מצויין, אבל הקוד הנוכחי שלנו שאפתני יותר - הוא טוען ש**שתי** איטרציות יספיקו. רגע, לא, הוא טוען אפילו יותר מזה! הוא טוען שאיטרציה **אחת** תספיק! השורה השניה, אם תסתכלו טוב, כולה הערה: היא מתחילה בשני לוכסנים. כנראה שמה שקרה הוא שבמקור השורה השניה הייתה חלק מהקוד שרץ בפועל, ומתישהו המתכנת

הרלוונטי אמר "אוקיי בואו נסיר אותה ונראה אם משהו מעניין השתנה" והתוצאה הייתה שמצד אחד הקוד רץ מהר יותר ומצד שני לא נראה שום נזק בעל חשיבות, ולכן הוחלט לוותר על השורה השניה לגמרי.

זה אומר שעיקר העבודה של הפונקציה מתבצעת בשלוש השורות שראינו קודם, של "הקירוב ההתחלתי". איכשהו מתבצע שם קסם שכזה שאחריו מספיקה הפעלה בודדת של ניוטון-רפסון כדי שכל העסק יעבוד טוב.

את שני החלקים הללו של הקוד אפשר להבין באופן בלתי תלוי זה בזה. לכן אתחיל דווקא מהתיאור של ניוטון-רפסון, שהיא שיטה פשוטה יחסית, ואחר כך אעבור לדבר על הטירוף של שורות הקירוב ההתחלתי.

פרק שלישי (ובו סקירה מהירה של השיטה המהירה של ניוטון-רפסון)

ניוטון-רפסון היא שיטת **קירוב**. אנחנו רוצים לחשב שורש של מספר כלשהו, למשל $\sqrt{2}$? בשיטת הייצוג העשרוני הרגילה שלנו יש למספר הזה אינסוף ספרות אחרי הנקודה, והן לא מחזוריות. אז נצטרך להפסיק מתישהו. נאמר, אחרי ארבע ספרות זה מספיק לנו? במקרה הזה כל מספר שמתחיל ב-1.4142 יהיה מספיק טוב לנו. מה שניוטון-רפסון עושה הוא לקחת קירוב התחלתי למספר שאנחנו רוצים לחשב, ואז לשפר את הקירוב הזה שוב, ושוב, ושוב. בכל שיפור אנחנו מרוויחים כמה ספרות מדויקות חדשות אחרי הנקודה. כשאנחנו רואים ש"התקבעו" לנו מספיק ספרות, אנחנו עוצרים.

איך הקסם הזה קורה? מעשית, ניוטון-רפסון מנוסח כך: יש לנו פונקציה $f: R \to R$ ואנחנו רוצים למצוא x כך ש-$f(x) = 0$ מה שנקרא, למצוא נקודת חיתוך של f **עם ציר x** למשל, עבור $\sqrt{2}$ אנחנו נסתכל על הפונקציה $f(x) = x^2 - 2$ שאותה קל לנו לחשב באמצעות פעולות בסיסיות בלבד (להבדיל נאמר מהפונקציה $f(x) = x - \sqrt{2}$ שגם אצלה נקודת החיתוך היא ב- $x = \sqrt{2}$ אבל אנחנו לא יודעים איך לחשב אותה). הכלי שבאמצעותו אנחנו ניגשים לבעיה הזו הוא **הנגזרת** של f. הרעיון האינטואיטיבי של נגזרת הוא שהיא מאפשרת לנו לקרב את f בכל נקודה על ידי קו ישר - מה שנקרא **קירוב לינארי**. כלומר, למצוא קו ישר ש"באופן מקומי" מתנהג כמו f. ההנחה של ניוטון היא שאם f. היא נחמדה מספיק ולא משתוללת, ואם אנחנו כבר עכשיו די קרובים לנקודת החיתוך של f עם ציר x, אז נקודת החיתוך **של הקירוב הלינארי** של f עם ציר x תהיה אפילו עוד יותר קרובה לנקודת החיתוך האמיתית מאשר המיקום הנוכחי שלנו.

מבחינה חישובית קל מאוד להגיע לנוסחה המדויקת של השיטה - כל כך קל, שאפשר להראות את זה כבר בתיכון לתלמידים שלמדו חדו"א וגאומטריה אנליטית, ואף פעם לא הבנתי למה לא לעשות את זה. הרעיון הוא כזה: נניח שאנחנו כרגע בנקודה xnxn, ואנחנו רוצים למצוא קירוב טוב יותר, x_{n+1}. הנגזרת בנקודה x_n היא הערך הפונקציה $f'(x_n)$. המספר הזה הוא **השיפוע** של הקו הישר שמקרב את f בנקודה x_n. עכשיו, בגאומטריה אנליטית אנחנו לומדים איך למצוא את השיפוע של הקו הישר שעובר דרך שתי נקודות נתונות. אם הנקודות הן (x_1, y_1) ו-(x_2, y_2) אז השיפוע הוא $\frac{y_2 - y_1}{x_2 - x_1}$ (אלא אם $x_1 = x_2$

ואז הסיפור קצת יותר מסובך). עכשיו, במקרה שלנו אנחנו יודעים על אחת משתי הנקודות - הנקודה $(x_n, f(x_n))$ שבה אנו מחשבים את הקירוב הלינארי. הנקודה השניה שמעניינת אותנו היא נקודת החיתוך של הישר עם ציר x , ומה שאנחנו מחפשים הוא את קואורדינטת ה-x שלה, מה שאני קורא לו x_{n+1} . כלומר, הנקודה השניה היא $(x_{n+1}, 0)$. נציב את שתי הנקודות הללו ואת הערך של השיפוע במשוואה שתיארתי קודם, ונקבל:

$$\frac{f(x_n) - 0}{x_n - x_{n-1}} = f'(x_n)$$

כלומר, לאחר העברת אגפים נקבל:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

למשל, בדוגמא של $f(x) = x^2$ שלנו נקבל ש-$f'(x) = 2x$ ולכן הנוסחה שניוטון-רפסון נותן לנו היא:

$$x_{n+1} = x_n - \frac{x_n^2 - 2}{2x_n}$$

ואם אנחנו רוצים למצוא קירוב ל-$\sqrt{a}$ עבור a כללי, הנוסחה תהיה:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{z_n}\right)$$

במילים אחרות, ניוטון-רפסון אומר לנו במקרה הזה "כדאי לכם להסתכל על **הממוצע החשבוני** בין הקירוב הנוכחי שלכם לבין המספר ש'משלים אותו' על ידי כפל ל-a". תדמיינו שהמטרה שלנו היא למצוא ריבוע ששטחו a. אנחנו מתחילים עם מלבן, ואז לוקחים את הממוצע בין אורכי הצלע הקצרה והארוכה, ובונים מלבן חדש שהמספר שקיבלנו הוא אורך אחת מצלעותיו ואת השניה אנחנו בונים כדי שהשטח יהיה שוב פעם a. סדרת המלבנים שלנו תלך ותתקרב לריבוע.

בשביל השיטה הזו למציאת שורש אין צורך בניוטון - היא ככל הנראה הייתה ידועה כבר לבבלים ונמצאת בכתביו של המתמטיקאי הרון מאלכסנדריה. אבל זה נחמד מאוד שהיא מתקבלת מניוטון בתור מקרה פרטי פשוט.

עכשיו, משהבנו בערך מה הולך פה, בואו ניישם את ניוטון עבור המקרה שלנו: אנחנו רוצים לחשב לא את $\sqrt{a}$ אלא את $\frac{1}{\sqrt{a}}$, שהוא קצת יותר מסובך. במקרה הזה, נבחר בתור הפונקציה שלנו את:

$f(x) = \frac{1}{x^2} - a$ ונקבל , $f'(x) = -\frac{2}{x^3}$

לכן:

$$x_{n+1} = x_n + \frac{x_n^2}{2}\left(\frac{1}{x_n^2} - a\right) = x_n + \frac{x_n}{2} - \frac{x_n^3}{2}a = x_n\left(\frac{3}{2} - \frac{a}{2}x_n^2\right)$$

האם הנוסחה האחרונה נראית לכם מוכרת? בואו נסתכל שוב בשורות הרלוונטיות בקוד:

```
const float threehalfs = 1.5F;
x2 = number * 0.5F;
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
```

השורה האחרונה פה היא **בדיוק** הנוסחה שהגענו אליה כרגע. עד לרמת ה-3232 שנכתב במפורש בקוד והשימוש ב-$x2$ בתור $\frac{a}{2}$. ומה עם y? כזכור, הערך שלו הוא הקירוב ההתחלתי שמחושב בצורה מתוחכמת למדי קודם. זה הצעד הבא שנצטרך להבין; אני חושב שאת החלק של הניוטון-רפסון אנחנו מבינים מושלם עכשיו. אנחנו מוכנים לפרק הבא!

פרק רביעי (ובו ביטים עושים דברים)

בגדול, אפשר לומר שחלק נכבד מהיקום כולל דברים שמורכבים מדברים. יצירות לגו מפוארות מורכבות מאבני לגו בסיסיות. מולקולות חומר פשוטות ומסובכות בנויות מאטומים (והם בתורם בנויים מ... עזבו, לא מאמר בפיזיקה). המידע הגנטי שלנו שמקודד ב-DNA בנוי מארבע "אותיות בסיסיות A,T,C,G". התמונה שאתם רואים במסך המחשב מורכבת מ**פיקסלים** - נקודות על המסך שכל אחת מהן היא בעלת צבע אחיד (שבתורו מורכב משלושה צבעים - אדום, ירוק, כחול - בעוצמות משתנות). כאשר מדברים על משהו שמורכב מאבני יסוד בסיסיות לא מספיק לומר מה אבני היסוד - גם צריך להסביר איך הן מתחברות זו לזו כדי ליצור דברים. אצטון ופרופיונאלדהיד הן שתי מולקולות שונות שמורכבות בדיוק מאותם אטומים אבל מחוברים בצורה שונה. כרגע עומד מולי רובוט לגו שבעזרת אותן אבני בניין בדיוק שלו יכלתי להרכיב גם מסוק או טנדר.

כאשר מדובר על לגו, יש אינספור אבנים בסיסיות, אבל אצלנו במדעי המחשב יש בדיוק שני אבני בניין: הספרות 0 ו-1, שבהקשר הזה נקראות **ביטים**. אנחנו בונים מהן הכל. כל פריט מידע במחשב הוא, בסופו של דבר, ביטים. המספרים השלמים; והמספרים הממשיים; וקומנדר קין והרפתקאותיו והמסמך שאני כותב כרגע וכל המידע שאי פעם נכתב בפייסבוק וכל סרט קולנוע שאי פעם אוכסן במחשב - כולם בסופו של דבר בנויים רק מ-0 ו-1. למה? למה לא לאפשר אבני בניין מורכבות יותר? כי קל, ברמת החומרה של המחשבים, לעבוד רק עם שתי אבני הבניין הללו (בלשון ציורית ולא מדויקת, קל להבדיל ביניהן במערכת אלקטרונית בעזרת "יש זרם חשמלי" ו"אין זרם חשמלי"). גם האופן שבו אנחנו מחברים את 0 ו-1 זה לזה הוא פשוט ביותר - אנחנו פשוט כותבים אותם בשורה. למשל: 011010101

רצף הביטים הזה הוא דוגמא לפריט מידע שמאוחסן במחשב. אבל **איזה** מידע? ובכן, כאן ההקבלה ללגו או למולקולות קצת משתנה. המחשב יכול לקחת את אותה סדרה של אפסים ואחדות ולחשוב עליה כאילו היא מייצגת דברים שונים ומשונים. היא יכולה לייצג מספר, והיא יכולה באותה מידע בדיוק גם לייצג אות. בשל כך המחשב על פי רוב מבצע איזה שהוא סוג של **פירוש** כדי להבין איך לחשוב על הסדרה הזו כרגע. זה דומה לאופן שבו מילים נהגות בתור סדרה של הברות בסיסיות, אבל אותו צליל, בשפות שונות, יכול להיות בעל משמעויות שונות. "היא" בעברית ו-he באנגלית נשמעים אותו דבר אבל הם **מתפרשים** שונה, בהתאם לשומע והשפה שהוא מצפה לשמוע באותו הרגע.

שפות תכנות משתמשות ב**משתנים**. משתנה הוא מקום בזיכרון של המחשב שניתן לו שם קליט בתוך הקוד של התוכניות ובאמצעות השם הזה אפשר לומר לתוכנית לעשות עם המקום הזה דברים - לכתוב שם הרבה פעמים 0, לכתוב שם הרבה פעמים 1, לכתוב 01010101 וכדומה. כדי שלתוכנית יהיה קל להבין מה בדיוק אמור לקרות עם המקום הזה בזכרון, למשתנים בדרך כלל יש **טיפוס** - משהו שכולל מידע על "מה המשתנה אמור לייצג". מה זה בדיוק אומר - זה משתנה משפת תכנות לשפת תכנות, ואפילו מסוג אחד של טיפוס לסוג אחר, מבחינת רמת הפירוט שאליה ההגדרה נכנסת. למשל, זה יכול לכלול מידע על כמות הביטים שהמשתנה משתמש בהם (לפעמים בכמה ביטים **בדיוק** הוא משתמש, ולפעמים בכמה ביטים **לכל הפחות** הוא אמור להשתמש). פרט לכמות הביטים הטיפוס גם כולל לפעמים מידע על איך אמורים להתייחס אליהם. אותנו מעניינים בהקשר של הקוד שלנו שני טיפוסים שבהם משתמשים בשפת C: הראשון הוא long, שמיועד לתאר ערכים מספריים שלמים, והשני הוא float שנועד לתאר מספרים שיכולים להיות גם שבריים ומיוצגים בייצוג שנקרא **נקודה צפה** ואסביר בקרוב.

נתחיל בלדבר על long. זו דוגמה לטיפוס שמגדיר את ה"בערך" אבל ההגדרה שלו לא נכנסת לפרטים מדוייקים. אין הגדרה חד משמעית לכמות הביטים שמשתנה מסוג long משתמש בהם, אבל התקן קובע שהוא ישתמש **לפחות** ב-32 ביטים. בהקשר של הקוד שהופיע ב-Quake אנחנו יודעים שהכוונה הייתה ל**בדיוק** 32 ביטים כי אחרת לא ברור מה הולך שם. לצורך מה שקורה בקוד חשוב שמספר הביטים של ה-long יהיה שווה למספר הביטים ש-float משתמש בו (והמספר הזה הוא **חד משמעית** 32, כי כך קובע התקן). אין גם הגדרה חד משמעית לאופן שבו הביטים של משתנה מטיפוס long אמורים להתנהג, אבל

בפועל מה ש-long תמיד עושה הוא לחשוב על הביטים שלו כמייצגים מספר שלם בבסיס בינארי. יש לי פה הסבר על בסיסי ספירה אבל הנה הרעיון הבסיסי: בבסיס בינארי כל מספר מיוצג על ידי סדרת ביטים שמתארת אותו כסכום של חזקות של 2. למשל, סדרת הביטים 1101 אומרת "זה המספר שמיוצג על ידי הסכום 20+22+23=1+4+8=13". הביט הכי **שמאלי** מייצג את החזקה **הכי גבוהה** של 2 שמחברים. כל זה קורה גם בבסיס 10, כמובן: אנחנו רגילים כבר לתרגם אוטומטית משהו כמו 1,089 ל-"אלף ועוד שמונים ועוד תשע" בלי אולי לשים לב לכך שאנחנו מחברים חזקות של 10 שנכפלות במקדם כלשהו בבסיס בינארי המקדם הוא רק 0 או 1, אבל הרעיון הוא אותו רעיון.

יש לייצוג מספרים על ידי long רמת סיבוך נוספת שאני חוסך מכם במאמר הזה כי היא לא רלוונטית - האופן שבו מיוצגים מספרים שליליים. לא ניכנס לזה כרגע. ובמקום זה נעבור לדבר על הייצוג של מספרים על ידי נקודה צפה.

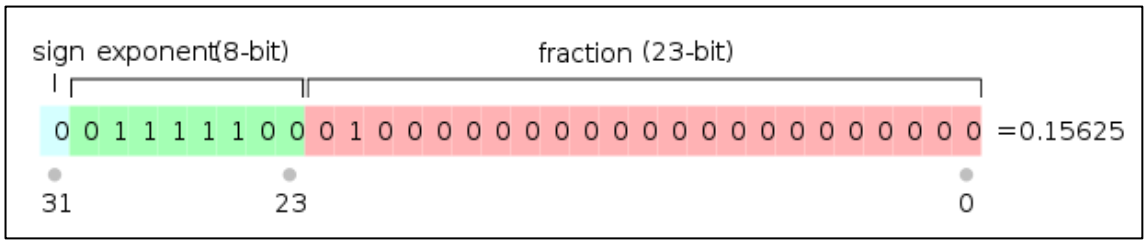
פרק חמישי (ובו נתוודע לפרנקנשטיין של שפות התכנות - הנקודה הצפה)

בואו נתחיל מכך שאודה ששיקרתי לכם. קודם הצגתי את העניינים כאילו הרעיון ב-long הוא ייצוג של מספר שלם והרעיון ב-float הוא ייצוג של מספר "ממשי", בפרט כזה שיכול להיות שבר. ובכן, ראשית כל float, לא יכול לייצג מספר ממשי כללי, למשל את π. כל מה שהוא יודע לייצג הוא מספרים רציונליים - שברים שאפשר להציג בתור $\frac{a}{b}$ כאשר a, b שניהם שלמים. שנית, אם כל מה שהייתי רוצה הוא לייצג רציונליים הייתי יכול פשוט להשתמש בזוג long, לא לגמרי ברור שצריך טיפוס נתונים חדש בשביל זה. אם כן, "לייצג מספר ממשי" או "לייצג שבר" **איננה** הסיבה שבגללה אנחנו מתעניינים ב-float. אז מה כן הסיבה?

הסיבה היא שלפעמים לא אכפת לנו אם המספר שלנו לא מיוצג בצורה **מדוייקת**. לפעמים אפשר לחפף ולעגל קצת, אם זה משתלם לנו. הרעיון ב-float הוא לוותר קצת על הדיוק המושלם ש-long מציע ותחת זאת להרחיב בצורה **משמעותית** את טווח המספרים שאפשר לייצג באמצעות 32 ביט. אם באמצעות long אפשר לייצג במדויק **כל מספר** בתחום שבין 0 ל-2^{32} (מי שרוצה לנטפק - תזכרו, אמרתי שלא אכנס לשלמים שליליים פה) הרי שבאמצעות float אפשר לייצג מספרים עד בערך 2^{127}, ושברים עד בערך 2^{-262} ואפילו קטן יותר מכך. המחיר הוא שאי אפשר לייצג **את כל המספרים** בטווחים הללו; יש לנו מגבלת דיוק. על פי רוב, בשימושים של float שמעניינים אותנו המגבלה הזו לא מפריעה לנו.

ב-float גם כן יכולים להיות מספרים שליליים, והפעם גם אתייחס לאופן שבו מייצגים אותם כי הוא קצת יותר פשוט מאשר ב-long ולשם שינוי גם מוגדר היטב. בכלל, לנקודה צפה יש יתרון שהיא מוגדרת יחסית טוב בסטנדרט של ה-IEEE ורוב מי שמממש נקודה צפה (בתוכנה/חומרה) יתאים את עצמו לסטנדרט. הקוד של 0x5f3759df מתבסס על זה, כמובן.

מספר בייצוג float מורכב מ-32 ביט, שמחולקים לשלוש קבוצות: הביט הראשון, השמאלי ביותר, הוא **הסימן** של המספר. אם הוא 0, המספר חיובי; אם הוא 1, המספר שלילי. 8 הביטים הבאים נקראים **האקספוננט** של המספר, ו-23 הביטים הנותרים נקראים **המנטיסה** שלו.



כדי להבין את המשמעות של אלו, בואו נראה לרגע על דרכים שונות שבהן אפשר לייצג את המספר 314.15. אני יכול לכתוב סתם 314.15, אבל אני גם יכול לכפול בחזקות של 10: למשל, לכתוב $31.415 \cdot 10^1$ או $3.1415 \cdot 10^2$, או $3141.5 \cdot 10^{-1}$ וכדומה. הבנתם את הרעיון: אני לוקח את המספר הבסיסי 314.15, ו"מזיז" את הנקודה העשרונית ("מציף" אותה) כשהמחיר הוא כפל בחזקה מתאימה של 10. הזזתי את הנקודה שמאלה? אני כופל בחזקה חיובית של 10. הזזתי אותה ימינה? אני כופל בחזקה שלילית. באופן הזה אפשר להחליט שכל מספר ייוצג בצורה "נורמלית" שבה יש בדיוק ספרה אחת משמאל לנקודה העשרונית; הייצוג ה"נורמלי" של 314.15 יהיה, אם כן $3.1415 \cdot 10^2$, ה**אקספוננט** של המספר הזה הוא החזקה של 10 בייצוג הנורמלי, וה**מנטיסה** שלו היא המספר שבו מכפילים מצד שמאל.

בואו נראה עוד דוגמה. את המספר 1,000 קל לייצג עם נקודה צפה: $1.0 \cdot 10^3$. מה על מספר ששונה ממנו טיפ-טיפה, נאמר 1,002? אותו אפשר לייצג על ידי $1.002 \cdot 10^3$. שימו לב מה קרה - נזקקנו ליותר ספרות במנטיסה כדי לייצג את המספר הזה מאשר את 1,000 שמיוצג בצורה ישירה באמצעות האקספוננט. באופן דומה, אם אני ארצה לייצג את מיליון זה יהיה קל, אבל אם ארצה לייצג את "מיליון ועוד 2" אצטרך עוד הרבה ספרות במנטיסה. וגם את 10^{100} קל לייצג, אבל לייצג את $10^{100}+2$ כבר יהיה יותר מדי עבורי - אין לי מספיק מקום במנטיסה בשביל זה כי אצטרך לכתוב 1.000...2 כאשר יש בערך מאה אפסים. הנה כי כן, זו בדיוק מגבלת ה"חוסר דיוק" שדיברתי עליה. את $10^{100}+2$ אני לא יכול לייצג, אבל אני יכול להסתפק ב-10^{100} שאותו אני כן יכול לייצג והוא קירוב מצויין ל-$10^{100}+2$. כל עוד אני לא **חייב** ייצוג מדויק של כל המספרים הללו.

בסדר עד כאן? אז בואו נסבך קצת. הצגתי את מספרי הנקודה הצפה שלי כאילו הם כתובים בבסיס עשרוני, אבל בפועל float מיוצג על ידי ייצוג בינארי (יש גם נקודה צפה של מספרים עשרוניים אבל זה לא רלוונטי לכאן). זה אומר שהאקספוננט והמנטיסה שניהם נכתבים בבסיס בינארי ואנחנו כופלים את המנטיסה בחזקה של 2 ולא 10, אבל זה גם אומר עוד משהו - אין צורך לזכור את הספרה הבודדת שמשמאל לנקודה העשרונית במפורש; אנחנו יודעים שהיא לא 0, כי ככה מוגדרת הצורה הנורמלית של מספר - יש משמאל לנקודה בדיוק ספרה אחת שאינה 0. לכן, עבור מספר נקודה צפה בבסיס בינארי, המנטיסה מתארת רק את מה שקורה **מימין** לנקודה העשרונית - החלק השברי של המספר.

סיבוך נוסף שטרם דיברתי עליו הוא האופן שבו מאפשרים למספרים שליליים להופיע בתוך האקספוננט. מה שעושים הוא להשתמש במשהו שנקרא bias.יש 8 ביטים של אקספוננט, מה שאומר שאפשר לייצג איתם כל מספר מ-0 עד 255. מכיוון שרוצים חצי חיוביים וחצי שליליים, מגדירים bias של B=127 ומגדירים שהוא תמיד מחוסר מהאקספוננט. כלומר, אם E מייצג את 200 אז האקספוננט של המספר יהיה $2^{E-B}=2^{73}$ בצורה הזו האקספוננט הגבוה ביותר הוא **לכאורה** 2^{128} והנמוך ביותר הוא 2^{-127} אבל בפועל הסטנדרט לא מרשה לנו להשתמש באקספוננטים 11111111 ו-0000000 באופן חופשי: את 11111111 שומרים כדי לייצג את אינסוף ואת NaN ערך שאומר "לא קיבלתי מספר") ואילו 00000000 שמור כדי לאפשר ייצוג של 0 ושל מספרים נמוכים במיוחד (משהו שנקרא denormalized numbers שאני פשוט לא הולך לדבר עליהם כאן כי לא צריך את זה). לכן טווח האקספוננטים החוקי הוא מ-2^{127} ועד 2^{-126}.

בואו נחזור על מה שיש לנו במספר float:יש 32 ביטים בסך הכל. ביט אחד, שנקרא לו s, הוא ביט הסימן. 8 הביטים הבאים, שנקרא להם E, הם הביטים של האקספוננט: אפשר לכתוב $E = E_7E_6E_5E_4E_3E_2E_1E_0$ כאשר כל E_i באגף ימין הוא ביט בודד. לסיום, המנטיסה תסומן ב-M (ברשותכם, לא אכתוב את כל הביטים שלה). עכשיו, בהינתן s,E,M אפשר לחשב את הערך המפורש של המספר שהם מייצגים ככה:

$$(-1)^8 \cdot 1.M \cdot 2^{E-B}$$

לפעמים במקום 1.M יותר נוח וקריא לכתוב $1+\frac{M}{2^{23}}$ או אפילו לסמן $m=\frac{M}{2^{23}}$ ואז לכתוב:

$$(-1)^8 \cdot (1+m) \cdot 2^{E-B}$$

סיימנו עם זה! עכשיו אנחנו מבינים מספרי נקודה צפה ברמה שתספיק להמשך המאמר. נעבור סוף סוף לשאלת השאלות: מה קורה כשאני לוקח float וחושב על הביטים שלו כמגדירים long?

פרק שישי (ובו שלמים ושברים ולוגריתמים יפים אלו דברים שאותי משמחים)

בואו נסתכל על שורת ה"המרה" הידועה לשמצה מהקוד:

```
i = * ( long * ) &y; // evil floating point bit level hacking
```

כפי שאמרתי קודם, מה שקורה בשורה הזו איננו המרה - אנחנו לא אומרים לתוכנית לקחת את ה-float שלנו ולעגל אותו עד שיתקבל מספר שלם או משהו. אנחנו עושים משהו ברוטלי ומסוכן באופן כללי: לוקחים את 32 הביטים בזיכרון שמיוצגים על ידי y ואומרים לתוכנית לחשוב עליהם בכוח בתור long. למי שסקרן, זה האופן הטכני שבו זה נעשה: ראשית אנחנו מבקשים "נא לתת לנו את הכתובת בזכרון שבה המידע של y שוכן". זה מה שעושה האופרטור & כשהוא מוצמד ל-y. אחר כך אנחנו לוקחים את הכתובת הזו, שכרגע התוכנית חושבת עליה בתור "כתובת של float", ואנחנו מבצעים עליה פעולה שבשפת c הנקראת casting ומתבצעת על ידי ה-* long שבסוגריים (הסוגריים עצמם אומרים לתוכנית שיש כאן פעולת casting). הפעולה הזו אומרת לתוכנית - "נכון שיש לך כתובת שאת חושבת עליה בתור כתובת של float? מעכשיו

תחשבי עליה בתור כתובת של long. הכוכבית האחרונה בתחילת השורה היא דרך לומר "אוקיי, עכשיו נא לתת לי את הערך המספרי שכתוב בתוך כתובת הזיכרון שלך". התוכנית עושה את הדבר הבא: בשלב הזה, יש לה כתובת זכרון ולידה סימון "הכתובת הזו מכילה long. "אז התוכנית לוקחת את 32 הביטים מהכתובת, מתייחסת אליהם בתור מספר long ומציבה בתוך i. כל זה ברור למי שמכיר את השפה, ואני מנחש שמי שלא מכיר אותה כבר הלך לאיבוד. לא נורא, לא צריך להבין מה השורה עושה ברמה הטכנית הזו, רק מה האפקט שזה משיג.

ומה זה עושה בפועל, למספר? מיש-מש, זה מה שזה עושה. ביטים שלפני רגע הייתה להם משמעות אחת מקבלים משמעות לא לגמרי קשורה, אבל גם לא לגמרי שונה. אנחנו עדיין יכולים לחשוב על הביטים בתור שלוש קבוצות - s,E,M - רק שעכשיו כל קבוצה תורמת משהו למספר השלם שמיוצג על ידי ה-long.

המשך הניתוח שאציג מתבסס בעיקר על המאמר הזה שמספק ניתוח יפה מאוד של הסיפור הזה. יש עוד ניתוחים שונים ומשונים שאפשר לעשות וקשה לי לומר מי מהם הוא ה"נכון", אבל זה שאציג כרגע הוא ללא ספק הפשוט ביותר (והנחמד ביותר, לטעמי) מביניהם.

בגדול, מאוד בגדול, מה שקורה כשמעבירים ככה מספר מ-float אל long הוא שממירים אותו **ללוגריתם** של עצמו כפול איזה שהוא קבוע. בואו נזכר מה זה לוגריתם בכלל. אם $x = 2^y$ אז $\log x = y$ כלומר, הלוגריתם של x הוא המספר שאם מעלים את 2 בחזקה שלו, מקבלים את x (אני מציג פה את מה שנקרא "לוגריתם על בסיס 2" כי זה מה שרלוונטי לנו במאמר הזה). למשל $\log 8 = 3$, כי $2^3 = 8$ לעומת זאת $\log 7$ לא הולך לצאת מספר יפה אלא משהו אי-רציונלי שנמצא אי שם בין 2 ל-3.

אנחנו אוהבים לוגריתמים כי הם מבצעים מעין "הורדה בדרגת הקושי" לפעולות חשבוניות מסובכות. כפל וחילוק הופכים להיות חיבור וחיסור, ואילו העלאה בחזקה והוצאת שורש הופכות להיות כפל וחילוק. הנה הכללים המתאימים:

$$\log(a \cdot b) = \log a + \log b$$

$$\log(\frac{a}{b}) = \log a - \log b$$

$$\log a^n = n \log a$$

$\log \sqrt[n]{a} = \frac{1}{n} \log a$ (זה בעצם נובע מכך ש-$\sqrt[n]{a}$ הוא $a^{\frac{1}{n}}$)

בימים עברו, לפני המצאת המחשבון, השתמשו ב**טבלאות לוגריתמים** כדי לבצע חישובים מסובכים: טבלת לוגריתמים כללה ערכים של מספרים ולידם את הערך של הלוגריתם שלהם. אם הייתי רוצה לבצע פעולה מסובכת כמו כפל 128 ב-512 מה שהייתי עושה הוא להסתכל בטבלת הלוגריתמים, לראות שהלוגריתמים של שני המספרים הללו הם 7 ו-9 בהתאמה, **לחבר** את 7 ו-9 לקבלת 16, ואז להסתכל בטבלת הלוגריתמים ולראות שהמספר שהלוגריתם שלו הוא 16 הוא המספר 65536. היו גם כלים מיוחדים בשם **סרגלי**

חישוב שסייעו לעשות את החישוב הזה. בצורה הזו אמנם נדרשה עבודה ראשונית ביצירה של טבלת הלוגריתמים/סרגל החישוב, אבל בעבודה היומיומית הם חסכו הרבה כאב ראש בביצוע פעולות חשבון. מה שאני רוצה לומר כאן הוא שלוגריתמים זה דבר נפלא שכשלומדים אותו בתיכון לפעמים בכלל לא מבינים בשביל מה הוא טוב.

המקרה הנוכחי מושלם עבור לוגריתמים $\frac{1}{\sqrt{x}}$. זו דרך אחרת לכתוב $x^{-\frac{1}{2}}$ נפעיל על זה לוגריתם ונקבל $\log\left(x^{-\frac{1}{2}}\right) = -\frac{1}{2}\log x$. אז ברמת הלוגריתמים כל פעולת החישוב המסובכת שאנחנו רוצים לבצע היא בסך הכל כפל במינוס חצי. ומה תגידו? כפל כזה מתבצע בפועל!

```
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

תתעלמו לרגע מהקבוע המסתורי שלנו. מה שיש באגף ימין הוא $-\frac{i}{2}$. בשביל לראות את זה אני צריך להסביר סוף סוף מה אומר ה-<<1 הזה. ובכן, << זה אופרטור שמופעל על מספר שלם ומבצע **הזזה ימינה** של הביטים שלו. ה-1 שמצד ימין של האופרטור אומר כמה להזיז ימינה - 1 פירושו להזיז בדיוק פעם אחת. כלומר, המספר 01100110 יהפוך להיות המספר 00110011 וכן הלאה. בפועל הפעולה הזו מבצעת חלוקה ב-2 של המספר השלם (עם עיגול למטה במקרה שמקבלים שבר). אם כן, מה שהשורה המסתורית הזו הוא הוא לכפול את i במינוס חצי ולהוסיף לו את הקבוע המסתורי בתור... לא לגמרי ברור בתור מה עדיין. אז בואו נמשיך עם הפרטים.

פרק שביעי (שבו התעלומה באה על פתרונה והקוראים מתלוננים על אנטי-קליימקס)

כעת, אמרנו שמספר float מיוצג על ידי ביט אחד של s, אחריו ביטים של E ואחר כך ביטים של M. הביטים של M הם הראשונים, ולכן הם אכן מייצגים בדיוק את המספר M. הביטים E, לעומת זאת, מתחילים החל מהמקום ה-24. אם הביט במקום ה-1 מייצג את הספרה שמתאימה ל-2^0 הרי שהביט במקום ה-24 מיצג את הספרה שמתאימה ל-2^{23}, ולכן E מייצג את המספר:

$$2^{23}E_0 + 2^{24}E_1 + \cdots + 2^{30}E_7 = 2^{23}(E_0 + 2^1E_1 + \cdots + 2^7E_7) = 2^{23}\cdot E$$

ולבסוף, הביט s של הסימן מייצג את 2^{31}. כלומר, המספר כולו מתפרש בתור ה-long הבא: $2^{31}s + 2^{23}E + M$ בפועל, אפשר להתעלם מהביט s של הסימן: הוצאת שורש היא פעולה שאנחנו מבצעים רק עבור מספרים חיוביים, ולכן הסימן של ה-float הוא חיובי, מה שאומר ש-s=0 בכל מה שנעשה. על כן המספר מתפרש בתור L·E+M כאשר $L=2^{23}$ הוא קבוע שיאפשר לנו לקרוא יותר בקלות מכאן ואילך. זכרו שהמספר המקורי בתור float היה $x = (1+\frac{M}{L})2^{E-B}$. נשאלת כעת השאלה - עד כמה L·E+M הזה אכן יהיה דומה ל-$\log x$? לצורך כך כדאי לקבל הערכה כלשהי לערך של $\log x$, ולצורך כך אני אשתמש ב**קירוב** מוכר במתמטיקה: זה ידוע שכאשר t הוא קטן יחסית, אז $\log(1+t) \approx t$ (למעוניינים, זה נובע מפיתוח טיילור של $\log(1+t)$ במקרה שלנו, $\frac{M}{L}$ הוא קטן יחסית (כי הגדול של M חסום על ידי L) ולכן אפשר להשתמש

בקירוב $\log(1+\frac{M}{L}) \approx \frac{M}{L}$. מצד שני, אין סיבה שנשתמש בקירוב הזה באופן עיוור ופשוט נתעלם מכך שאולי כדאי להוסיף "תיקון" כלשהו שיפצה על החלקים שהעפנו מהקירוב. אז נגדיר פרמטר σ שאת הערך שלו נוכל לבחור באופן שרירותי ונשתמש בקירוב הבא: $\log\left(1+\frac{M}{L}\right) \approx \frac{M}{L} + \sigma$ לא לגמרי ברור בשלב הזה אילו ערכים של σ הם טובים לנו ואיזה לא (אולי σ=0 הוא טוב?) ולכן אנחנו לא מתחייבים על ערך ספציפי עבורו.

כעת, נקבל מהזהויות שקשורות בלוגריתם שראינו למעלה את הדבר הבא:

$$\log(x) = \log\left(\left(1+\frac{M}{L}\right)2^{E-B}\right) = \log\left(1+\frac{M}{L}\right) + \log 2^{E} \approx \frac{M}{L} + \sigma + E - B$$

עכשיו, אם נסמן $y = \frac{1}{\sqrt{x}}$, הרי ש-y הוא המספר שאנחנו מחפשים. בייצוג על ידי נקודה צפה גם הוא ישתמש בפרמטרים E,M, אבל כאלו שיהיו שונים מאלו של x. לכן נשתמש בסימונים כדי להבדיל ביניהם: את M,E שהשתמשתי בהם עד כה אסמן מעכשיו ב-E_x ו-M_x ואילו את האקספוננט והמנטיסה של y, שאותם אני מחפש, אסמן ב-E_y ו-M_y אותו חישוב כמו קודם עובד גם עבור yy, ולכן יש לנו עכשיו שלוש משוואות:

$$\log(x) \approx \frac{M_x}{L} + \sigma + E_x - B$$

$$\log(y) \approx \frac{M_y}{L} + \sigma + E_y - B$$

$$\log(y) = -\frac{1}{2}\log x$$

נשלב את המשוואות הללו יחד:

$$\frac{M_y}{L} + \sigma + E_y - B \approx -\frac{1}{2}\left(\frac{M_x}{L} + \sigma + E_x - B\right)$$

נעביר את הקבועים σ,B אגף ונקבל:

$$\frac{M_y}{L} + E_y \approx \left(B + \frac{1}{2}B\right) - \left(\sigma + \frac{1}{2}\sigma\right) - \frac{1}{2}\left(\frac{M_x}{L} + E_x\right)$$

כלומר:

$$\frac{M_y}{L} + E_y \approx \frac{3}{2}(B-\sigma) - \frac{1}{2}\left(\frac{M_x}{L} + E_x\right)$$

לסיום, נכפול את שני האגפים ב-L ונקבל:

$$M_y + LE_y \approx \frac{3}{2}L(B-\sigma) - \frac{1}{2}(M_x + LE_x)$$

ותראו מה קיבלנו! ה- $M_y + LE_y$ באגף ימין הוא **בדיוק** הערך של המספר שממנו התחלנו, כשמפרשים את הביטים שלו בתור long; והערך באגף שמאל הוא מספר שאם נפרש את הביטים שלו בתור float אז המנטיסה שלו תהיה M_y והאקספוננט שלו יהיה E_y. זה גם בדיוק מה שעושים בשורה הבאה:

```
y = * ( float * ) &i;
```

ועל כן, המשוואה שלעיל היא בדיוק מה שמנחה את שורת ה-what the fuck?הידועה לשמצה:

```
i = 0x5f3759df - ( i >> 1 ); // what the fuck?
```

זה מסביר למה היא נראית ככה וגם מיהו הקבוע המסתורי: הוא פשוט $\frac{3}{2}L(B-\sigma)$. זכרו ש-L הוא פשוט המספר 2^{23} ו-B הוא המספר 127 - אלו פרמטרים שנטועים עמוק בהגדרה של ה-IEEE למהו float, אבל גם אם הערכים שלהם היו שונים היינו עדיין מקבלים משוואה דומה, רק עם "קבוע מסתורי" שונה.

כמובן שעכשיו נשאלת השאלה איזה ערך של פרמטר σ הולך לתת את הקבוע 0x5f3759df תוך הביטוי $\frac{3}{2}L(B-\sigma)$. התשובה היא שזה σ=0.0450465, אבל זה בעצם לא אומר לנו שום דבר. כאן בעצם מגיע החלק המאכזב ביותר בכל הסיפור - מי שיצפה לראות איזה הגיון קוסמי שבזכותו נוצר דווקא המספר 0x5f3759df ולא אחרים לא ימצא אותו - זה ככל הנראה מספר שכותב הקוד הגיע אליו אחרי קצת ניסוי וטעיה - ראה שהוא עובד מספיק טוב, ולא ניסה לשפר יותר. עדיין, אם מישהו רוצה ניתוח קצת יותר מפורט של ערכים אפשריים אחרים, אפשר להסתכל בתזה הזו, שבכלל נמנעת משימוש בלוגריתמים ומסתכלת בצורה מפורשת מאוד על ההבדל בתוך ה-float שגורמות הפעולות שמבצעים עליו. מסבירים שם, למשל, למה קבוע שנותן ערך **טוב יותר** בתור הקירוב אחרי השורה הזו הוא **פחות טוב** באופן כללי, כי ניוטון-רפסון מחזיר עליו תוצאה פחות נחמדה, וגם נותנים ערך טוב יותר מ-0x5f3759df בתור קבוע קסם מסתורי עבור הפונקציה. מבחינתי זה מחסל את 0x5f3759df המסכן לגמרי - הוא לא כל כך מעניין אם הבחירה בו הייתה כל כך שרירותית. אולי יום אחד אתבדה ואגלה שהוא נבחר מסיבות מצויינות שאיני מכיר.

פרק שמיני (שבו אנחנו תוהים בשביל מה כל זה היה טוב)

הסיפור שלנו מתקרב לסופו, אבל אני רוצה להזכיר למה בכלל נכנסנו אליו מלכתחילה. כזכור, שם המשחק הוא גרפיקה. הגרפיקה הזו:

בשביל לייצר גרפיקה יפה שכזו צריך לדעת לחשב כל מני חישובים. למשל, איך אור משתקף מכל מני משטחים. כשהיינו בימי Wolf3D העליזים כל המשטחים היו פשוטים מאוד - קירות שעמדו בזווית של 90 מעלות ביחס לרצפה וזהו. אבל בעולם תלת-ממדי שנראה טוב, זה לא המצב. יש משטחים באלכסונים, ויש משטחים מעוגלים ועוד ועוד. כשרוצים לחשב איך תתנהג קרן אור שפוגעת במשטח בנקודה כלשהי, אנחנו צריכים לדעת משהו על "הכיוון המקומי" של המשטח באותה נקודה. הכיוון הזה מיוצג באמצעות **וקטור יחידה** במרחב התלת ממדי. "וקטור יחידה" פירושו שהאורך של הוקטור הוא 1. למה דווקא 1? כי פעולות שמערבות את הוקטור הזה דורשות שהוא יוכפל סקלרית בדברים, ואם האורך שלו הוא לא 1 אז הוא "ינפח" את הדברים הללו באופן מלאכותי. בפועל מה שקורה הוא שקודם כל מוצאים את הכיוון של הוקטור - כלומר, מוצאים וקטור **כלשהו** שמצביע בכיוון הנכון, ואז **מנרמלים** את הוקטור - מחלקים אותו באורך של עצמו. אם v הוא וקטור, אז האורך שלו הוא $\|v\| \triangleq \sqrt{u \cdot u}$. על כן, הוקטור המנורמל $\frac{v}{\|v\|}$ שווה ל-$v \cdot \frac{1}{\sqrt{u \cdot u}}$. הופס! אנחנו צריכים למצוא את ההופכי של שורש של משהו!

אם מסתכלים בקוד ומחפשים שימושים של Q_rsqrt זה בדיוק מה שמוצאים. למשל:

```
189  void VectorNormalizeFast( vec3_t v )
190  {
191          float ilength;
192
193          ilength = Q_rsqrt( DotProduct( v, v ) );
194
195          v[0] *= ilength;
196          v[1] *= ilength;
197          v[2] *= ilength;
198  }
199
```

שמופיע בקובץ q_math.c (ואפשר לראות כרגע כאן).

"רגע, זה הכל?" אולי אתם שואלים. ובכן, צריך לזכור שאנחנו מחשבים את הוקטורים הללו עבור אינספור נקודות על כל המשטחים שסביבנו. ככל שיש יותר וקטורים, כך התיאור שלנו של המשטחים נראה יותר ריאליסטי. לכן הפונקציה הזו הולכת להיקרא המון פעמים. ככה בדיוק זה אופטימיזציות: בסופו של דבר צוואר הבקבוק הוא בדיוק בפונקציות הכי קטנות ופשוטות ושם שוברים את הראש על מציאת דרכים טובות יותר לבצע את החישוב.

אז בעצם, מה גרם לחישוב להיות כל כך טוב? שילוב של שני דברים: ראשית, ניוטון-רפסון, שהיא שיטה מגניבה באופן כללי; ושנית, שימוש (קונספטואלי, לכל הפחות) שבוצע ללא שום המרה מפורשת אלא פשוט התייחסות קצת שונה לביטים של הערכים שפעלנו עליהם. אלו הרעיונות המגניבים כאן. ומה עם המספר המסתורי 0x5f3759df? למה בדיוק הוא נבחר? האם יש לו איזו תכונה קסומה שבעטיה הוא נבחר? כנראה שלא, אבל זו תישאר אחת מהתעלומות הקטנות של מדעי המחשב גם לדורות הבאים.

המאמר נכתב במקור כפוסט בבלוג "לא מדויק" של דוקטור גדי אלכסנדרוביץ', לפוסטים המקוריים ולעוד פוסטים בנושא זה ואחרים, ניתן להכנס ל:

http://www.gadial.net/2017/08/24/0x5f3759df_part_2/

http://www.gadial.net/2017/08/22/0x5f3759df_part_1/