1. פיתוח משחק "Geometry Dash"
2. אהרון משה שמסיאן: 205779598

**3. הסבר כללי של התרגיל:**

במסגרת התרגיל פיתחנו את המשחק **Geometry Dash** המשחק הוא פרויקט גדול ומבני, הכולל שימוש בתכנות מונחה עצמים (OOP), בניית מחלקות שונות ניהול משאבים, ירושות, ופולימורפיזם.

המשחק מאפשר לשחקן לשלוט בדמות קופצת, לעבור מכשולים ולהגיע לסוף השלב מבלי להיתקל בהם. בנוסף, פיתחנו **תפריט ראשי**, **חנות דמויות** לרכישת דמויות עם כסף שהשחקן אסף, מערכת צלילים ומערכת ניהול כללית.  
• הטענת שלבי המשחק מקובצי טקסט – כל תו בקובץ מייצג אובייקט (שחקן, אויב, מגרש, דלת יציאה וכו׳), והאובייקטים נוצרים דינמית באמצעות תבנית ה־Factory שלנו.

• מנגנון פיזיקה בסיסי לניהול קפיצה ונפילה (מהירות אנכית, כוח כבידה) באמצעות מחלקת Move.

• טיפול בהתנגשויות בזמן ריצה (Double Dispatch) המאפשר החלטה מבוססת סוגי שני האובייקטים המעורבים בהתנגשות.

• הפרדה ברורה בין אובייקטים ניידים לנייחים וניהולם ב־std::vector נפרדים ב־GameController.

**4. רשימות קבצי המשחק:**

מערכת התפריטים במשחק אחראית על כל ממשק המשתמש הגרפי (GUI) מחוץ למשחק עצמו, כולל התפריט הראשי, החנות, כפתורי פעולה, תצוגת עזרה ויציאה. היא נבנתה בצורה מודולרית תוך שימוש בעקרונות תכנות מונחה עצמים (OOP) וכוללת מחלקות בסיסיות ויורשות, מחלקות תצוגה וניהול, וכפתורים ייעודיים לכל פעולה נדרשת במשחק.

**Background**  
מחלקה זו מנהלת את הרקע הגרפי של התפריטים והחנות. היא טוענת קובץ תמונה ומציגה אותו בכל מסך נדרש, כך שהתפריטים ייראו מושכים ואחידים בכל חלקי המשחק.

**Button**  
מחלקת בסיס כללית לכל כפתורי המשחק. היא אחראית על ציור הכפתור, שמירת מיקומו ובדיקת לחיצות. מחלקות כפתור ייעודיות אחרות יורשות ממנה ומגדירות את פעולתן הייחודית.

**ButtonData**

מרכזת את כל הקבועים הגרפיים של הכפתורים, כולל גדלים ומיקומים בתפריט הראשי ובחנות, כדי לאפשר ניהול אחיד ונוח של מיקומי האלמנטים במסכים.

**Cancel**  
כפתור ביטול בתפריטים או בחנות. בעת לחיצה עליו, הוא מחזיר את המשתמש לתפריט הקודם או סוגר את החלונית הנוכחית בהתאם למיקום בו נמצא.

**CharactersButton**  
מחלקה עבור כפתורי הדמויות בחנות. מציגה את תמונת הדמות, עלותה, מצב הנעילה שלה ומאפשרת רכישתה. יורשת מהמחלקה Button ומוסיפה לוגיקת תצוגת מחיר ונעילה.

**Done**  
כפתור אישור פעולות בתפריטים. למשל, בסיום רכישה בחנות או לאחר הגדרות כלשהן, הכפתור מאפשר לשמור את השינויים ולחזור למסך הקודם.

**Exit**  
כפתור יציאה מתפריט המשחק. בלחיצה עליו, חלון המשחק נסגר והמשחק מסתיים.

**FontHolder**  
מחלקה סטטית לניהול פונטים. היא טוענת את קובץ הפונט בתחילת הריצה ומספקת טקסטים עם הפונט לכל כפתור ותפריט לפי צורך, כדי להבטיח אחידות עיצובית.

**GameStore**  
מחלקת ניהול חנות הדמויות. מציגה את הדמויות לרכישה, את מחירן, בודקת אם יש לשחקן מספיק מטבעות ומבצעת את פעולת הרכישה בהתאם.

**GitHubLink**  
כפתור ייחודי המאפשר לפתוח קישור ל־GitHub של הפרויקט ישירות מתוך התפריט. הכפתור שומר כתובת URL ומבצע את הפתיחה בלחיצה.

**Help**  
כפתור עזרה המציג הוראות הפעלה ומשחק לשחקן, כולל מידע כיצד לעבור מכשולים, להשתמש בכפתורים ולנווט בחנות.

**MenuAction(enumclass)**  
מכיל את כל סוגי הפעולות האפשריות בתפריטים, כגון התחלת משחק, יציאה, פתיחת עזרה או חנות. משמש כערך החזרה מכפתורים לפעולה שתבוצע בפועל.

**MenuManager**  
מחלקת ניהול כללית של כל מערכת התפריטים. אחראית על בניית התפריט הראשי, ניתוב בין תפריטים שונים והרצת הלולאות הגרפיות שלהם.

**PlayerAlpha / PlayerBeta / PlayerGamma / PlayerDelta / PlayerEpsilon / PlayerZeta**  
מחלקות מייצגות דמויות ספציפיות הניתנות לרכישה בחנות. כל מחלקה מציגה דמות אחרת עם עלות שונה ומורשת מ־CharactersButton.

**Start**  
כפתור התחלת המשחק. בלחיצה עליו, המשחק עובר מהתפריט הראשי לריצה בפועל של השלב.

**State**  
מחלקה לניהול מצבי התפריטים והמשחק, כגון האם השחקן בחר להתחיל לשחק, להיכנס לחנות, לקבל עזרה או לצפות במשחק.

**Store**  
כפתור לפתיחת חנות הדמויות מתוך התפריט הראשי. מוביל את השחקן למסך החנות לרכישת סקינים חדשים.

**Watch**  
כפתור צפייה במשחק או הדגמת gameplay. מאפשר לשחקן לראות את המשחק ללא שליטה אקטיבית, לצרכי למידה והיכרות.

**Object**

זוהי מחלקת הבסיס לכל האובייקטים במשחק. היא מטפלת במאפיינים בסיסיים כמו מיקום, ספריט, והאם האובייקט נמצא בתצוגה או "מת". היא גם מגדירה פונקציות וירטואליות לציור, זיהוי התנגשויות וטיפול בסוגים שונים של התנגשויות עם תת-מחלקות אחרות.

**MoveObject.**

המחלקה הזו יורשת מ object היא מחלקה ראשית של כל האובייקטים הנעים במשחק ,ויש בה את האובייקט שאחראי על הקפיצה הפיזיקלית ומשתנים לגבי מידע מצב האובייקט על לוח המשחק.

**Enemy**

מחלקה המממשת את האויב שבמשחק , האויב זז על הלוח עד שהוא מתנגש עם אובייקט אחר , אם כן הוא משנה כיון. אם הוא נפגש בשחקן , השחקן מת.

**Player**

הוא האובייקט העיקרי במשחק שהוא השחקן שמשחק . יש בו את ניהול ההתנגשויות עם האוביקטים השונים. השחקן קופץ בעזרת שימוש במחלקה move שממשת קפיצה לפי גרביטציה .

**SataticObject**

המחלקה הזו יורשת מ object היא מחלקת אב של כל האובייקטים שאינם זזים במשחק כגון קירת ומכשולים סטטים

,**ExitDoor**

`ExitDoor` היא מחלקה שיורשת מ-`StaticObject` ומייצגת דלת יציאה בשלב במשחק. היא מטפלת בהתנגשות עם אובייקטים מסוג `Player` ובעדכון המידע במשחק (באמצעות `ObjectInformation`) כדי לסמן שיש לעבור לשלב הבא. היא מכילה משתנה בוליאני `m\_nextLevel` שקובע אם יש להתקדם לשלב הבא.

**Obstcale**

`Obstacle` היא מחלקה שיורשת מ-`StaticObject` ומייצגת מכשול קבוע במשחק. היא מטפלת בהתנגשויות עם אובייקטים מסוג `Player

**Platform**

`Platform` היא מחלקה שיורשת מ-`StaticObject` ומייצגת משטח קבוע במשחק שעליו שחקנים יכולים לעמוד. הקוד מראה שהיא מקבלת מיקום וספרייט בבנאי שלה. ישנה גם שורת קוד מוערת המרמזת על שימוש במנגנון רישום במפעל (`Factory`) ליצירת אובייקטי פלטפורמה.

**ObjectInformation**

`ObjectInformation` היא מחלקת מאגר נתונים המשמשת להעברת מידע על מצב המשחק בין רכיבים שונים. היא עוקבת אחר דגלים כגון האם יש לטעון את השלב הבא (`m\_nextLevel`), האם השחקן מת (`m\_PlayerDead`), ומספר המטבעות שנאספו (`m\_coins`). היא מספקת פונקציות להגדיר ולשלוף את המידע הזה, כולל פונקציה ייחודית (`moveCoins`) שמחזירה את מספר המטבעות ומאפסת אותם**.**

**Move**

מחלקה שמטרתה לנהל את הפיזיקה הבסיסית של תנועה עבור אובייקטים במשחק. היא מתמקדת בעיקר בתנועה אנכית, השפעת כוח הכבידה ומכניקת קפיצה, ומאפשרת לאובייקטים לנוע ולהגיב לכוחות אלו בצורה מציאותית יחסית.  
**GameController** מחלקה מרכזית שמפעילה את לולאת המשחק: קוראת עדכונים (update), מציירת (render), טוענת שלבים מקובץ, ומנהלת את שני הוקטורים של אובייקטים (ניידים ונייחים).

**Factory** מבנה סטטי (std::map<char, FactoryFunction>) שאחראי לרישום דינמי של כל מחלקות ה־Object. כל מחלקה יורשת מ־Object ובונה משתנה static bool שמריץ בעת קומפילציה קריאה ל־Factory::registerIt.

**SoundManager** מחלקה סטטית שמטפלת בטעינה והפעלה של אפקטים ומוזיקה. מאפשרת שינוי עוצמה גלובלי ושמירה על המשך השמעה בין מצבי המשחק.

**5. פירוט מבנה נתונים עיקריים:**

בפרויקט נעשה שימוש במבני נתונים שונים לניהול התפריטים, החנות והדמויות. להלן הפירוט:

* **std::vector<std::unique\_ptr<Button>>m\_buttonStore**  
  וקטור המאגד את כל כפתורי החנות, כאשר כל כפתור מאוחסן כ־unique\_ptr על מנת לאפשר פולימורפיזם וניהול זיכרון בטוח מבלי צורך בניהול מחזור חיים ידני.
* **std::vector<std::pair<std::unique\_ptr<Button>, bool>> m\_character**  
  וקטור המכיל זוגות של כפתורי דמויות בחנות עם משתנה בוליאני המציין אם הדמות נרכשה או לא. מאפשר לבדוק בקלות את מצב הרכישה ולהציג נעילה בהתאם.  
  **std::vector<std::unique\_ptr<Object>> m\_movingObjects** וקטור המכיל את כל האובייקטים הניידים במשחק, כמו השחקן, אויבים, ומטבעות שנעים. כל אובייקט נשמר כ־unique\_ptr כדי לאפשר פולימורפיזם וניהול זיכרון בטוח ללא צורך בניקוי ידני.
* **std::vector<std::unique\_ptr<Object>> m\_staticObjects** וקטור המכיל את כל האובייקטים הנייחים במשחק, כגון קירות, משטחים ודלתות יציאה. גם כאן השימוש ב־unique\_ptr מאפשר שמירה על עקרונות OOP בצורה נקייה.
* **std::map<char, std::function<std::unique\_ptr<Object>(sf::Vector2f)>> m\_factoryMap** מבנה נתונים הממפה כל תו בקובץ שלב (למשל 'P' לשחקן, '#' לפלטפורמה) לפונקציית יצירה מתאימה. מנגנון זה מופעל בזמן ריצה ומאפשר טעינה דינמית של שלבים לפי תוכן טקסטואלי.
* **ObjectInformation m\_info** אובייקט מידע המשמש כמאגר נתונים משותף בין האובייקטים במשחק. הוא כולל מידע כמו האם יש לעבור לשלב הבא, האם השחקן מת, וכמות המטבעות שנאספו. מחלקות שונות במשחק מעדכנות אותו, ו־GameController מתבסס עליו כדי לדעת כיצד להמשיך.

**6. אלגוריתמים חשובים שהשתמשנו בהם.**

1. Double Dispatch

במערכת המשחק, ניהול התנגשויות בין אובייקטים מסוגים שונים מתבצע באמצעות מנגנון הנקרא **Double Dispatch** – שיטה המאפשרת קבלת החלטות בזמן ריצה על בסיס **שני הסוגים הדינמיים** של האובייקטים המעורבים בהתנגשות.

**איך זה עובד?** נניח שיש התנגשות בין אובייקט מסוג Player לבין אובייקט מסוג Obstacle. הקריאה מתחילה מהאובייקט הראשון:  
(player.handleCollision(obstacle

כיוון ש־Player יורש מ־MovingObject, והפונקציה handleCollision מוגדרת כווירטואלית, הקריאה הולכת ל:

Void Player::handleCollision(StaticObject& object);

בתוך פונקציה זו, מתבצעת קריאה חוזרת לאובייקט השני, אך הפעם עם הפנייה אליו:

object.handleCollisin(\*this)

אם object הוא בפועל Obstacle, הקריאה תנותב לפונקציה:

Void Obstcale::handleCollision(Player& player);

באמצעות שני שלבים אלו, כל אחד מהאובייקטים משתתף באופן אקטיבי בהחלטה כיצד להגיב להתנגשות – בהתאם לסוג האמיתי של שני הצדדים.

1. MOVE:  
   מחלקת Move מממשת פיזיקה בסיסית של קפיצה ונפילה במשחק, באמצעות ניהול של מהירות אנכית וכוח כבידה. המשתנה m\_velocityY מייצג את המהירות בציר האנכי (ציר Y), כאשר ערך שלילי פירושו תנועה כלפי מעלה וערך חיובי פירושו תנועה כלפי מטה. כאשר האובייקט קופץ, נקבעת לו מהירות התחלתית שלילית (m\_jumpSpeed), מה שגורם לו לעלות. בכל פריים, הפונקציה update מופעלת ומעדכנת את המהירות האנכית על ידי הוספת כוח הכבידה (m\_gravity \* deltaTime), וכך נוצרת האצה כלפי מטה – בדיוק כמו בעולם האמיתי.

ככל שהזמן עובר, המהירות עולה, כלומר האובייקט מאט את העלייה עד שהוא נעצר בשיא הקפיצה, ואז מתחיל ליפול. במהלך הנפילה, המהירות ממשיכה לגדול, ולכן הנפילה נעשית מהירה יותר. הפונקציה update אחראית גם על עדכון מיקום האובייקט על המסך בהתאם למהירות הנוכחית. אם האובייקט נמצא על הקרקע (m\_onGround == true), אין שינוי במיקום או במהירות האנכית, והמהירות מתאפס.

כאשר מתבצעת קפיצה (באמצעות startJump), האובייקט יוצא ממצב "על הקרקע" ומתחילה תנועה כלפי מעלה. החזרה למצב "על הקרקע" מתבצעת על ידי הקריאה ל־setOnGround(true) כאשר האובייקט נוחת, מה שמאפס את המהירות האנכית ומונע המשך נפילה. מנגנון זה מדמה תנועה טבעית של קפיצה ונפילה ויוצר תחושת פיזיקה חלקה במשחק.  
3.**Factory Pattern (תבנית מפעל):** המשחק עושה שימוש בתבנית העיצוב Factory לצורך יצירה דינמית של אובייקטים בזמן ריצה. כל מחלקת אובייקט (לדוגמה Player, Enemy, Platform) רושמת את עצמה בעת קומפילציה למבנה סטטי מסוג:  
 std::map<char, std::function<std::unique\_ptr<Object>(sf::Vector2f)>>.  
 בעת טעינת שלב, כל תו בקובץ (כגון 'P', 'E', '#') מפוענח ומומר לפונקציית יצירה שהוגדרה מראש, וכך נוצר האובייקט המתאים.

הודות למנגנון זה, ניתן להוסיף מחלקות חדשות למשחק בקלות – כל מחלקה אחראית לרשום את עצמה באופן עצמאי דרך שורת קוד מסוג:

cpp

CopyEdit

static bool registered = Factory::registerIt('X', [](sf::Vector2f pos) { return std::make\_unique<NewObject>(pos); });

**Lambda Functions 4 (ביטויי למדה):** הרישום למבנה ה־Factory מבוצע באמצעות למדה, המאפשרת להעביר בנאים מותאמים אישית לכל אובייקט. כל למדה מקבלת מיקום ומחזירה אובייקט חדש:

cpp

CopyEdit

[](sf::Vector2f pos) { return std::make\_unique<Player>(pos); }

באופן זה, נוצר מנגנון רישום פשוט וגמיש שמבוזר לכל מחלקה, בלי צורך בטבלאות ריכוז או תנאים מורכבים.

**טעינה חד־פעמית של משאבים (תמונות וצלילים):** מחלקות הגרפיקה והסאונד (כגון FontHolder, SoundManager, TextureHolder) משתמשות במשתנים **סטטיים פרטיים (private static)** כדי לטעון את המשאבים (תמונות, פונטים, קבצי סאונד) **פעם אחת בלבד** במהלך ריצת המשחק.  
 משאבים אלו נטענים בתחילת התוכנית או בעת הגישה הראשונה אליהם, ונשמרים בזיכרון כך שכל חלקי המשחק יכולים לגשת אליהם ללא טעינה חוזרת – דבר החוסך זמן ריצה ומשפר ביצועים.  
 שיטה זו מבטיחה גם אחידות עיצובית וגם יעילות גבוהה בשימוש במשאבים.

**7. תיכון הפרויקט:  
תכנון הפרויקט נעשה בגישת תכנות מונחה עצמים (OOP) תוך שמירה על עקרונות של הפרדה בין אחריויות, שימוש בפולימורפיזם, ו־תבניות עיצוב.**

#### **מבנה כללי:**

* **GameController  
   מחלקה מרכזית שמנהלת את ריצת המשחק בפועל (השלב). היא אחראית על:**
  + **ניהול לולאת המשחק (update + render)**
  + **טעינת שלבים מקבצי טקסט**
  + **ניהול שני וקטורים: אחד לאובייקטים ניידים (כמו שחקן ואויבים), ואחד לאובייקטים נייחים (כמו פלטפורמות ודלתות)**
  + **בדיקת התנגשויות בין כל זוג אובייקטים**
  + **התקדמות בין שלבים, זיהוי סוף שלב, מוות של שחקן ועוד**
* **MenuManager  
   אחראית על כל הממשק הגרפי (GUI) שמחוץ למשחק עצמו: התפריט הראשי, חנות הדמויות, תפריט העזרה וכו'. היא מציגה כפתורים על המסך, בודקת קלט מהמשתמש, ומפעילה פעולות לפי בחירה (כגון פתיחת חנות, התחלת משחק, יציאה וכו').**
* **Object  
   מחלקת בסיס משותפת לכל האובייקטים במשחק – ניידים או נייחים. מגדירה ממשק אחיד לציור, עדכון מיקום, והתנגשות. ממנה יורשים:**
  + **MovingObject – שחקן, אויבים, מטבעות**
  + **StaticObject – פלטפורמות, דלתות, מכשולים**
* **Player / Enemy / Coin / Obstacle / ExitDoor / Platform  
   כל אחת מהן יורשת מהמחלקות הנ״ל ומוסיפה התנהגות ייחודית. לדוגמה: השחקן קופץ, האויבים זזים אוטומטית, דלת היציאה בודקת אם יש לעבור שלב.**
* **Factory  
   מנגנון שממפה תווים לקונסטרוקטורים של אובייקטים. מאפשר לטעון שלבים מקבצי טקסט, כאשר כל תו במפה מייצג אובייקט. כל מחלקה שרוצה להירשם למשחק מוסיפה static bool שמבצע רישום למפה בזמן קומפילציה.**
* **ObjectInformation  
   מעין "מרכז בקרה" קטן, שדרכו האובייקטים מדווחים על אירועים במשחק: מוות, איסוף מטבע, סיום שלב. דרכו גם ה־GameController יודע מתי לסיים שלב או להפעיל פעולה.**
* **Move  
   מחלקה שמבצעת פיזיקה בסיסית – קפיצה, נפילה, כוח כבידה. בשימוש בעיקר על ידי Player, ומעדכנת את המיקום לפי זמן (deltaTime).**
* **SoundManager / FontHolder / TextureHolder  
   מנהלים משותפים לכל הרכיבים הגרפיים והקוליים. טוענים את המשאבים פעם אחת בלבד ומשתפים אותם בין כל החלקים. כולם משתמשים ב־משתנים סטטיים כדי למנוע כפילויות בזיכרון.**

#### **זרימת העבודה:**

1. **השחקן נכנס למשחק – מוצג תפריט ראשי דרך MenuManager.**
2. **לחיצה על Start מעבירה את השליטה ל־GameController.**
3. **GameController טוען קובץ שלב, יוצר את האובייקטים דרך ה־Factory.**
4. **בזמן המשחק:**
   * **כל אובייקט מעודכן ומצויר**
   * **בדיקות התנגשויות בין אובייקטים ניידים לנייחים**
   * **כל התנגשות מעובדת דרך Double Dispatch**
   * **במקביל, מידע מתעדכן ב־ObjectInformation**
5. **כשמתקיים תנאי מעבר (כמו סיום שלב) – המשחק נטען מחדש עם שלב חדש או חוזר לתפריט.**

**8. אין באגים ידועים.**

**9. הערות כלליות:**

**-מצורף סרטון הסבר על Factory בקישור המצורף:**

* + [**סרטון הסבר על Factory**](https://drive.google.com/file/d/1sDeCAl4uoa6ZbMhBQKVwEdqBCAL1Yoa8/view?usp=sharing)