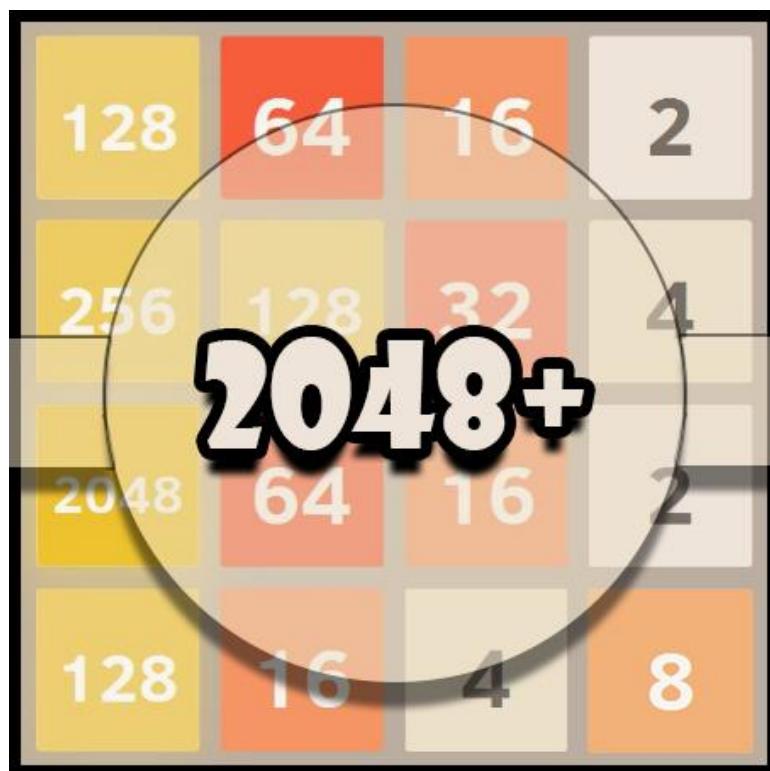


עבודת גמר

לקבלת תואר טכני תוכנה



בושא: שימוש במבנה מלאכותית לפתירת המשחק 2048.

מגיש: ארץ דרוטין

שמות המנהלים: מיכאל צ'רנובלסקי ואלון חיימוביץ

תש"ף

אפריל 2020

תוכן עניינים

3.....	הצעה נושא לפרויקט	•
6.....	לוחות זמנים פרויקט י"ג	•
7.....	מבוא	•
7.....	 ○ מטרת	
7.....	 ○ תיאור המערכת	
8.....	 ○ שפת תכנון ופירוט סביבת העבודה והכליים	
9.....	• מפרט תוכנה	
9.....	 ○ תיאור כללי	
10.....	 ○ ניקוד במשחק	
10.....	 ○ תחזזה במשחק	
10.....	 ○ מהלך במשחק	
11.....	 ○ ניסוח וניתוח הבעיה האלגוריתמית	
13.....	 ○ פיתוח הפתרון ויישומו	
19.....	 ○ תיאור אלגוריתמים	
25.....	 ○ מבנה נתונים	
26.....	 ○ תרשימים מחלקות – UML Class Diagram	•
27.....	 ○ תכנון	•
35.....	 ○ מדריך למשתמש	•
41.....	 ○ ביבליוגרפיה	•
42.....	 ○ נספחים	•

הצעת נושא לפרויקט

נושא הפרויקט: משחק מחשב אשר מדמה את המשחק 2048, שמטרתו היא "להציג" כמה שייתר זוגות מספרים ובעקבות כך להציג הגבואה ביותר תוך כדי ניסיון להשיג משכנת עם הערך 2048, אשר קובעת את סיום המשחק וניצחון המשמש. בנוסף, המשחק יגמר כאשר לא יהיה ניתן להציג עוד מספרים, והמשתמש יפסיד. המשחק יאפשר למשתמש לשחק במשחק בגרסתו הרגילה (משחקן יחיד) או מול המחשב.

גודל הלוח: 4x4

תזוזה במשחק – ביצוע תור:

חז למעלה: מעלה את המספרים בלוח כלפי מעלה.

חז למטה: מורד את המספרים בלוח כלפי מטה.

חז שמאלה: מזיז את המספרים בלוח כלפי צד שמאלה.

חז ימינה: מזיז את המספרים בלוח כלפי ימינה.

התרכשות המשחק:

בתחילת המשחק יופיעו 2 מספרים על המסך. המספרים יכולים להיות 2/2, והם לא בהכרח יהיו צמודים זה לזה. לאחר ביצוע כל תור (תזוזה של חז = תור), יתווסף ללוח מספר נוסף, שייהיה ברוב המקרים 2 אך לעיתים יהיה 4. בעת הצמדת זוג מספרים זהים, זוג המספרים "יעלם" ובמקומם יופיע מספר שהוא תוצאה החיבור של זוג המספרים. התוצאה תופיע בהתאם לכיוון החץ בעת הפעולה.

מצב ניצחון – מצב שבו הושגה משכנת שערכה הוא 2048.

מצב הפסד – מצב שבו לא ניתן לבצע מלים נוספים, כלומר הלוח מלא במספרים שלא יכולים ליצור צמדים עם הספרות שסבבם.

ניקוד המשחק:

הניקוד במשחק מתקבע באמצעות הצמדת זוגות של מספרים זהים. בעת תזוזה, יכולת לחיצה על אחד ממוקשי החיצים, יכולים להיווצר מספר זוגות באוטה העת. בנוסף, בעת היוצרים זוג מספרים, תופיע תוצאה החיבור של המספרים במקומות זוג המספרים (במיוחד שאלה החץ סימן), וכן על כר הניקוד של המשתמש יעלה בתוצאה החיבור שלהם.

המצבים השונים במשחק:

אחרי הצמדה:

חץ שמאלה + חץ מטה + חץ מטה



לפני הצמדת מספרים:

אמצע משחק – לפני הצמדה



מצב התחלתי:

לפני ביצוע מהלך כלשהו



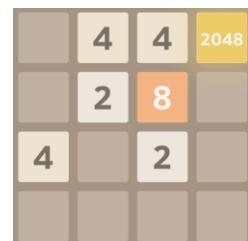
מצב הפסד:

אין מהלכים תקינים לביצוע



מצב ניצחון:

הגעה לאריך 2048



אסטרטגיות פעולה:

1. לנסות להוריד כמה שיותר צמדים באותו תור:

צמדים בבת אחת באמצעות לחיצה על החץ למטה וailo ניתן להוריד צמד אחד באמצעות לחיצה על החץ שמאלה, עדיף ללחוץ על החץ מטה, שכן כך נוכל "לנקות" את המסר, מה שיאפשר לנו "לשروع" יותר זמן.



2. עדיף להוריד צמד מספרים גדולים על פני צמד מספרים קטנים:

להוריד צמד של מספרים גדולים (כगון 128) לעומת צמד של מספרים קטנים יותר (כגון 2), לרוב כדאי את צמד המספרים הגדל יותר שכן קשה יותר להוריד אותו באופן כללי מאשר להוריד צמד מספרים קטנים (כי צריך לבצע יותר פעולות על מנת להגיע לכל מספר גבוה).



3. **כדי להיצמד לפינוקת הלוח:** כיוון שמספרים יופיעו באופן אוטומטי במקומות רנדומליים ברחבי המסר לאחר ביצוע כל פעולה, מומלץ לבחור באחת מפינוקות הלוח ולהיצמד אליה.



תיאור המערכת:

- **IMPLEMENTATION (Implementation):**
 - הצגת לוח המשחק בצורה ויזואלית אשר יעדכן לאחר כל תור.
 - ייצרת אণימציה בעת הצגת תורות המשתמש.
 - הצגת ניקוד המשתמש בכל עת והדפסת הودעה מתאימה כאשר המשחק מסתיים.
 - אפשרות למשחק חוזר, או לחלוּפִין חזרה למסך הפתיחה של המשחק.
- **SYSTEM (System):** ישום `bot` שישתמש באלגוריתם מורכב לצורך מימוש שחזור ממוחשב, אשר ישחק במשחק.

פיתוח המערכת:

פיתוח המערכת יעשה באמצעות שפת התכנות C# והספרייה MonoGame.

דרישות המערכת:

1. מימוש המשחק 2048 בגרסתו הרגילה.
2. מימוש אלגוריתם חכם ויעיל לצורך מימוש המשתמש ממוחשב.
3. מימוש גרסה "חديدة" של המשחק 2048, אשר מאפשר שימוש לשחק מול `bot` שיהווה שחזור ממוחשב.

עמדת פיתוח:

- מערכת הפעלה - Windows.
- סביבת עבודה – Visual Studio 2017 – .NET framework 4.X ומעלה.

דרישות מהמשתמש:

- מערכת הפעלה – Windows 7+.
- Framework נדרש.NET framework 4.X ומעלה.

לוחות זמנים פרויקט י"ג

מספר	תאריך	תיאור
1	09/11/2019	בחירה פרויקט הגשת נסח ראשוני (First Draft) – הצעת ותיאור הפרויקט (Use & PRD) (Cases)
2	17/11/2019	הגשת נסח סופי (final draft) - הצעת ותיאור הפרויקט (Use & PRD) (Cases) אישור ראשוני
3	01/12/2019	הגשה למשרד החינוך אישור
4	20/12/2019	IMPLEMENTATION GUI מרכזי של המשחק – מימוש לוח המשחק, מימוש פעולות במשחק, מימוש חוקי המשחק ועיצוב מסך המשחק (כולל אנימציות רלוונטיות).
5	10/01/2020	תחילת פיתוח האלגוריתם – הגדרת האלגוריתם החכם שנשתמש בו לצורך יצירה bot שיהווה השחקן הממוחשב.
6	26/01/2020	DOI
7	01/02/2019	המשך פיתוח האלגוריתם – תחילת מימוש האלגוריתם החכם במשחק שישמש בוט.
8	23/02/2020	הגשת DOI ביןימים – התקדמות הפיתוח עד כה
9	סוף מרץ 2020	שדרוג כללי של התוכנה – בדיקת מצבים קצה, תיקון בעיות, שיפור חווית משתמש, ועוד.
10	אפריל 2020	סיום פיתוח (Code Freeze)
11	אפריל 2020	בחינת מתכונת כולל תיק פרויקט
12	אפריל 2020	סיום תיקון שגיאות (Bug Freeze) – סופי
13	אפריל 2020	הגשת תיק מלאה של הפרויקט סופי
14	אפריל – מאי 2020	הגנה על עבודות הגמר – פנימית – סופית
15	אפריל – מאי 2020	הגנה על עבודות הגמר – חיצונית

מבוא:

מטרה:

פיתוח המשחק 2048 בשפת התכנות #C framework שפותח ע"י מייקروسופט עבור פיתוח משחקי לSYSTEM.NET. הנקרא MonoGame ופיתוח בינה מלאכותית לשחקן ממוחשב תוך שימוש באלגוריתמים ובמבנה נתונים מורכבים ויעילים. אחת המטרות המרכזיות שלו היא ללמד כלים לשיפור חווית המשתמש אשר ישפרו את יכולותיו כמפתח ויעזרו לו בעתיד.

תיאור המערכת:

המשחק המקורי 2048 הינו משחק לשחקן יחיד הפועל בדפדפן וטלפונים. הוא נוצר ב-2014 ע"י גבריאלה צ'ירולו וזכה להצלחה רבה בעקבות היוטו כה פשוט ובמקביל כה מסובך. מטרת המשחק היא להחליק אריחים מנומספרים על גבי לוח המשחק בכך לשלב אותם וליצור אריח שערכו 2048.

המשחק בניו מלוח משਬצות בגודל 4X4 עם אריחים מנומספרים בצבעים שונים. על מנת להציג את האריחים, השחקן משתמש בחצי המקלדת (או טלפון – מחליק את האריחים לכיוון הרצוי). אם 2 אריחים בעלי אותו מספר מתנגשים תוך כדי התנועה, הם יתמזגו לאריח עם ערך השווה לסכום של שני האריחים שהתנגשו. במקביל, בכל תור של המשתמש יתווסף אריח חדש עם ערך 2 או 4 למשבצת ריקה אקראיית על גבי הלוח.

המשחק **יסתאים ב涅צחון** כאשר המשתמש יופיע אריח בעל הערך 2048 על גבי הלוח, או **יסתאים בהפסד** אם המשתמש לא יכול לבצע עוד מהלכים (כיוון שהלוח מלא באריחים שלא יכולים להתמזג).

במערכת שלו יש 3 אפשרויות למשתמש והן:

1. משחק רגיל, אשר פועל בצורה זהה למשחק המקורי 2048.
2. משחק מול ממוחשב, אשר מאפשר לשחקן לשחק מול ממוחשב.
3. משחק ממוחשב, אשר מאפשר לשחקן לצפות במוחשב משחק ובכך להפיק לקחים ואסטרטגיות ממשחק המוחשב.

מראה לוח המשחק:



שפת התכנות ופירוט סביבת העבודה והכליים:

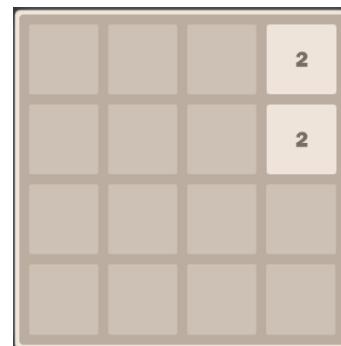
פיתוח המערכת נעשה בשפת התוכנה C# באמצעות framework שפותח ע"י מיקרוסופט עבור פיתוח משחקים בסביבת NET. הנקרא MonoGame (קיים ב-X.NET framework 4.0, Microsoft Visual Studio 2017 ועוד), ובמהלכו הוצע שימוש בכלים סביבת העבודה NET. מציעה.

מפורט תוכנה:

תיאור כללי:

המשחק 2048 הינו משחק חשיבה המשוחק על לוח בגודל 4x4 והוא מיועד לשחקן יחיד כאשר מטרת המשחק היא להגיע לאירוע בעל הערך 2048. המשחק מתחליל כאשר 2 אריחים בעלי הערך 2 או 4 (כאשר יש 90% לאל אריך להיות 2 ו-10% להיות 4) מופיעים במקומות רנדומלי על גבי לוח המשחק. המשחק מסתיים כאשר המשחק מגיע לאירוע עם הערך 2048 (נצחון) או כאשר לשחקן אין עוד מהלכים שהוא יכול לבצע (הפסד).

מראה הלוח בתחילת המשחק:



מראה הלוח בסיום המשחק (נצחון):



מראה הלוח בסיום המשחק (הפסד):



ניקוד במשחק:

הניקוד במשחק מתאפשר באמצעות הצמדת זוגות של מספרים זהים. בעת תזוזה, יכולם לחייב על אחד ממקשי החיצים, יכולם להיזכר מספר זוגות באותו העת. בנוסף, בעת הייזכרות זוג מספרים, תופיע תוצאה החיבור של המספרים באותה זוג המספרים (במיוקם שאלוי החץ סימן), וכן על כך הנקוד של המשתמש יעלה בתוצאה החיבור שלהם.

תזוזה במשחק:

התזוזה מתבצעת באמצעות חיצי המקלדת באופן הבא:

חץ מעלה: מזיז את הארכחים בלוח כלפי מעלה.

חץ מטה: מזיז את הארכחים בלוח כלפי מטה.

חץ שמאל: מזיז את הארכחים בלוח כלפי צד שמאל.

חץ ימינה: מזיז את הארכחים בלוח כלפי צד ימין.

מהלך במשחק:

בתחילת המשחק, יופיעו 2 ארכחים ממושפרים במיקום אקראי על גבי הלוח. ערכי הארכחים יכולים להיות 2 או 4, כאשר הסיכויים של כל ארכח לקבל את הערך 2 הם 90% והסיכויים לקבלת הערך 4 הם 10%. לאחר ביצוע כל תור (תזוזה), יתווסף ללוח ארכח ממושפר נוסף ע"פ האופן שצוין לעיל.

בעת ביצוע תזוזה, 2 ארכחים עשויים להתמזג (Fuse) לארכח אחד (שיופיע בהתאם לכיוון החץ שגרם להתמזגות הארכחים) במידה וערכם זהה והוא צמודים זה לזה. ערך הארכח שיתקיים מהמתזגות שני הארכחים יהיה סכום 2 הארכחים שהתמזגו.

מצב ניצחון – מצב שבו הושג ארכח עם הערך 2048.

מצב הפסד – מצב שבו לא ניתן לבצע מהלכים נוספים, ככלומר הלוח מלא בארכחים ואין זוג ארכחים שיכולים להתמזג.

השחקן צובר ניקוד המשחק ע"י מיזוג ארכחים. בעבור מיזוג של כל זוג ארכחים, השחקן יקבל תוספת ניקוד בהתאם לסכום זוג הארכחים שהתמזג, כאשר בתור אחד ניתן לגרום למיזוג של כמה זוגות ארכחים במקביל.

ניסוח וניתוח הביעות האלגוריתמיות:

בעיה אלגוריתמית: תיאום מערכת לבחירות השחקן

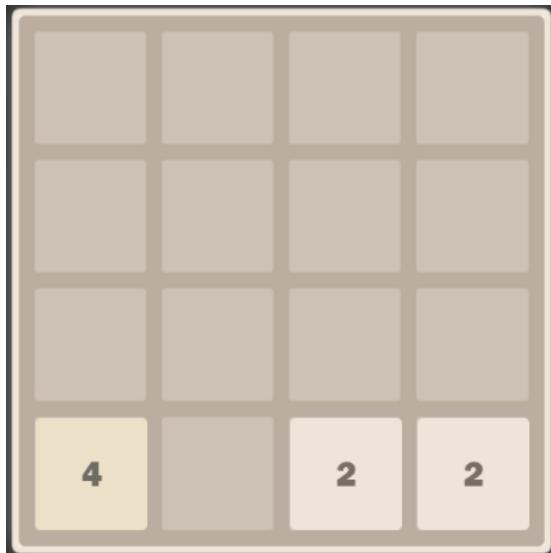
הבעיה האלגוריתמית שעומדת מאחורי תיאום מערכת לבחירת השחקן נובעת ממה צורך לאפשר למשתמש לבחר דרך מבחן מספר דרכים שונות לשחק במשחק: משחק רגיל, משחק של שחקן ממוחשב ומשחק של שחקן אנושי נגד שחקן ממוחשב. בהתאם לבחירת המשתמש, علينا להפעיל פעולות שונות אשר יתאמו בין בחירת המשתמש לבין מה שצריך להציג לו. לדוגמה, אם השחקן בחר לשחק נגד שחקן ממוחשב, אז על המערכת הראשית לפועל כangkan ממוחשב ולאחר מכן לאפשר לשחקן האנושי לשחק ולנסות להביס את השחקן ממוחשב.

מעבר לסוג המשחק שנבחר ע"י השחקן, המערכת צריכה להתחשב בבחירהו בוגר לרמת הקושי של האלגוריתמים ולסוג האלגוריתם הנבחר. לדוגמה, כאשר השחקן מעוניין לצפות באלגוריתם (AI) משחק, עליו לבחור את סוג האלגוריתם (AI / ExpectiMax AI). (Random AI /

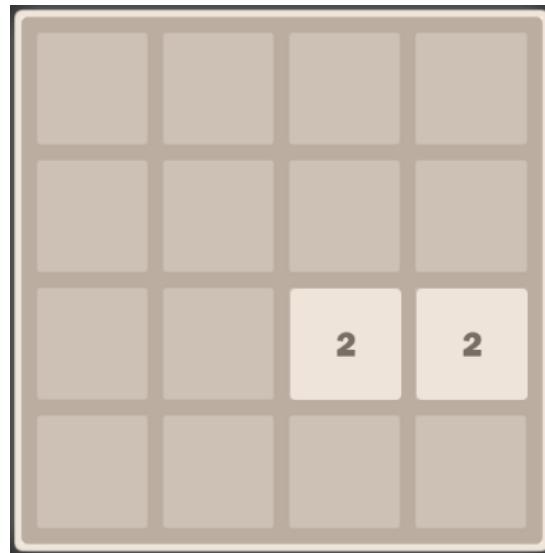
בעיה אלגוריתמית: בדיקת תקינות מהלך

הבעיה האלגוריתמית שעומדת מאחורי בדיקת תקינות מהלך היא לזהות אלו תזוזות ישפיעו על המשחק ואלו לא בכל רגע נתון. המשחק יכול לזרז ארך ורך באמצעות חיצ'י המקלדת כאשר כל חץ מסמל תזוזה שונה אשר משפיעה על היזזת כל הארכיים על הלוח כלפיון החץ. במקרה, כל מהלך נדרש לתקן פרט למקרים בהם התזוזה לא משפיעה על לוח המשחק. לדוגמה, ניקח את 2 לוחות המשחק הבאים:

לוח ב'



לוח א'



ניתן לראות כי בלוח א' המשחק יכול לנوع לכל כיוון (מעלה, מטה, ימינה או שמאלה), וזאת מכיוון שכל תזוזה תגרום לשינוי בלוח, אשר בעקבותיו גם יתווסף אריך חדש לשחקן.

מנגד, בלוח ב' המשחק יכול לנوع לכל הכיוונים פרט למטה (מעלה, ימינה או שמאלה). זאת מכיוון שתזוזה למטה לא תשפיע על הלוח, משום שככל הארכיים בזמן הנutanן כבר צמודים למטה ולכן תזוזה למטה לא תשפיע עליהם.

בעיה אלגוריתמית: התמצגות אריכים

כיוון שבמשחק 2048, אריכים בעלי ערך זהה אשר נפגשים זה עם זה יכולים להתמזג לאריך אחד, הייתי צריך לחשב על דרך שתאפשר לי ללהות מתי זוג אריכים צריכים להתמזג לאריך אחד שערכו שווה לסכום 2 האריכים. אם נתבונן שוב בלוחות א' וב' שהציגו בעמוד הקודם, נראה כי בשנייהם יש 2 זוגות אריכים בעלי ערך זהה (2 הזוגות בעלי הערך 2) אשר יכולים וצריכים להתמזג לאריך אחד שערכו יהי 4 במידה והמשתמש יבחר לנوع לצד ימין או שמאלי (כיוון שתנועה לאחד מהכיווןים הללו תגרום להתמצגות האריכים). על מנת להתמודד בעיה זו, הייתי צריך לבדוק מתי והאם כל זוג אריכים בלוח יכולים להתמזג, ומתי לא. בעקבות כל מהלך מספר זוגות אריכים יכולים להתמזג במקביל.

בעיה אלגוריתמית: אלגוריתם לשחקן ממוחשב

את הדרכים הנפוצות ביותר לחישוב מהלך המשחק לוח עם "מזל" (כלומר, בעקבות כל מהלך יש הסתברות לקבל תוצאהRndומלית ולא חד משמעותית) הוא באמצעות האלגוריתם מונטה קרלו. עם זאת, לאחר מחקר מעמיק וניתוח תוצאות שפורסמו ברחבי האינטרנט, הגעתנו למסקנה שהאלגוריתם הייעיל והטוב ביותר לצורך פתרון שחזור ממוחשב המשחק 2048 הוא האלגוריתם ExpectiMax. אלגוריתם זה מתבסס על האלגוריתם המוכר MiniMax, אך מעט מסווך יותר ומותאם למשחקים עם "מזל", כמו המשחק 2048. בעוד שבדבוק MiniMax פעם בוחרים את הצומת בעלת הערך המינימלי, ופעם את הצומת בעלת הערך המקסימלי עד שmaguiim לשורש, באlgorigthm ExpectiMax בוחרים בכל פעם את ממוצע הבנים של כל צומת, ובסיוף כאשר maguiim לשורש בוחרים את הבן שערכו הוא הגבוה ביותר.

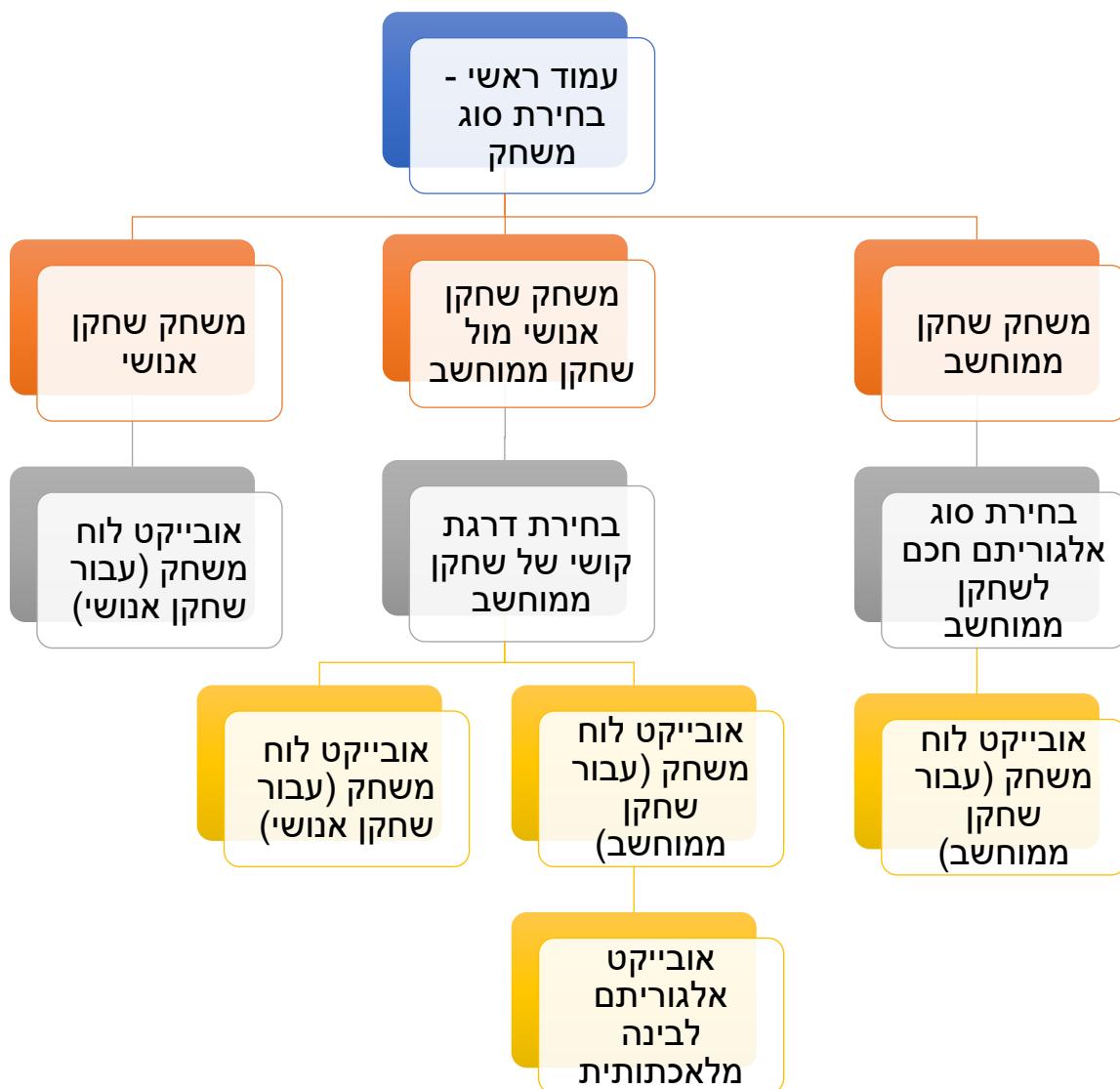
פיתוח הפתרון ויישומו:

פיתוח פתרון לבעה אלגוריתמית: תיאום מערכת לבחירות השחקן

במערכת המשחק ישן 3 אפשרויות מרכזיות:

- משחק רגיל (ע"פ כללי המשחק 2048).
- משחק בו שחקן אנושי מתמודד מול שחקן ממוחשב.
- משחק בו השחקן הממוחשב מנסה להגיע לתוצאה הגדולה ביותר (ובכך מאפשר לשחקן האנושי ללמידה ולסגל לעצמו אסטרטגיות משחק ע"י צפייה בו).

על המערכת לייצר תיאום בין בחירות המשתמש לבין המתרחש במערכת, לכן לשם פתרון בעיה זו החלטתי ליצור מחלקה אחת ראשית אשר מטרתה תהיה לנוהל את המעברים בין "הסצינות" השונות במשחק. למחלקה זו קוראים SceneManager, והוא בונה בצורה גנרייה מאוד בצד' שנייתן יהיה להוסיף סצינות למשחק בקלות. התרשימים הבא מתאר את צורת פעילות מחלקה Scene Manager באופן כללי:



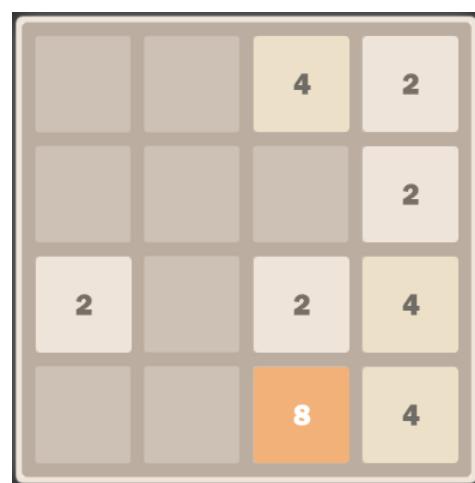
בזכות המחלקה `SceneManager` אשר עוזרת לנו לנהל את הכל, ניתן לראות כיצד בעקבות כל אחת מהخيارات המשמש נוצר אובייקט / נטען עמוד בהתאם לבחירה. בהתאם לבחירות המשמש המשחק הרלוונטי יתחייב. לדוגמה, אם המשתמש בחר במשחק רג'יל – יוצר לוח משחק ויתבען עמוד המשחק הרג'יל.

פיתוח פתרון לבעה אלגוריתמית: בדיקת תקינות מהלך

כאשר המשתמש לוחץ על אחד מחיצ'י המקלדת בזמן משחק, מטבחצת בדיקת תקינות מהלך שביצע. במידה והמהלך שבייצע עתיד להשפיע על הלוח, המהלך יחשב לתקן, אחרת, המהלך יחשב לא נכון. בעקבות זאת, כתבתת אלגוריתם שבודק את כל השפעת התזוזה שנבחרה ע"י המשתמש על לוח המשחק, ובהתאם לתזוזה שיבצע המשתמש האלגוריתם ייחזר האם הלוח השתנה או לא. על מנת להזוהה אם המהלך שבייצע המשתמש אכן או לא, נעזרתי בלוולאות ובמשתנה מסווג `bool` שמאוחסל עם הערך `false` לבדיקה האם בעקבות המהלך שהמשתמש ביצע הלוח השתנה או לא. באמצעות הלולאות עברתי על גבי כל הארכיהם בלוח ובדקתי כיצד הלוח מושפע מתזוזות המשתמש. במידה ובסיום המעבר באמצעות הלולאות על הלוח ערכו של המשתנה מסווג `bool` הפך ל`true`. במידה ובסיום המעבר באמצעות הלולאות על הלוח ערכו של המשתנה מסווג `bool` לא השתנה, אז הלוח לא הושפע כתוצאה מההלך וההלך נחשב לא נכון (ולכן גם לא נספר כמהלך).

פיתוח פתרון לבעה אלגוריתמית: התמצגות אריכים

על מנת להתמודד עם בעיה אלגוריתמית זו, הייתה צריכה לחשוב על דרך להזוהה מתי זוג אריכים מתמצג, ולהשתמש בדרך זו על גבי כל הלוח כיוון שמספר זוגות אריכים יכולים להתמצג בעקבות מהלך אחד של המשתמש. לדוגמה, נתבונן בלוח הבא:



בעקבות תזוזה מטה/מעלה, קיבל ש2 זוגות אריכים יתמצגו. זוג האריכים שערכם 2 בצד ימין למטה יתמצג לאריך שערכו 4 וזוג האריכים שערכם 4 בצד ימין למטה יתמצג לאריך שערכו 8.

בעקבות זאת, פיתחתי דרך ראשית להזוהה מתי זוג אריכים מתמצג לאריך אחד, ואז נעזרתי בלולאה להזוהה את כל זוגות האריכים שמתמצגים בעקבות התזוזה.

פיתוח פתרון לבעיה אלגוריתמית: אלגוריתם לשחקן הממוחשב

הבעיה האלגוריתמית המרכזית בפרויקט היא פיתוח הבינה המלאכותית לצורכי מימוש שחקן ממוחשב, ועל כן היא גם הייתה המסובכת ביותר. על מנת להתמודד אליה, ראשית קראתי מאמריהם שעוסקים באסטרטגיות למשחק 2048. העעת למסקנה שהאסטרטגיה הטובה ביותר במשחק 2048 היא לשמר על הערך הנוכחי בערך הגבוה ביותר בצד שמאל למעלה של הלוח, ובכל פעם להציג אליו את שאר הארכיחס בעלי הערכים הגבוהים ביותר.

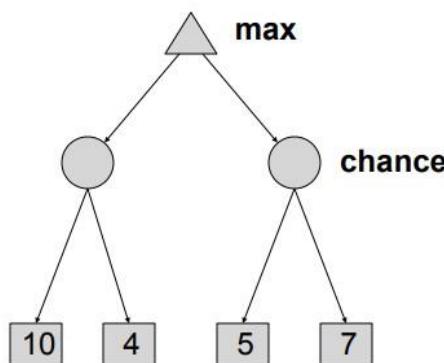
לדוגמא – ניתן לראות כיצד בלוח הבא הערך 256 (הגבוה ביותר) ממוקם בצד שמאל למעלה כאשר לצדיו ממוקם המספר הגבוה ביותר הבא בתו:



באמצעות חישובים מתמטיים, הגיעתי לשיטה שבאמצעות מימושה עם האלגוריתם ExpectiMax ניתן יהיה לשמר על הערך הנוכחי בערך הגבוה ביותר לאורך הרוב המוחלט של המשחק בצד שמאל למעלה, מה שיאפשר לשחקן הממוחשב להגיע לתוצאות גבוהות יותר.

להלן אסביר כיצד האלגוריתם ExpectiMax עובד באופן כללי, ולאחר מכן אסביר כיצד הוא פועל בפרויקט עצמו. האלגוריתם ExpectiMax רושית מחשב את מוצע האופציות האפשריות לפעולה מסוימת, לאחר מכן עולה "שלב" בעז ומבצע את אותו הדבר עד שהוא מגיע לשורש, כאשר עבורי הוא בוחר את הבן שערכו גובה יותר. למשל, בעצם בוחרים בכל פעם את המוצע (chance) ובשורש בוחרים את המקסימום (max).

לצורך הסבר ברור יותר, נניח שיש לנו את העץ הבא:



נניח שעבור סיטואציה מסוימת – החלקת המסר ימינה, יש 2 דרכי פעולה אפשריות, ובעקבות שתיהן עשויה להתקבל תוצאה שונה.

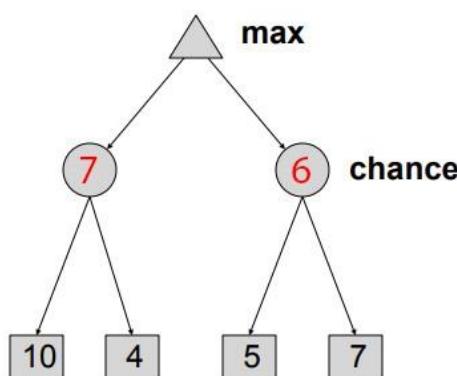
אפשריה א: החלקת המסר ימינה שבעקבותיה זוג מספרים יתמזג ויתווסף אריך עם הערך 2.

אפשריה ב: החלקת המסר ימינה שבעקבותיה זוג מספרים יתמזג ויתווסף אריך עם הערך 4.

- על מנת לא לסביר את הדוגמא, לא ATIICHOS פה לכמה מקומות הרנדומליים האופציונליים למקום ערך 2 או לכמה מקומות הרנדומליים האופציונליים למקום ערך 4 וכן שיש 2 אפשרויות בלבד למקום כל אחד מהם.

עת, כאשר יש לנו את הערכים (4, 10) ו(5, 7), אנחנו יכולים לחשב את ערך ההוריהם שלהם, שהוא הממוצע של כל זוג מספרים. לכן, בשלב הבא של ריצת האלגוריתם ExpectiMax על העץ הנוכחי, נקבל את הערך 7 בצד שמאל ואת הערך 6 בצד ימין.

עת, העץ נראה如下:

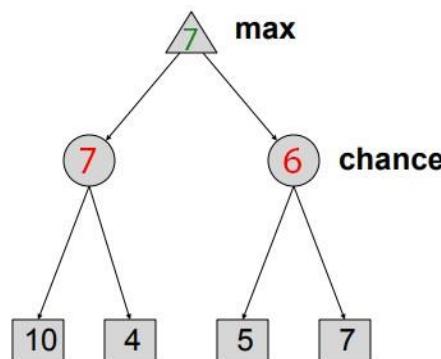


עת, הגיענו לשלב האחרון בבניית העץ, והוא בחירת הבן בעל הערך המקסימלי.

- ברוב המקרים יהיו עוד שלבים לפני שנגיע לשורש (תלו בעומק העץ), אך במקרה זה נניח שבשלב זה נגיע לשורש ולכן נפעל בהתאם לכך.

הריצת האלגוריתם ExpectiMax עבור העץ שלעיל, נקבל שערך של שורש העץ יהיה 7, כיוון שלשורש יש שני בניים, וערךם הוא 6 ו-7, ו-7 גדול מ-6 (כאמור, בוחרים את הבן בעל הערך המקסימלי).

לכן, בסיום ריצת האלגוריתם, נקבל את העץ הבא:



איך ניתן לישם את האלגוריתם ExpectiMax בפרויקט?

ראשית, צריך להבין מהן האופציות שעומדות פנויו בכל מהלך ולהחליט איך כדאי למשתמש באלו. האלגוריתם קר שבסומו של דבר, השחקן הממוחשב יגיע לביצועים מקסימליים. כפי שהסבירתי מוקדם יותר, האסטרטגיה הטובה ביותר למשחק 2048 היא לשמר את האրיך בעל הערך המקסימלי בצד שמאל למעלה של הלוח. על מנת לשמר על קר, בעוד שהוא עוברם על כמה אופציות גדולות מאוד, ניעזר בחישובים מתמטיים שיאפשרו לנו לשמר על האריך בעל הערך המקסימלי תמיד בצד שמאל למעלה.

איך ניתן לניקוד למחלכים במהלך?

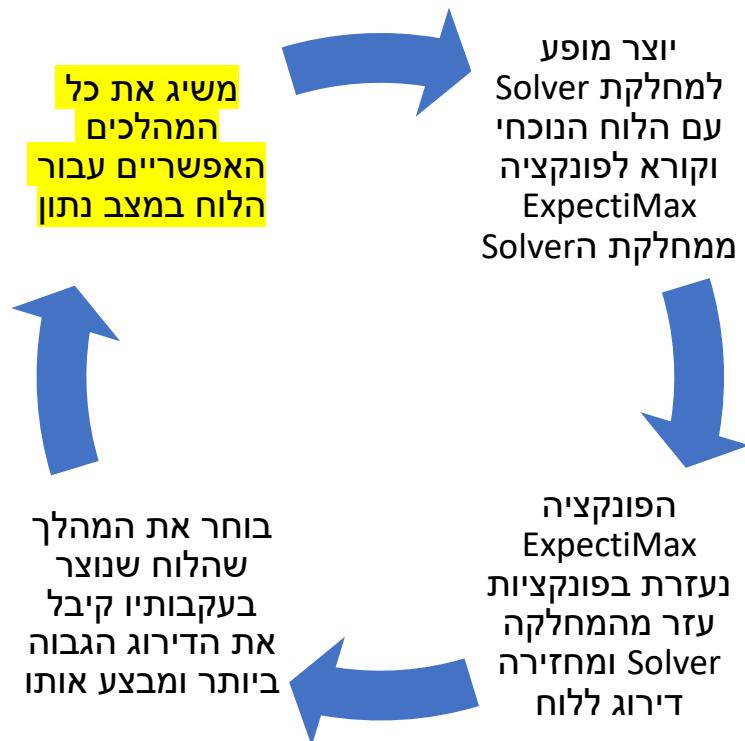
על מנת לשמר על האריך בעל הערך הגבוה ביותר בלוח בצד שמאל למעלה של הלוח, החלטתי למסקל את הארכיהם ע"פ "קדימותם" בלוח וערכם. על מנת לבצע זאת, השתמשתי בנוסחה הבאה: $y = \log_2\left(\frac{1}{4} * x\right)$, כאשר $x = \text{Tile's Value}$, $y = \text{Tile's Index}$. האינדקס של האריך נקבע ע"פ מיקומו בלוח והוא נע בין 0 ל-15, וערךו של האריך הוא הערך שהאריך מחזיק באותו הזמן. לדוגמה, אם ארייך הוא בעל ערך 8 והוא ממוקם באינדקס 0,0 (האריך בצד שמאל למעלה של הלוח), אז נתיחס אליו כאל $\log_2\left(\frac{1}{4} * 8\right)$, כלומר, כל ערך 3.

במציאות שיטה זו, ניתן להבטיח שהאריך בעל הערך הגדול ביותר יהיה בצד שמאל למעלה **ברוב המוחלט של המחלכים**, כאשר יהיו צמודים אליו (מאחד הצדדים) שאר הארכיהם בעלי הערך הגבוה ביותר, וכך שבכל תור הערך שהכי משפיע על התוצאה הסופית הוא ערך האריך באינדקס 0,0, כלומר בצד שמאל למעלה בלוח.

איך נקבע המהלך שצורך לבצע?

על מנת לקבוע איזה מהלך הוא הטוב ביותר ביותר ברגע נתון, צריך לדעת מה השפעת המהלך על הלוח במצב הנוכחי (לפניהם כניסה לעומק). במידה והמהלך לא משפיע על הלוח, המהלך מסור מרשימה המחלכים האפשריים בעבור התור הנוכחי. במידה ובזמן ריצת האלגוריתם (עם עומק) מתגלה שהשחקן הממוחשב עשוי להפסיד, אותו מהלך יקבל את העונש הגורע ביותר – והוא את ערך `double.MinValue`, כלומר הערך הנמוך ביותר שניתן לייצג באמצעות משתנה מסוימת. סוג כיוון שמטרת השחקן הממוחשב היא לשרוד כמה שיותר זמן ולהגיע לניקוד הגבוה ביותר.

כעת, לאחר שיש לנו כבר את לוחות המשחק שנוצרים בעקבות ביצוע כל מהלך, علينا ליצור סר של $Z * 2 - 32$ לוחות כאשר Z מייצג את כמות הארכים המאותחלים בלוח. הסיבה לנובעת מכך שאנו לא יודעים האם לאחר יתווסף ארייך שערכו 2 או שיתווסף ארייך שערכו 4, ומעבר לכך אנחנו לא יודעים באיזה מיקום יתווסף האריך. לכן, במקרה הגורע ביותר, ניאלץ ליצור 30 לוחות (אם יש ארייך 1 בלבד על הלוח לאחר ביצוע המהלך, ככלומר היה מיזוג של 2 ארכיים בעקבות המהלך הנוכחי), כאשר 15 מהלוחות יכולים להיות עם ערך 2 במיקום רנדומלי ו-15 מהלוחות יכולים להיות עם ערך 4 במיקום רנדומלי על הלוח. את יצירת הלוחות ניאלץ לבצע בעבור כל מהלך על מנת לקבוע את דירוגו. כמות הלוחות שיוצרים היא כפונקציה שונה בהתאם לעומק שיבחר בעבור האלגוריתם, כאשר האלגוריתם פועל בצורה היעילה ביותר במידת האפשר מבל' לפגוע ביציאותו. בהתאם לדיאגרמת העץ שמוסברת לעיל בקשר לפעולותExpectiMax, קר גם האלגוריתם לשחקן ממוחשב פועל. בהתאם לעומק הנבחר, האלגוריתם יבחר את מוצע הבנים של כל צומת ולבסוף עבור שורש העץ הוא יבחר את הבן שערכו הוא הגבוה ביותר, ויבצע את המהלך מבין 4 המחלכים שערכו הוא הגבוה ביותר.

התרשים הבא מייצג את האופן שבו הAI פועל בפרויקט:**מסומן בצהוב – נקודת מפתחה לתרשים הזרימה.****הסביר מילוי לתרשים הזרימה:**

בפועל, בעבר כל פעולה שהAI מבצע, הAI ראשית מייצר עד 4 לוחות כאשר כל לוח מייצג את הלוח שיוצג בעקבות ביצוע תזוזה לכיוון כלשהו. הלוחות אליהם הAI יתייחס הם הלוחות השונים מהלוח הנוכחי **בלבד**, וזאת לפני שיתווסף אליהם אורך במיקום רנדומלי. לאחר מכן, בעבר כל אחד מהלוחות, הAI יקרא לפונקציה ExpectiMax שמודדרת במחלקה Solver על מנת לדרג את כל אחד מלוחות המשחק שנוצרו, ובסיום הAI **בוחר ומבצע** את המהלך שהלוח שנוצר בעקבותיו קיבל את הדירוג הגבוה ביותר, כיוון שההלך זה הוא הכספי ביותר.

תיאור אלגוריתם:

תיאור אלגוריתם: תיאום מערכת לבחירות השחקן | סיבוכיות: לא רלוונטי

האלגוריתם הוא מחלקת הנקראת `SceneManager` ותפקידה הוא לנוהל את "הסצינות" השונות במהלך המשחק. המחלקת משתמש פונקציות מההפקט `framework` שנקרו `MonoGame` שפותח ע"י מייקרוסופט לצורך פיתוח משחקים בסביבת `.NET`. והfonקציות הללו אחראיות לטעינה, הצגת קליטת מידע.

- קיים enum הנקרא `DrawState`, ומטרתו היא לעזור למחלקת לקבע איזה "סצינה" צריך לצייר ברגע נתון.
- חלק מהfonקציות נקראות באופן אוטומטי כחלק מהגדרות framework וחלק נקראות באופן אינטידואלי כאשר יש בהן צורך. הfonקציות `Draw`, `Update` נקראות פעמי אחת ע"י הגדרת framework ולאחר מכן הן קוראות לעצמן באופן רקורסיבי לאורך ריצת המשחק.

להלן אציג את האלגוריתם לתיאום מערכת לבחירות השחקן ממהמחלקה `SceneManager`:

1. **אתחל סצינה – InitializeScene :**

a. צור "ריבועים" (Rectangles) רלוונטיים לסצינה.

2. **טען סצינה – LoadContent :**

a.טען מידע רלוונטי לסצינה הנוכחי.

3. **עדכן מסך – Update :**

a. אם הסצינה לצייר כתה היא `Menu` אז:

i. אם הceptor לבחירת משחק שחקן אנושי נלחץ אז:

1. בטל סצינה.

2. עדכן משתנה לקביעת סצינה הנוכחי לשחקן אנושי.

3. עדכן ערך דגל לאמת.

ii. אם הceptor לבחירת משחק שחקן מול שחזור מחשב נלחץ אז:

1. בטל סצינה.

2. עדכן משתנה לקביעת סצינה הנוכחי לשחקן מול מחשב.

3. עדכן ערך דגל לאמת.

iii. בדוק אם הceptor לבחירת משחק שחקן מחשב נלחץ.

1. בטל סצינה.

2. עדכן משתנה לקביעת סצינה הנוכחי לשחקן ממוחשב.

3. עדכן ערך דגל לאמת.

iv. אם אחד הceptors נלחץ (דגל = אמת) אז:

1. אתחל סצינה.

2. טען סצינה.

3. נעל את העכבר.

v. אחרת:

1. בטל את נעלת העכבר.

b. אם הסצינה לצייר כתה היא `PlayerMode` אז:

- . בדוק כמה delay נותר עד שהמשתמש יכול לבצע את התוור הבא.
- ii. אם הcptor לתחילה משחק מחדש נלחץ אז:
 - 1. אם העכבר לא נעל אז:
 - a. צור לוח משחק חדש.
 - b. נעל את העכבר.
 - iii. אם הcptor לחזרה נלחץ אז:
 - 1. אם העכבר לא נעל אז:
 - a. בטל סצינה.
 - b. עדכן משתנה לקביעת סצינה נכון לסתינת Menu.
 - c. אתחל סצינה.
 - d. טען סצינה.
 - e. נעל את העכבר.
- iv. אחרת:
 - 1. בטל את נעלית העכבר.
- c. אם הסצינה לצייר כתת היא ChooseDifficulty אז:
 - i. אם הcptor למצב קל נבחר אז:
 - 1. תן למשתנה depth את הערך 0.
 - 2. עדכן ערך דגל לאמת.
 - ii. אם הcptor למצב קשה נבחר אז:
 - 1. תן למשתנה depth את הערך 2.
 - 2. עדכן ערך דגל לאמת.
 - iii. אם הcptor למצב מוצע נבחר אז:
 - 1. תן למשתנה depth את הערך 4.
 - 2. עדכן ערך דגל לאמת.
- iv. אם אחד הcptורים נלחץ (דגל = אמת) אז:
 - 1. בטל סצינה.
 - 2. צור לוח משחק חדש (עבור השחקן האנושי).
 - 3. צור לוח משחק חדש עם depth Um AIState.Exi depth (עבור השחקן הממוחשב).
 - 4. עדכן משתנה לקביעת סצינה נכון לסתינת AiPlay.
 - 5. אתחל סצינה.
 - 6. טען סצינה.
 - 7. נעל את העכבר.
- v. אחרת:
 - vi. בטל את נעלית העכבר.
- d. אם הסצינה לצייר כתת היא PlayVsAi אז:
 - i. אם המצב לצייר ע"פ לוח המשחק של המחשב הוא הפסד אז:
 - 1. בדוק כמה delay נותר עד שהמשתמש יכול לבצע את התוור הבא.
 - 2. אם הcptor לתחילה משחק מחדש נלחץ אז:
 - a. אם העכבר לא נעל אז:
 - b. צור לוח משחק חדש (עבור השחקן).

- c. צור לוח משחק חדש עם depth AIState ו-Zהים לאלו שיש ללוח משחק של המחשב כרגע.
- 3. אם הcptor לחזרה נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - i. בטל סצינה.
 - ii. עדכן משתנה לקביעת סצינה נוכחית לשינית Menu.
 - iii. אתחל סצינה.
 - iv. טען סצינה.
 - v. נעל את העכבר.
 - b. אחרת:
 - i. בטל את נעלית העכבר.
- ii. אחרת:
 - 1. אם עברו 0.02 שניות מאז התור הקודם שבוצע ע"י הוא אז:
 - a. אם הוא של לוח שחזור המחשב הוא Ex אז:
 - i. קרא לפונקציה ()RandomAI של לוח שחזור המחשב.
 - ii. אם הוא של לוח שחזור המחשב הוא Ex אז:
 - . קרא לפונקציה ()ExpectiMaxAI של לוח שחזור המחשב.
 - b. בדוק כמה delay נותר עד שהמחשב יכול להציג את האנימציה תזוזה של הארכחים.
 - 3. אם הcptor לתחילת משחק חדש מוחדר נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - b. צור לוח משחק חדש (עבור השחקן).
 - c. צור לוח משחק חדש עם depth AIState ו-Zהים לאלו שיש ללוח משחק של המחשב כרגע.
- 4. אם הcptor לחזרה נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - i. בטל סצינה.
 - ii. עדכן משתנה לקביעת סצינה נוכחית לשינית Menu.
 - iii. אתחל סצינה.
 - iv. טען סצינה.
 - v. נעל את העכבר.
 - b. אחרת:
 - i. בטל את נעלית העכבר.
- e. אם הסצינה לצירור כרגע היא ChooseAI אז:
 - i. אם הcptor לבחירת Randi נלחץ אז:
 - 1. בטל סצינה.
 - 2. עדכן משתנה לקביעת סצינה נוכחית לשינית AiWatch.
 - 3. צור לוח משחק חדש עם ה AI Randi.

4. עדכן ערך דגל לאמת.
- ii. אם הceptor לביית Ex נלחץ אז:
 1. בטל סצינה.
 2. עדכן משתנה לקביעת סצינה נוכחת לשכינת AiWatch.
 3. צור לוח משחק חדש עם הAI Ex ועומק 4.
 4. עדכן ערך דגל לאמת.
- iii. אם אחד הceptors נלחץ (דגל = אמת) אז:
 1. אתחל סצינה.
 2. טען סצינה.
 3. נעל את העבר.
- iv. אחרת:
 1. בטל את נעילת העבר.
- f. אם עברו 0.02 שניות מאז התור הקודם שבוצע ע"י הוא אז:
 - i. אם הוא של לוח שחזור המחשב הוא Randi אז:
 1. קרא לפונקציה ()AIrandom של לוח שחזור המחשב.
 - ii. אם הוא של לוח שחזור המחשב הוא Ex אז:
 1. קרא לפונקציה ()AIexpectimax של לוח שחזור המחשב.
 - iii. בדוק כמה delay נותר עד שהמחשב יכול להציג את האנימציה תזוזה של הארכחים.
 - iv. אם הceptor לתחילת משחק חדש נלחץ אז:
 1. אם העבר לא נעל אז:
 - a. צור לוח משחק חדש עם אותו AIstate וdepthi זהים לאלו שיש ללוח המשחק כרגע.
 - b. נעל את העבר.
 - v. אם הceptor לחזרה נלחץ אז:
 1. אם העבר לא נעל אז:
 - a. בטל סצינה.
 - b. עדכן משתנה לקביעת סצינה נוכחת לשכינת Menu.
 - c. אתחל סצינה.
 - d. טען סצינה.
 - e. נעל את העבר.
 - v. אחרת:
 1. בטל את נעילת העבר.

4. ציר סצינה – Draw:

- a. אם הסצינה לציר כרגע היא Menu אז:
 - i. ציר את כלtextures והרלוונטיים לשכינה.
- b. אם הסצינה לציר כרגע היא PlayerMode אז:
 - i. ציר את כלtextures והרלוונטיים לשכינה.
 - ii. אם מצב לוח המשחק מורה על ניצחון אז:
 1. ציר אתtexutre והרלוונטיים למצב ניצחון.
 - iii. אם מצב לוח המשחק מורה על הפסד אז:

1. צייר את ה `rectangle` וה `texture` הרלוונטיים למצב הפסד.
- c. אם הסצינה לציר כרגע היא `ChooseDifficulty` אז:
 - i. צייר את הרקע לסצינה (שמכל בתוכו את כל מראה הסצינה).
 - d. אם הסצינה לציר כרגע היא `PlayVsAi` אז:
 - e. צייר את כל ה `textures` וה `rectangles` הרלוונטיים לסצינה.
 - ii. אם מצב הלוח של המשחק הממוחשב שונה מהמפסד אז:
 1. צייר את `Tiles` של לוח המשחק הממוחשב.
 - iii. אחרת:
 1. צייר את `Tiles` של לוח המשחק האנושי.
 - iv. אם מצב לוח המשחק הוא המפסד אז:
 1. אם ניקוד המשחק האנושי > ניקוד המשחק הממוחשב אז:
 - a. צייר את `rectangle` וה `texture` הרלוונטיים למצב ניצחון שחזור על שחזור מממוחשב.
 2. אחרת:
 - a. צייר את ה `texture` וה `rectangle` הרלוונטיים למצב הפסד של שחזור מול שחזור מממוחשב.
 - e. אם הסצינה כרגע היא `ChooseAi` אז:
 - i. צייר את הרקע לסצינה (שמכל בתוכו את כל מראה הסצינה).
 - f. אם הסצינה כרגע היא `WatchAi` אז:
 - i. צייר את כל ה `textures` וה `rectangles` הרלוונטיים לסצינה.
 - ii. אם מצב הלוח הוא המפסד אז:
 1. צייר את ה `texture` וה `rectangle` הרלוונטיים למצב הפסד של שחזור מממוחשב.

5. בטל סצינה – `UnloadScene`:

- a. עדכן את רשימת `textures` לרשימה חדשה וריקה.

תיאור אלגוריתם: בדיקת תקינות מהלך + התמצגות ארכיון | (ח)ו

- B מתיחס לחלק התמצגות הארכיון, בעוד שני החלקיםividually ביחס לבדיקה תקינות המהלך (אם בסיסים הפונקציה ערך המשתנה `_boardChanged` הוא אמת אז המהלך הוא תקין, אחרת המהלך לא תקין).
- 1. עברו ז כאשר $4 \dots =0$ בצע:
 - 1.1. בצע כל עוד (do-while) ערך המשתנה `change` הוא אמת:
 - a. עברו j כאשר $j=1 \dots 4$ בצע:
 - b. אם ערך התא במיקום $1-j$, מהטבלה שונה מ-1-, וגם ערך התא במיקום ה- j, מהטבלה שונה מ-1-, וגם הארייך באינדקס שהוא הערך של התא במיקום ה-1-j, מהטבלה ברשימה הארכיון שווה ערך לארייך באינדקס שהוא הערך של התא במיקום ה- j, מהטבלה { $[j][1-j]$ }.
במילים ה- j, מהטבלה { $[j][1-j]$ } Equals(`TilesList`[table][i][j].Equals(`TilesList`[table][i][1-j], j), אז:
 - א. הוסיף את הארייך לרשימה הארכיון שהתמצג.
 - ב. הציב עבור הארייך הראשון (j, i) ברשימה הארכיון את המצב "שדרוג".

- III. הצב עבור הארייח השני (j-i, j) ברשימה הארייחים את המיצב "הרווס".
- IV. הצב עבור התא באינדקס 1-j, j בטבלה את הערך 1.
- V. הצב למשתנה change את הערך true.
- VI. עוצר לולאה.
- C. אם ערך התא במיקום ה-1-j, מהטבלה שווה ל-1. וערך התא במיקום j, שונה מ-1 אז:
 - I. עדכן את הערך בטבלה באינדקס 1-j, j, לערך שנמצא באינדקס j, j.
 - II. עדכן את הערך בטבלה באינדקס j, j, ל-1.
 - III. הצב למשתנה change את הערך true.
 - D. הצב למשתנה boardChanged את ערכו או את הערך של change.
- 2. החזר את זוג הערכים {טבלת הערכים המעודכנת, מילון אינדקסים הערכים שהתמצג}.

תיאור אלגוריתם: אלגוריתם לשחקן ממוחשב | (n)0

- 1. אם המקלדת לא נועלה וגם מצב הציר במשחק הוא לא הפסד אז:
 - a. אתחול את טבלת לוח המשחק באמצעות קרייה לפונקציה ()AIListToMatrix.
 - b. אתחול את המשתנה bestScore עם המינימלי ביותר שניתן לאחסן בו, ואת המשתנה moveRating עם 0.
 - c. אתחול את מילון המיצבים הטובים ביותר כמילון ריק.
 - d. אתחול את מילון המיצבים האפשריים באמצעות קרייה לפונקציה getAllMoveStates() של AI.
 - e. בעבר כל זוג מפתח-ערך move moveRating מהמילון moves:
 - i. הצב במשתנה moveRating את הערך שמתקיים מקרייה לאלגוריתם ExpectiMax מהמחלקה של AI.
 - ii. אם moveRating גדול מ- bestScore אז:
 - .moveRating = bestScore.
 - .moveRating += moveRating * 1.
 - .moveRating -= moveRating * 2.
 - iii. אם moveRating שווה ל- bestScore אז:
 - .movesBest = move.
 - .movesBest += move.
 - f. אם כמות הערכים בmovesBest גדולה מ- 0 אז:
 - i. קרא לפונקציה לביצוע פעולה עם המפתח הראשון בmovesBest.
 - g. נעל את המקלדת.

מבנה נתונים:**רשימה (List):**

מבנה הנתונים העיקרי בפרויקט הוא רשימה. מבנה זה נלקח מהספרייה System.Collections.Generic של NET. והוא מאפשר לשמר ערכים באופן DINAMI ומסודר. אחד השימושים המרכזיים שהוא לighbה נתונים זה הוא יציג לוח המשחק. את לוח המשחק יציגו באמצעות רשימת ארכחים (`List<Tile>`) אשר אפשרותו לשמור רק את הארכחים שמתבצע בהם שימוש בלוח באותו הזמן, כאשר כל ארכח ברשימה הכל תכונות רבות, וביניהן גם 2 נקודות (Points) שנלקחו מהספרייה Microsoft.Xna.framework. ייצגו את המיקום של כל ארכח בלוח לפני ואחרי מהלך, כאשר מתייחסים ללוח כאלו מטריצה דו מימדית בגודל 4x4. שימוש מרכזי נוסף שביצעת עם מבנה נתונים זה היה בחילוק של הבינה המלאכותית, כאשר היעתית ציר לחשב את השפעת מהלך כלשהו על המשחק. באמצעות רשימה של מצבים, יציגו את הלוחות שעשויהם להיזכר בעקבות מהלך מסוים, ולאחר מכן נעזרתי בכך על מנת לקבוע איזה מהלך הוא הטוב ביותר לביצוע ע"פ האלגוריתם ExpectiMax.

מטריצה (Matrix):

בפרויקט, ובמיוחד בחילוק הבינה המלאכותית בפרויקט, נעזרתי במטריצה, שהיא בעצם מערך דו מימדי ריבועי, כלומר כמות השורות והטורים בו זהה, כמוון שהיה לנו יותר לייצג ולעבוד עם לוחוק המשחק באופן זה, כאשר מתייחסים לכל ארכח (גם אם ערכו אינם מואתחל) בחישובים לקביעת המהלך הטוב ביותר עבור מצב נתון.

מילון (Dictionary):

בפרויקט, ובמיוחד בחילוק הבינה המלאכותית בפרויקט, נעזרתי במילון לצורך שיזור בין המהלך שמבצעים לבין הלוח שמתתקבל בעקבות ביצוע המהלך. מילון הוא מבנה נתונים מסווגר בספרייה System.Collections.Generic של NET. ומטרתו היא לאפשר שמיירה של מפתחות לצד ערכים. אחד השימושים המרכזיים במבנה נתונים זה בפרויקט בא לידי ביטוי בחילוק הבינה המלאכותית בו האלגוריתם צריך לדרג כל מהלך, ולצורך ביצוע החישובים המתאים שיבילו לתוצאות מתאימות, השתמשתי במילון על מנת לחבר בין המפתחות – המהלך שמתבצע (ימינה, שמאלה, מעלה, מטה), לבין הערכים – הלוחות שמתתקבלים בעקבות ביצוע המהלך.

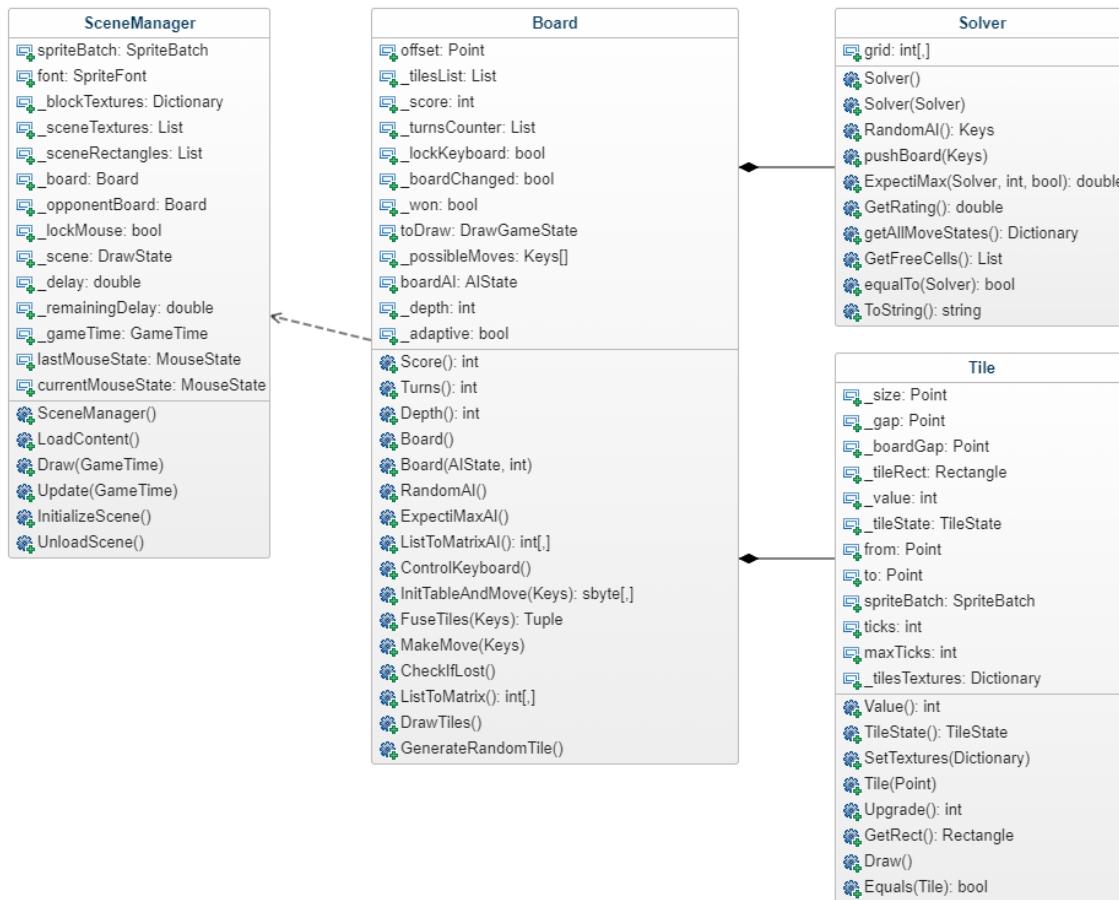
מערך (Array):

בפרויקט, השתמשתי במערך לצורך אחסון וייצוג המהלך המותרים המשחק (ימינה, שמאלה, מעלה, מטה).

ט ממויין (Sorted Set):

ט ממויין הינו מבנה נתונים המאפשר לנו לאחסן ערכים בעודם **ממויינים**. כאמור, אם ניקח סט ממויין יוכל את הערכים בתחום 0-15, האיבר הראשון בסט יהיה 0, השני יהיה 1, וכך הלאה עד האיבר האחרון שערכו יהיה 15. התכוונה החשובה ביותר של מבנה נתונים זה היא שלא משנה איפה פועלה נבצע, הוא תמיד ישאר ממויין. כאמור, אנו יכולים להוסיף או להסיר מספר מהסט מבלי לחשב על ההשפעות של הפעולה שבירצענו על סידור הערכים, כמוון שהם תמיד ישארו ממויינים. אני מבצע שימוש במבנה נתונים זה בפרויקט על מנת לאחסן את אינדקסי האריכחים הפנויים בלוח **בייעילות גבוהה** בזכות תוכנה זו.

תרשים מחלקות – UML Class Diagram



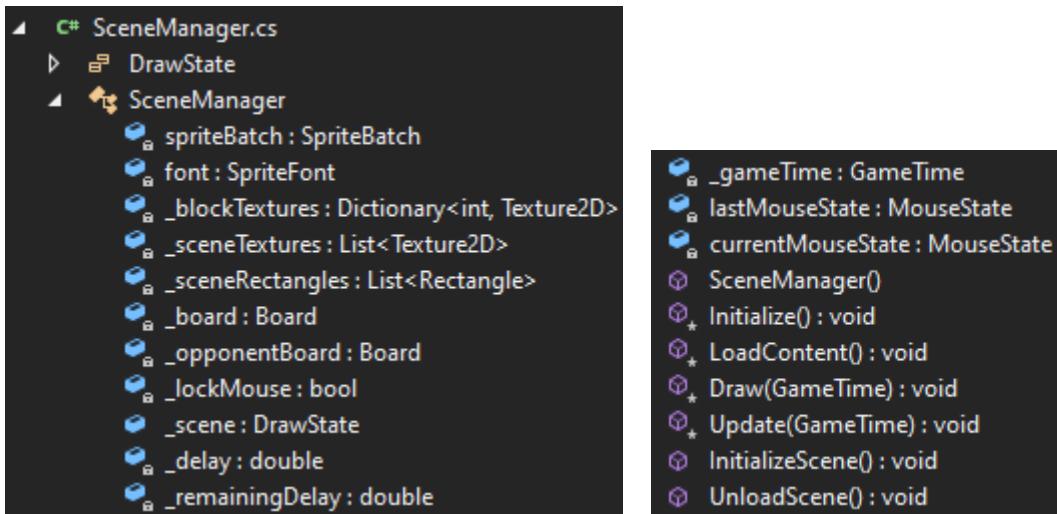
הסבר כללי על המחלקות:

- המחלקה **SceneManager** היא המחלקה האחראית על ה-UI של המשחק והוא מבצעת זאת באמצעות שימוש בפונקציות של *MonoGame*.
- המחלקה **Board** היא המחלקה האחראית על לוח המשחק ועל כל הלוגיקה שפועלת מאחוריו.
- המחלקה **Tile** מייצגת אריח בלוח המשחק, ומכליה מידע אשר עוזר לנו למקם את האריכים על גבי לוח המשחק ולבוד אם כל אריח בaczora יעליה יותר.
- המחלקה **Solver** אחראית על פתרית לוח המשחק ומכליה 2 אלגוריתמים לפתרית לוח בתוכה.

תיכוֹן:

מחלקה :SceneManager

מחלקה זו מנהלת את הסצנות שצריך לטעון. בנוסף, היא יורשת ממחלקה Game שמוגדרת בספריה Microsoft.Xna.Framework



טבלת נתוניים:

פירוט הנתון	שם הנתון
אחראי על ציור המשחק.	<code>private SpriteBatch spriteBatch</code>
אחראי לאחסון ה <code>font</code> בשימוש בעת ציור המשחק.	<code>private SpriteFont font</code>
מילון שיכיל ערכים לצד משתנים מסווג <code>Texture2D</code> שיכילו תמונות שייצגו את הארכח שערכו הוא המפתח.	<code>private Dictionary<int, Texture2D> _blockTextures</code>
רישימה שמכילה <code>Texture2D</code> לצורך אחסון התמונות שנציג במשחק בכל זמן נתון.	<code>private List<Texture2D> _sceneTextures</code>
רישימה שמכילה <code>Rectangle</code> לצורך נתינת "גוף פיזי" שבאמצעותו נוכל לzechות לחיצות על כפתורים ועוד.	<code>private List<Rectangle> _sceneRectangles</code>
מופע לוח משחק אשר בעזרתו נמש לחםוק עברו שחזור אונשי/ ממוחשב.	<code>private Board _board</code>
מופע לוח משחק אשר בעזרתו נמש לחםוק עברו שחזור ממוחשב.	<code>private Board _opponentBoard</code>
משתנה שמאפשר לנו להגביל את מספר הלחיצות על העכבר שהמשתמש יכול לבצע בביטחון.	<code>private bool _lockMouse</code>
מופע של הסט <code>DrawState</code> אשר מאפשר לzechות איזו סצינה צריך לצייר בכל זמן.	<code>public static DrawState _scene</code>

קבוע שטחןתו היא לקבוע דילאי בין מהלכים שהמשתמש מבצע.	<code>private const double _delay</code>
משתנה עזר למקב אחר כמהות הזמן שנותרה עד שהמשתמש יכול לבצע את המהלך הבא שלו.	<code>private double _remainingDelay</code>
משתנה עזר למקב אחר כמהות הזמן בין הפעולות שהמחשב.	<code>private GameTime _gameTime</code>
משתנה עזר למניעת לחיצה מרובה של המשתמש על כפתורים (באותו הזמן).	<code>private MouseState lastMouseState</code>
משתנה עזר למניעת לחיצה מרובה של המשתמש על כפתורים (באותו הזמן).	<code>private MouseState currentMouseState</code>

טבלת פעולה:

תיעוד הפעולה	שם הפעולה
בנייה המחלקה, יוצר עצם מטיפוס SceneManager שינהל את הסצינות שמוצגות למשתמש בכל עת.	<code>public SceneManager()</code>
אפשר למשחק לבצע אתחול כלשהו אשר נדרש לפני ריצת התוכנית. זהה פונקציה מובנית כחלק מהframework שנקרו MonoGame.	<code>protected override void Initialize()</code>
טוען את התוכן הרלוונטי מהirectory של התוכנית (אשר עברו מאותחל במבנה) בהתאם לשכינה הנוכחית. זהה פונקציה מובנית כחלק מהframework שנקרו MonoGame.	<code>protected override void LoadContent()</code>
נקראת כאשר המשחק צריך לצייר את עצמו, ותפקידה הוא לצייר את הסצינות השונות במהלך המשחק בהתאם למצב. זהה פונקציה מובנית כחלק מהframework שנקרו MonoGame.	<code>protected override void Draw(GameTime gameTime)</code>
פונקציה זו נקראת בכל פעם של המשחק עשו render ומוצג על המסך. בפונקציה זו קיימת כל הלוגיקה בסכינה כגון – זהה לחיצות משתמש על כפתורים, זהה מצבים ניצחון/הפסד של שחזור/AI, ועוד. זהה פונקציה מובנית כחלק מהframework שנקרו MonoGame.	<code>protected override void Update(GameTime gameTime)</code>
פונקציה שטחנתה היא לאתחול את הסצינות השונות בהתאם להתקבש על הסכינה הנוכחית.	<code>public void InitializeScene()</code>
פונקציה שטחנתה היא לאפס את הסכינה הנוכחית על מנת שנitin יהיה לטען סכינה חדשה.	<code>public void UnloadScene()</code>

מחלקה :Board

מחלקה זו מייצגת את לוח המשחק.

The image shows the structure of the `Board.cs` class. It includes properties like `Score`, `Turns`, `Depth`, and `offset`. It also contains methods for AI logic such as `RandomAI()`, `ExpectiMaxAI()`, and `FuseTiles(Keys)`.

```

C# Board.cs
  ▶ DrawGameState
  ▶ AIState
  ▶ Board
    ◀ offset : Point
    ◀ _tilesList : List<Tile>
    ◀ _score : int
    ◀ _turnsCounter : int
    ◀ _lockKeyboard : bool
    ◀ _boardChanged : bool
    ◀ _won : bool
    ◀ toDraw : DrawGameState
    ◀ _possibleMoves : Keys[]
    ◀ boardAI : AIState
    ◀ _depth : int
    ◀ _adaptive : bool

  ↗ Score : int
  ↗ Turns : int
  ↗ Depth : int
  ↗ Board()
  ↗ Board(AIState, int)
  ↗ RandomAI() : void
  ↗ ExpectiMaxAI() : void
  ↗ ListToMatrixAI() : int[,]
  ↗ ControlKeyboard() : void
  ↗ InitTableAndMove(Keys) : sbyte[,]
  ↗ FuseTiles(Keys) : Tuple<sbyte[,], Dictionary<int, int>>
  ↗ MakeMove(Keys) : void
  ↗ CheckIfLost() : void
  ↗ ListToMatrix() : int[,]
  ↗ DrawTiles() : void
  ↗ GenerateRandomTile() : void

```

טבלת נתוניים:

פירוט הנתון	שם הנתון
האינדקסים שמשם לוח המשחק מתחילה (על מנת שנוכל לצייר את הארכיהם).	<code>public static Point offset</code>
רשימת הארכים שנמצאים על הלוח בכל זמן נתון.	<code>private List<Tile> _tilesList</code>
מחזיק את תוצאת המשחק בכל זמן נתון.	<code>private int _score</code>
מחזיק את כמות המהלךים שבוצעו במשחק בכל זמן נתון.	<code>private int _turnsCounter</code>
מאפשר לנו לנהל את השימוש במקלדת בזמן המשחק.	<code>private bool _lockKeyboard</code>
מאפשר לנו לזרות האם המשחק השתנה בעקבות ביצוע פעולה או לא.	<code>private bool _boardChanged</code>
מאפשר לנו לזרות האם המשחק ניצח את המשחק או לא.	<code>private bool _won</code>
מאפשר לנו לזרות את מצב הלוח שציר לצייר בכל זמן נתון.	<code>public DrawGameState toDraw</code>
מערך שמכיל את הפעולות שניתן לבצע במהלך (מעלה, מטה, ימינה ושמאליה).	<code>Keys[] _possibleMoves</code>
מאפשר לנו לקבוע איזה סוג של AI צריך לפעול על לוח המשחק.	<code>public AIState boardAI</code>
מאפשר לנו לקבוע את עומק AI שהוא בשימוש במהלך המשחק.	<code>private int _depth</code>
משתנה עוזר לקביעת האם AI צריך לפעול בצורה אדפטיבית או לא.	<code>private bool _adaptive</code>

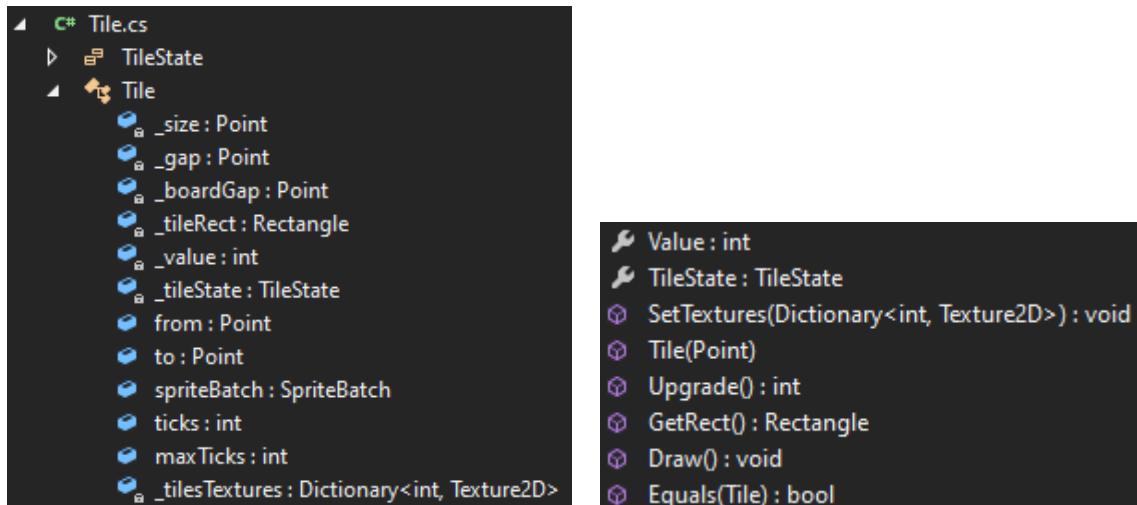
טבלת פעולות:

תיעוד הפעולה	שם הפעולה
בנאי המחלקה, יוצר עצם מטיפוס Board וזמן את הפונקציה להוספת אורך באופן רנדומלי פעמיים .	<code>public Board()</code>
בנאי המחלקה, יוצר עצם מטיפוס Board, מתחילה את תכונותיו הרלוונטיות בהתאם לערבים המתוקבלים וזמן את הפונקציה להוספת אורך באופן רנדומלי פעמיים .	<code>public Board(AIState state, int depth)</code>
אחריות על קישור בין בירה רנדומלית של מהלכים לוח המשחק אשר מוגדרת במחלקת AI ובכך מאפשרת לנו ליצור שחזור ממוחשב שבורח מהלכים באופן רנדומלי.	<code>public void RandomAI()</code>
אחריות על קישור בין בינה מלאכותית שפועל על פי האלגוריתם ExpectiMax שמנדר במחלקת AI לבן לוח המשחק ובכך מאפשרת לאלגוריתם למשתמש בחזון ממוחשב.	<code>public void ExpectiMaxAI()</code>
מירה בין רשימת הארכחים למטריצה שמאתחלת בערכי הארכחים ע"פ הרשימה כאשר הערך של כל אורך מהרשימה יוכנס כ $2^{\text{log}_2 \text{שלו}}$ (לדוגמא, עבור 8 יוכנס 3). המטריצה היא בגודל 4×4 כאשר התאים שלה יאותחלו בהתאם לערכי הארכחים (על פי תכונות הארכחים שמעידים על האינדקס שלהם במטריצה).	<code>private int[,] ListToMatrixAI()</code>
אפשרות לוח המשחק להיות מושפע מלחיצות מקשי המקלדת. פונקציה זו תזהה לחיצות על מקשי מקלדת ותפעל בהתאם אליהם.	<code>public void ControlKeyboard()</code>
מתחול מטריצה של בתים מסוימים אשר תעוזר לנו לזהות אלו ארכחים הושפעו כתוצאה מביצוע תזוזה.	<code>private sbyte[,] InitTableAndMove(Keys key)</code>
אחריות על זיהוי האם הלוח מושפע/לא מושפע כתוצאה ממחלך של משתמש ועל בדיקה אילו ארכחים התמזגו כתוצאה מהתזוזה במידה יש כן.	<code>private Tuple<sbyte[,], Dictionary<int, int>> FuseTiles(sbyte[,] table)</code>
הפעולה המרכזית שאחריאת על תזוזות המשמש. פועלה זו נעזרת בשתי הפעולות InitTable, FuseTiles ומבצעת את תזוזת הארכחים.	<code>public void MakeMove(Keys key)</code>
בודקת האם בזמן שהיא נקראת למשתמש לנתרו מהלכים לבצע, מה שמעיד על כך שהוא הפסיד, או לא. במידה והמשתמש הפסיד, הפונקציה תשנה את ערך המשתנה	<code>private void CheckIfLost()</code>

שם מיצג מה צריך לצייר כרגע בלוח במצב ה ^{90ד.}	
מミירה בין רשימת הארכחים למטריצה שמאתחלת בערכי הארכחים ע"פ הרשימה. המטריצה היא בגודל 4X4 כאשר התאים שלה יאותחלו בהתאם לערכי הארכחים (על פי תכונות הארכחים שמעודדים על האינדקס שלהם במטריצה).	<code>private int[,] ListToMatrix()</code>
אחריות על ציור הארכחים בלוח המשחק בכל רגע. פונקציה זו תזהה אילו ארכחים צריך "להשמיד", אילו ארכחים צריך "לשדרג" ואילו ארכחים צריך להזיז או להשאיר כפי שהם, בהתאם למצבם הנוכחי.	<code>public void DrawTiles()</code>
אחריות על צימון ארכח חדש ללוח המשחק. פונקציה זו בוחרת באופן רנדומלי את ערך הארכח כאשר יש 90% שערכו יהיה 2 ו-10% שערכו יהיה 4, אז בוחרת באופן רנדומלי מיקום על לוח המשחק שאילו יתווסף הארכח החדש.	<code>public void GenerateRandomTile()</code>

:tile מחלקת

מחלקה זו מייצגת ארכח בלוח המשחק.

**:טבלת נתונים:**

פירוט הנתונים	שם הנתון
גודל הארכח שמוסג למשתמש בזמן המשחק.	<code>private Point _size</code>
המרחק בין ארכחים צמודים בלוח המשחק.	<code>private Point _gap</code>
גודל השולטים של לוח המשחק.	<code>private Point _boardGap</code>

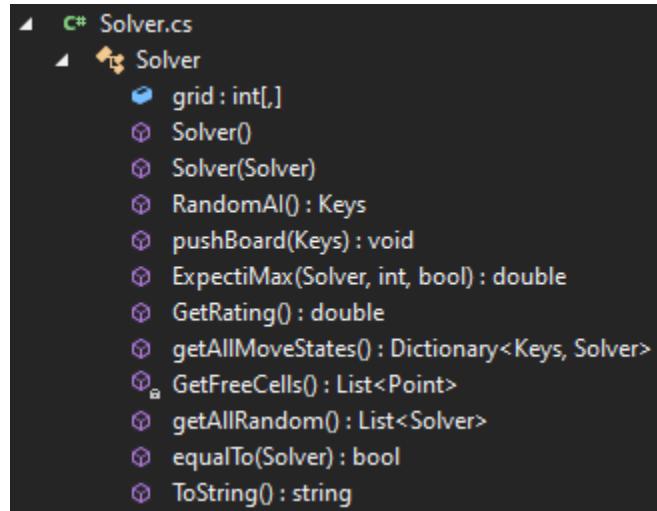
משתנה לייצוג "הגוף הפיזי" של הארייח. באמצעות משתנה זה נוכל לייצג את הארייחים השונים של הלוח במקומות שונים.	<code>private Rectangle _tileRect</code>
משתנה לאחסן ערך הארייח.	<code>private int _value</code>
משתנה לייצוג מצב הארייח ("הריסה"/"שדרוג"/"רגע").	<code>private TileState _tileState</code>
משתנה לייצוג אינדקס הארייח בלוח לפני ביצוע המהלך הנוכחי.	<code>public Point from</code>
משתנה לייצוג אינדקס הארייח בלוח אחרי ביצוע המהלך הנוכחי.	<code>public Point to</code>
משתנה שמטרתו היא לקשר בין <code>SceneManager</code> ו- <code>SpriteBatch</code> לבין מחלקת הארייח, כך שנוכל לצויר את הארייחים השונים דרך מחלקותם.	<code>public static SpriteBatch spriteBatch</code>
אפשר לנו לעקוב אחר כמות ההזדוזות שביצענו עבור כל ארייח בעקבות מהלך מסוים.	<code>public static int ticks</code>
אפשר לנו לקבוע כמה ההזדוזות אנחנו מעוניינים לבצע לאירועים עבור כל מהלך שמתרחש.	<code>public static int maxTicks</code>
אפשר לנו לשמר ולאחר מכן להציג את התמונות שמייצגות כל ארייח בהתאם לערכו.	<code>private static Dictionary<int, Texture2D> _tilesTextures</code>

טבלת פעולות:

תיעוד הפעולה	שם הפעולה
משווה את מילון <code>_tilesTextures</code> למילון המתקבל.	<code>public static void SetTextures(Dictionary<int, Texture2D> dict)</code>
בנייה המחלקה, יוצר עצם מטיפוס <code>Tile</code> ושם עבورو את ערכיו <code>X</code> וה <code>Y</code> של הנקודה שהתקבללה כערך <code>from</code> והסן שלו.	<code>public Tile(Point point)</code>
פעולה זו מעדכנת את ערך הארייח הנוכחי לפי 2 מערכו, מעדכנת את מצבו ל"רגע" ומחזירה את הערך החדש של הארייח.	<code>public int Upgrade()</code>
פעולה זו מחרירה את ה"ריבוע" שעל פ' אינדקסי <code>X</code> וה <code>Y</code> שלו לאחר מכן נצייר את הארייח הנוכחי על המסך.	<code>public Rectangle GetRect()</code>
פעולה זו אחראית לצויר כל ארייח ע"פ ה"ריבוע" שמתאים לו, אשר מופק ע"י <code>GetRect()</code> .	<code>public void Draw()</code>
מחזירה אמת אם הארייח הנוכחי שווה לארייח אחר. ארייח נחשב לידה לארייח אחר אם ערכו ומצבו זהים לאלו של הארייח ומצבו "רגע".	<code>public bool Equals(Tile other)</code>

מחלקה :Solver

מחלקה זו מכילה את הלוגיקה מאחורי האלגוריתמים שפותרים את הלוח (באמצעות בינה מלאכותית / באמצעות ביצוע מהלכים באופן רנדומלי).

**טבלת נתונים:**

שם הנתון	פירוט הנתון
<code>public int[,] grid</code>	מטריצה שמייצגת את לוח המשחק אשר מקשר למופע הנוכחי של המחלקה.

טבלת פעולות:

תיעוד הפעולה	שם הפעולה
בנאי המחלקה, יוצר עצם מטיפוס Solver ומאתחל את הgrid שלו להיות מטריצה בגודל 4x4.	<code>public Solver()</code>
בנאי המחלקה, יוצר עצם מטיפוס Solver ומאתחל את הgrid שלו להיות מטריצה בגודל 4X4 שערכיה זהים לערכי grid של s, Solver s שהבנאי מקבל.	<code>public Solver(Solver s)</code>
הפעולה בוחרת באופן רנדומלי מהלך מבין 4 המהלך האפשריים במשחק ומחזיר אותו.	<code>public static Keys RandomAI()</code>
הפעולה משפיעה על הgrid של Solver ע"י כך שהיא מבצעת את המהלך שהוא מקבלת. לדוגמה, אם היא מקבלת מקש שמאל אז היא מבצעת תזוזה שמאלית.	<code>public void pushBoard(Keys key)</code>
פעולות הבינה המלאכותית המרכזית של הפרויקט. פעולה זו פועלת ע"פ אופן הפעולה של האלגוריתם ExpectiMax, קלומר בוחרת בכל צומת את ממוצע הבנים של הצומת	<code>public static double ExpectiMax(Solver root, int depth, bool player)</code>

ובשורש את הבן שערכו הוא הגבוה ביותר. עבור כל צומת "בעץ" שנוצר, הפעולה מחשבת את הלוח שיוציאר, ובסיום מחזירה את דירוג המהalker האופטימלי לביצוע.	
פעולה זו מחשבת את הניקוד שמהלך מסוים צריך לקבל. פעולה זו מחזירה את הניקוד הגבוה מבין הניקודים ששתי הרשימות ScanGrids קיבלו.	<code>public double GetRating()</code>
מחזירה מיליון שוכל את התזוזה (מפתח) לצד מצב הלוח (ערך) בעקבות ביצוע כל אחת מהתזוזות האפשריות, כאשר במיון יהיו רק הפעולות שביצון ישפיע על לוח המשחק.	<code>public Dictionary<Keys, Solver></code> <code>getAllMoveStates()</code>
מחזירה רשימה של נקודות שמכילה את האינדקסים של כל התאים בgrid שערכם איננו מאותחל.	<code>private List<Point></code> <code>GetFreeCells()</code>
מחזירה רשימה של מצבים שמייצגים את כל اللוחות שעשוים להיווצר בעקבות התווספות אריך חדש ללוח המשחק.	<code>public List<Solver></code> <code>getAllRandom()</code>
מחזירה אמת אם grid של מופיע המחלקה שעריו הפונקציה נקרא זהה לgrid של מופיע המחלקה שהfonktsiya מקבלת.	<code>public bool equalTo(Solver alt)</code>
אפשררת לנו לראות את לוח המשחק בצורה פשוטה וברורה יותר בזמן debugging.	<code>public override string</code> <code>ToString()</code>

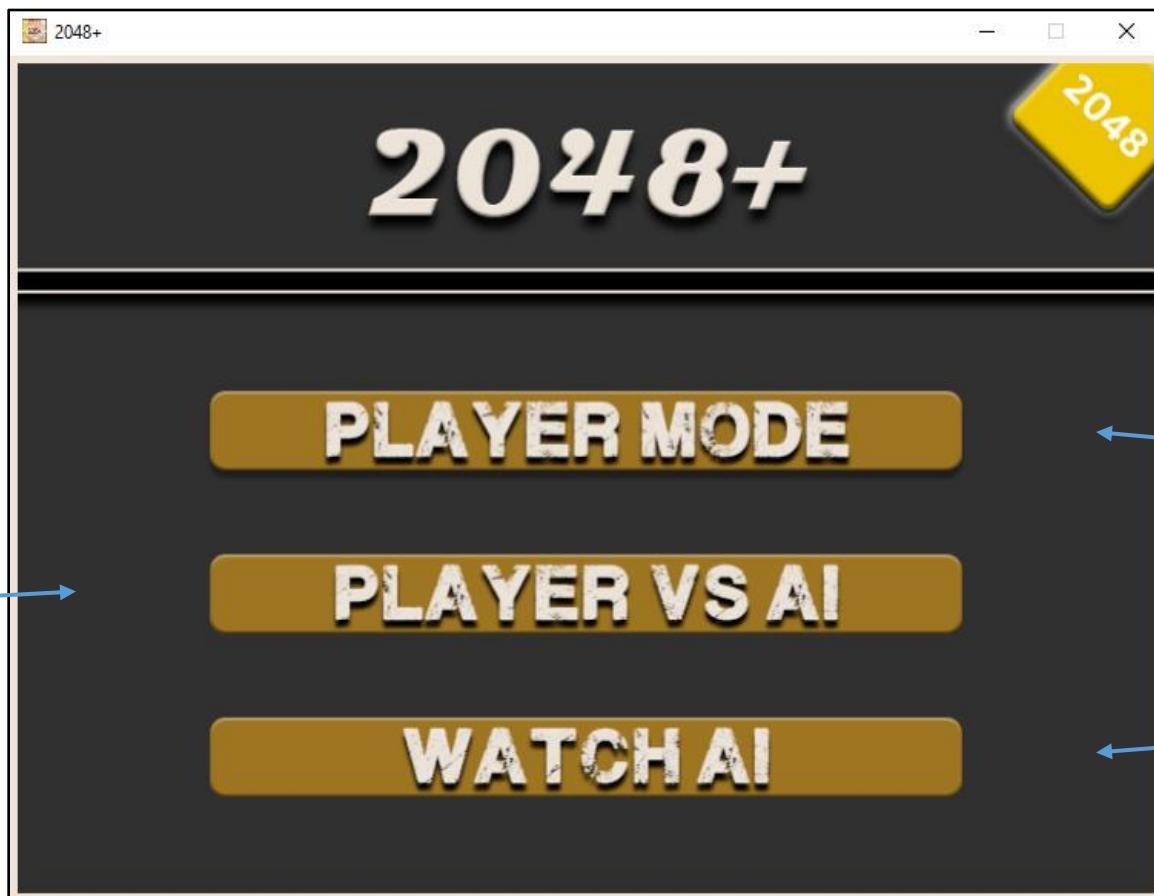
מדריך למשתמש

cutet אספיר בקצרה כיצד לתפעל את המערכת כראוי.

הוראות הפעלה:

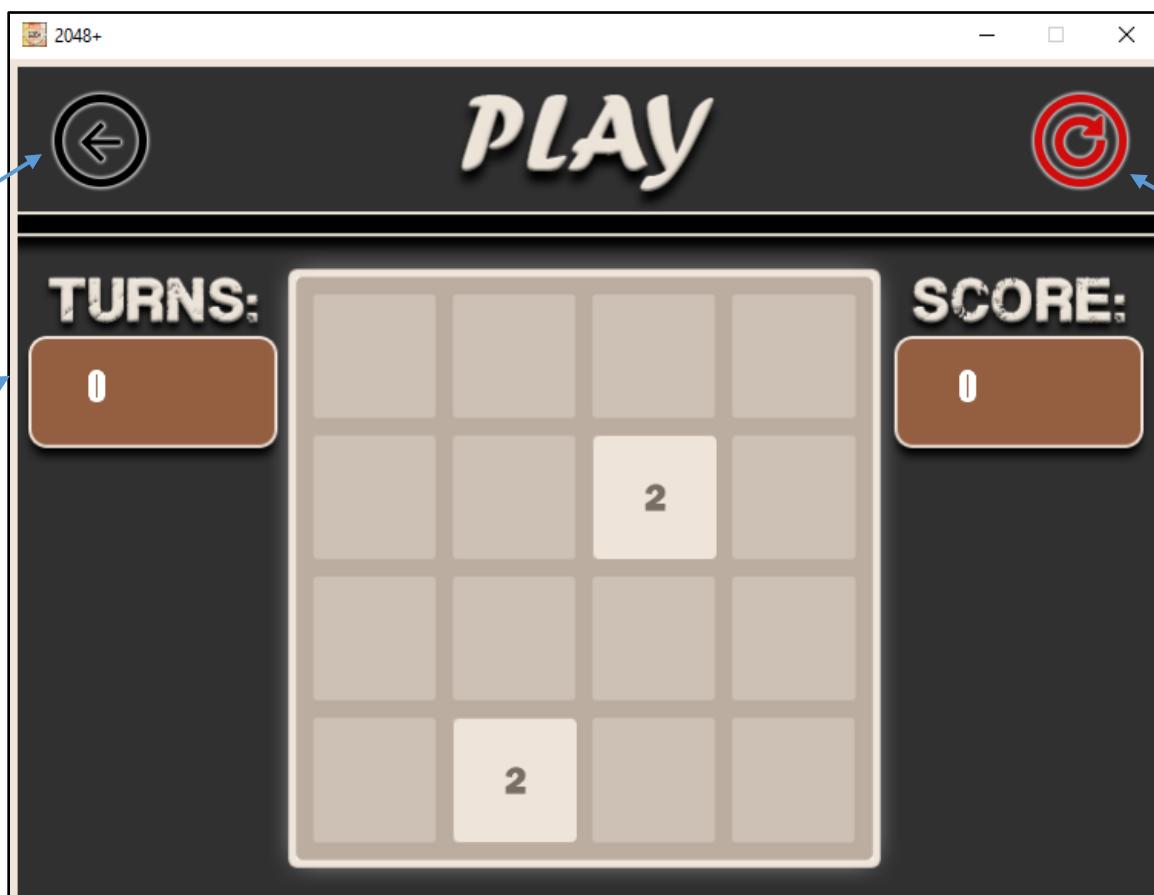
במוך זה המשתמש צריך לבחור את האופן שבו הוא מעוניין לשחק את המשחק. את המשחק ניתן לשחק בשלושת האופנים הבאים:

1. משחק שחקן אנושי (ගගՏתו הרגילה של 2048).
2. משחק אנושי מול שחקן ממוחשב (מתחרים להגיע לנצח הגדולה ביותר).
3. משחק שחקן ממוחשב (השחקן הממוחשב מנסה להגיע לנצח הגדולה ביותר).



מסך משחק שחקן אנושי:

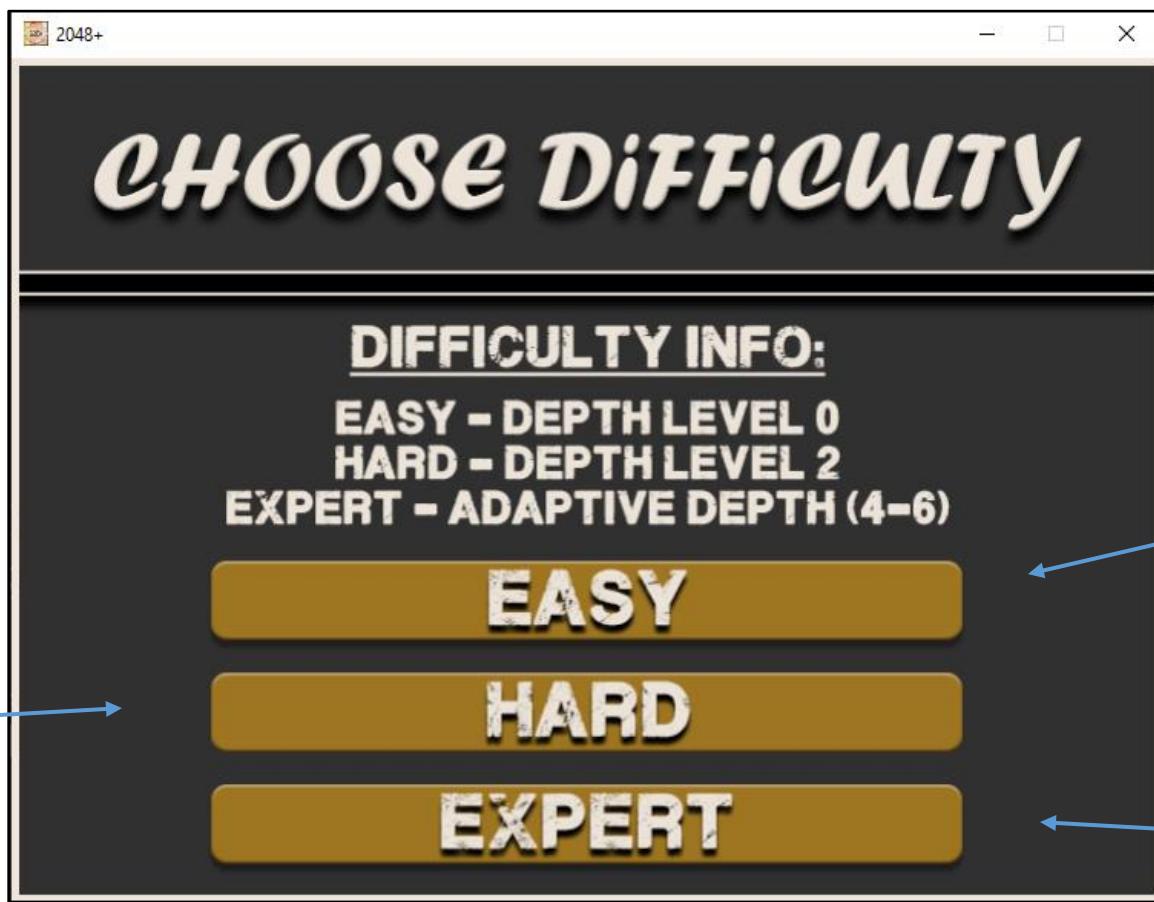
כאשר השחקן נמצא במסך זה, הוא יכול למשחק בגרסתו הרגילה, כאשר מטרתו של השחקן היא להגיע לאנרגיה בעל הערך 2048 על מנת לנצח.



מיסוך בבחירה דרגת קושי עבור שחקן ממוחשב:

השחקן מגיע למסך זה לאחר לחיצה על הכפתור למעבר למשחק שחקן אנושי מול שחקן ממוחשב דרך העמוד הראשי של המשחק. מטרת עמוד זה היא לאפשר לשחקן לקבוע מהו דרגת הקושי של השחקן הממוחשב שמולו הוא מעוניין לשחק.

האלגוריתם מולו השחקן האנושי יתמודד הוא ExpectiMax, ובהתאם לבחירת השחקן האלגוריתם יחשב יותר "לעומק" כל מהלך, מה שיוביל לכך שבסתומו של דבר הסיכויים שלו להיות טוב יותר יהיו גבוהים יותר.



מסר משחק שחון אנושי מול שחון ממוחשב:

כasher hashukan nmea b'meser zeh, hoa ycol leshukn b'mashuk 2048 bgertsot ha'mishuprot, casher matrato hia legayu lenikud gibva maza shashukan ha'mmochab ygeu alio. Rashiit yishukn hashukan ha'mmochab v'lachar shiyepsel, yishukn hashukan ha'anooshi.

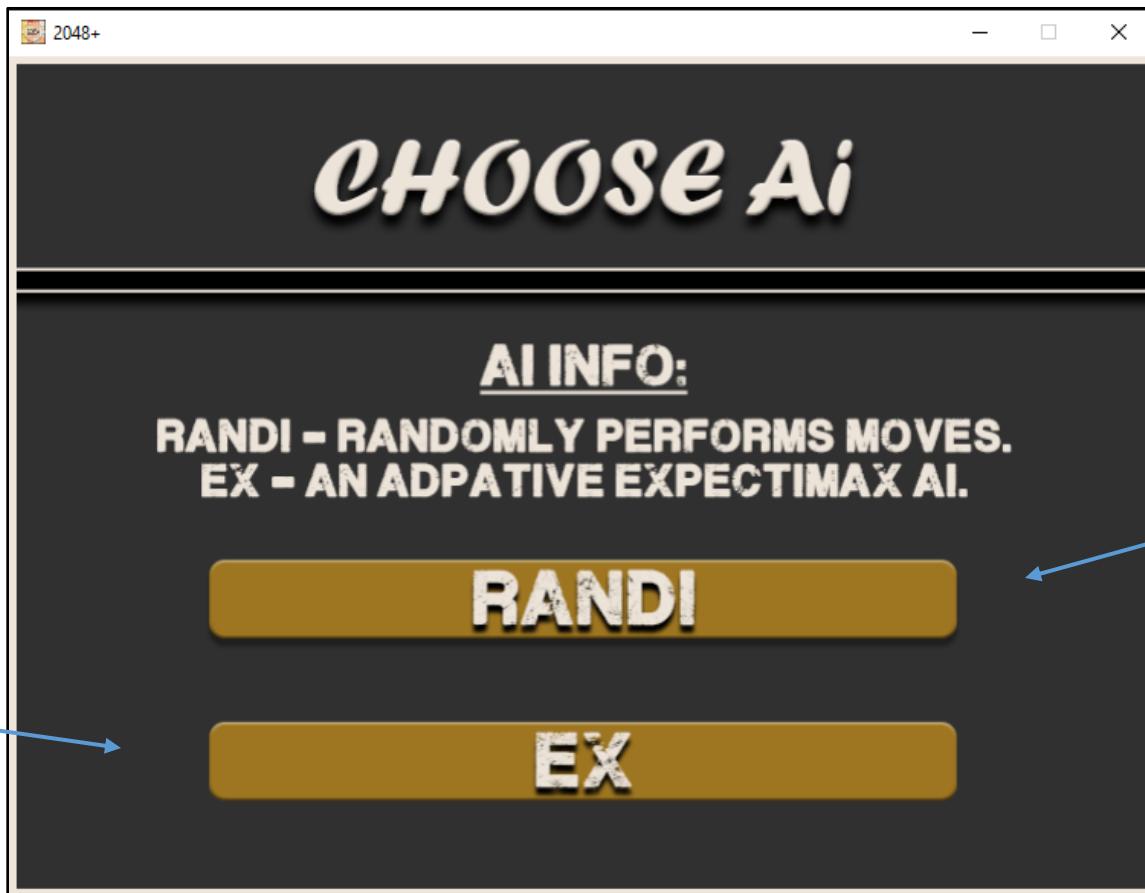
המנצח יקבע בסיום המשחק, כאשר שני השחקנים נפסלו, זהה שהגיעו לпозיציה הגבוהה יותר מבין
השניים – ניצח.



מיסוך בבחירה AI:

השחקן מגיע למסך זה לאחר לחיצה על הכפתור למעבר למשחק שחקן ממוחשב דרך העמוד הראשי של המשחק. מטרת עמוד זה היא לאפשר למשתמש לבחור את השחקן הממוחשב שהוא רוצה לראות משחקים.

במידה ומשתמש יבחר ב-**Randi**, השחקן הממוחשב יבחר ויבצע מהלכים באופן רנדומלי. במידה ומשתמש יבחר ב-**Ex**, השחקן הממוחשב יבחר ויבצע מהלכים ע"פ האלגוריתם **ExpectiMax**.



מיצר משחק שחקן ממוחשב:

כאשר השחקן נמצא במסך זה, הוא יכול למשחק במשחק 2048 בגרסתו המשופרת, כאשר מטרתו היא להגיע לניקוד גבוה מזו שהשחקן הממוחשב יגיע אליו. ראשית ישחק השחקן הממוחשב ולאחר מכן, ישחק השחקן האנושי.

המנצח יקבע בסיום המשחק, כאשר שני השחקנים נפסלו, וזה שהגיע לתוכאה הגבוה יותר מבין השניים – ינצח.



ביבליוגרפיה

תודות:

מייכאל צ'רנובולסקי

אלון חיימוביץ

אתרי אינטרנט:

<https://www.youtube.com> .1

<http://en.wikipedia.org> .2

<https://stackoverflow.com> .3

<http://www.monogame.net/> .4

מאמרם / מחקרים / אסטרטגיות:

1. מאמר אקדמי שمدבר על שימוש בבינה מלאכותית על מנת לנצח את המשחק 2048

<https://home.cse.ust.hk/~yqsong/teaching/comp3211/projects/2017Fall/G11.pdf>

2. מאמר אקדמי על שימוש באלגוריתם ExpectiMax על מנת לנצח את המשחק 2048

https://www.cs.uml.edu/ecg/uploads/AIfall14/vignesh_gayas_2048_project.pdf

3. אסטרטגיות למשחק 2048

<https://www.imore.com/2048-tips-and-tricks>

נספחים – קטעי קוד מרכזיים בתוכנית:

1. מחלקה לוח משחק – *:Board.cs*

```
// A public enum which will help us determine at which game state we are at any
given time:
public enum DrawGameState
{
    Play,
    Win,
    Lose
}

// A public enum which will help us determine which AI should play the game at
any given time:
public enum AIState
{
    Randi,
    Ex,
    None
}

public class Board
{
#region Variables Definition:
    public static Point offset = new Point(186, 140); // The indexes from
    which the board begins.
    private List<Tile> _tilesList = new List<Tile>(); // Creating a list of
    tiles.
    private int _score = 0; // A variable to store the game's score at any
    given moment.
    private int _turnsCounter = 0; // A variable to store how many turns have
    been made at any given moment.
    private bool _lockKeyboard = false; // Locking the keyboard between moves
    that are performed by the user.
    private bool _boardChanged = false; // Determining whether the board
    changed by a user's movement.
    private bool _won = false; // Determining when the user wins the game.
    public DrawGameState toDraw = DrawGameState.Play; // Defining a variable
    to help us determine what to draw.
    Keys[] _possibleMoves = { Keys.Up, Keys.Left, Keys.Right, Keys.Down }; // Initializing
    an array of possible moves.
    public AIState boardAI = AIState.None; // Defining a variable to help us
    determine what to draw.
    private int _depth; // A variable to store the depth to which the AI
    should reach.
    private bool _adaptive; // A variable to determine whether the AI should
    be "adaptive" or not.
#endregion
#region Getters & Setters:
    public int Score { get => _score; set => _score = value; } // A getter and
    a setter to the score variable.
    public int Turns { get => _turnsCounter; set => _turnsCounter = value; }
    // A getter and a setter to the turns variable.
}
```

```

        public int Depth { get => _depth; set => _depth = value; } // A getter and
a setter to the depth variable.
#endifregion
#region Constructors:
    /// <summary>
    /// General: A Constructor for the Board class.
    /// </summary>
    public Board()
    {
        GenerateRandomTile();
        GenerateRandomTile();
    }

    /// <summary>
    /// General: A Constructor of the Board object that receives and
initializes the AIState and the depth for the board class.
    /// </summary>
    /// <param name="state">The state of the AI in the board. The function
will initialize the boardAI's value to this instance.</param>
    /// <param name="depth">The depth of the AI. This function will initialize
the _depth value of this class to this instance.</param>
    public Board(AIState state, int depth)
    {
        boardAI = state;
        // If depth == -1 --> The AI should be adaptive.
        if (depth == -1)
        {
            _depth = 4;
            _adaptive = true;
        }
        // Otherwise, setting the depth to the received value:
        else
            _depth = depth;

        GenerateRandomTile();
        GenerateRandomTile();
    }
#endregion
#region AI:
    /// <summary>
    /// General: The main function which handles the random AI in the project.
    /// Process: Calling the RandomAI function from the Solver class and then
performing the received move.
    /// </summary>
    public void RandomAI()
    {
        // Variables Definition:
        if (!_lockKeyboard && toDraw != DrawGameState.Lose)
        {
            MakeMove(Solver.RandomAI());
            _lockKeyboard = true;
        }
    }

    /// <summary>
    /// General: The main AI in the project. This function handles all the
logic that lies between calculating the best move

```

```

        ///          to perform at a given time and between actually performing it
on the board.

        /// Process: Calling the ExpectiMax function of the Solver class to
calculate the score for every possible movement and it's effect
        ///          on the board. Then, iterating through those scores to find
the best score, and performing the movement that received it.
        ///          Also, if the class's bool _adaptive equals to true, then
changing the depth of the AI based on the amount of tiles on the board.

        /// </summary>
public void ExpectiMaxAI()
{
    Solver currentGame = new Solver(); // A variable to store a Solver
instance.

    // Running AI only if the game didn't end yet and the keyboard isn't
locked:
    if (!_lockKeyboard && toDraw != DrawGameState.Lose)
    {
        // Variables Definition:
        currentGame.grid = ListToMatrixAI();
        double bestScore = double.MinValue;
        List<Keys> newMoves = new List<Keys>();
        Dictionary<Keys, Solver> movesBest = new Dictionary<Keys,
Solver>();
        Dictionary<Keys, Solver> moves = currentGame.getAllMoveStates();
        double moveRating = 0;

        // Adaptive AI:
        // If there are less than/equal to 10 tiles on the board, the AI's
depth should be set to 4.
        // If there are 11-14 tiles on the board, the AI's depth should be
set to 5.
        // If there are 14-16 tiles on the board, the AI's depth should be
set to 6.
        if (_adaptive == true && _score >= 30000)
        {
            if (_tilesList.Count >= 8 && _tilesList.Count <= 12)
                _depth = 5;
            else if (_tilesList.Count > 12)
            {
                _depth = 6;
            }
            else
                _depth = 4;
        }

        // Calculating the rating for each possible move in the board at a
given moment, and then adding it
        // to a list that contains the best movement (or movements if
multiple movements share the same score):
        foreach (KeyValuePair<Keys, Solver> move in moves)
        {
            moveRating = Solver.ExpectiMax(move.Value, _depth, false);

            if (moveRating > bestScore)
            {
                bestScore = moveRating;
                newMoves.Clear();
            }
        }
    }
}

```

```

        }

        if (moveRating == bestScore)
        {
            newMoves.Add(move.Key);
        }
    }
    // Eventually, performing a move if we received back a move to
make:
if (newMoves.Count > 0)
    MakeMove(newMoves[0]);

    // Unlocking the keyboard after performing a movement:
    _lockKeyboard = true;
}
}

/// <summary>
/// General: Converting the list of tiles to matrix that contains their
log2 values.
/// Process: For each tile, based on it's X & Y values, placing it in the
new table with a log2 value for it's actual value.
///           For example, if a tile had the value 8 and was placed in the
index (3, 3), it will be added as a 3 (log2(8) = 3)
///           to the cell at the indexes (3, 3) in the matrix.
/// </summary>
/// <returns> Returns the new matrix initialized by the tiles list's log2
values. </returns>
private int[,] ListToMatrixAI()
{
    int[,] table = new int[4, 4];
    foreach (Tile curTile in _tilesList)
    {
        table[curTile.to.X, curTile.to.Y] = (int)Math.Log(curTile.Value,
2);
    }
    return table;
}
#endregion
#region Perform Movement:
/// <summary>
/// General: Letting the board be affected by keyboard keys presses.
/// Process: Using a bool to lock the keyboard between moves that are
performed by the user and calling
///           the MakeMove function with the key that the user pressed in
order to perform that move.
/// </summary>
public void ControlKeyboard()
{
    // Variables Definition:
    KeyboardState keyboard = Keyboard.GetState(); // Letting keyboard
handle user's keyboard input.

    // Checking whether any of the pressed keys is an arrow key
(up/down/left/right arrow):
    if (keyboard.GetPressedKeys().Intersect(_possibleMoves).Any())
    {

```

```

        // Checking whether the keyboard is locked or not (to make sure
that we can "make a move"):
        if (!_lockKeyboard)
        {
            // Calling the make move function and locking the keyboard:
            MakeMove(keyboard.GetPressedKeys()[0]);
            _lockKeyboard = true;
        }
    }
    else
    {
        // Else setting the lock keyboard to false because the entered key
doesn't affect the
        // board as it isn't a part of the _possibleMoves array:
        _lockKeyboard = false;
    }
}

/// <summary>
/// General: Initializing a matrix of SIGNED BYTES with values which will
help us determine which tiles
/// were affected as a result of the user's movement and which
weren't, and moving those that
/// were moved by changing their indexes in the table.
/// Process: Initializing the matrix's values based on the TilesLists
tiles' values, while for each tile
/// "performing" a movement and then setting a value for it in
the matrix. Also, using a counter
/// so that later we will know the order on which each tile has
been placed.
/// </summary>
/// <param name="key"> A key which resembles a movement to perform.
</param>
/// <returns> Returning the initialized matrix of SIGNED BYTES. </returns>
private sbyte[,] InitTableAndMove(Keys key)
{
    // Variables Definition:
    sbyte[,] table = new sbyte[4, 4] { { -1, -1, -1, -1 }, { -1, -1, -1,
-1 },
                                         { -1, -1, -1, -1 }, { -1, -1, -1, -1 } };
    sbyte counter = 0; // A helping variable to go through the tiles in
the tiles list.

    // Initializing each tile's matching slot in the table with the
counter's value, while each time
    // increasing the counter. This will allow us to later access each
tile which should be
    // changed by performing the move that the user selected:
    foreach (Tile curTile in _tilesList)
    {
        switch (key)
        {
            case Keys.Up:
                table[curTile.from.Y, curTile.from.X] = counter++;
                break;
            case Keys.Down:
                table[3 - curTile.from.Y, 3 - curTile.from.X] = counter++;
        }
    }
}

```

```

                break;
            case Keys.Left:
                table[curTile.from.X, curTile.from.Y] = counter++;
                break;
            case Keys.Right:
                table[curTile.from.X, 3 - curTile.from.Y] = counter++;
                break;
        }
    }

    // Returning the SIGNED BYTES matrix we created with the initialized
values:
    return table;
}

/// <summary>
/// General: Checking which tiles get fused as a result of the user's
movement.
/// Process: Using 3 loops to find and fuse tiles that should fuse as a
result of the movement by the user.
/// </summary>
/// <param name="table"></param>
/// <returns> Returning a Tuple that contains a table of SIGNED BYTES and
a dictionary of the indexes of the
///           tiles that got fused during the movement. </returns>
private Tuple<sbyte[,], Dictionary<int, int>> FuseTiles(Keys key)
{
    sbyte[,] table = InitTableAndMove(key); // Defining and initializing a
table for the current board based on the move.
    // Fusing all the tiles that should be fused as a result of the user's
movement:
    bool change = false; // A bool to determine whether the board changed
or not by an operation.
    Dictionary<int, int> fused = new Dictionary<int, int>(); // Dictionary
of fused tiles from the move.
    for (int i = 0; i < 4; i++)
    {
        do
        {
            change = false; // Resetting change's value each run.
            for (int j = 1; j < 4; j++)
            {
                // Checking whether both tiles values are equal, their
state is Normal
                // and they were affected by the user's movement:
                if (table[i, j - 1] != -1 && table[i, j] != -1
                    && _tilesList[table[i, j - 1]].Equals(_tilesList[table[i, j]]))
                {
                    fused.Add(table[i, j - 1], table[i, j]); // Adding the
fused tiles to the fused tiles dictionary.
                    _tilesList[table[i, j]].TileState = TileState.Upgrade;
// Upgrading the first tile.
                    _tilesList[table[i, j - 1]].TileState =
TileState.Delete; // Then deleting the 2nd tile.
                    table[i, j] = -1; // That position in table is no
longer taken as the tile has been moved.
                }
            }
        }
    }
}

```

```

change = true; // A fuse occured therefore setting
change's value to true.
occurred.
}

// Checking whether the 2nd tile was moved as a result of
// If it was moved, we would like to switch between it's
// to the first tile's value in the table in order to move
// matching direction (based on the movement direction)
if (table[i, j - 1] == -1 && table[i, j] != -1)
{
    table[i, j - 1] = table[i, j];
    table[i, j] = -1;
    change = true;
    break;
}
// The _boardChanged bool will be set to true when a fuse
occurs:
_boardChanged = _boardChanged || change;
}
while (change); // Because we can't determine how many
fuses/movements will occur in a row.
}

// Returning both the updated table and the fused dictionary:
return Tuple.Create(table, fused);
}

/// <summary>
/// General: The main function that is in charge of performing a movement.
This function used the FuseTiles function
///           that calls the InitTableAndMove function all in the sake of
performing a movement.
/// Process: Calling the FuseTiles to get the board after performing the
fusing and movement of the tiles using a matrix.
///           Then, converting the received matrix back to the tiles list
by setting the .to attribute of the matching tiles
///           to what it should be updated to based on the chosen movement,
fusing the tiles that should get fused and starting
///           the tiles movement "animation".
/// </summary>
/// <param name="key"> A key representing the movement to perform.
</param>
public void MakeMove(Keys key)
{
    // Variables Definition:
    _boardChanged = false; // Checking whether the movement changed the
board.
    Tuple<sbyte[,], Dictionary<int, int>> temp = FuseTiles(key); // Fusing
all the tiles that should be fused as a result of the user's movement.
    sbyte[,] table = temp.Item1; // Defining a table for the current board
based on the move.
}

```

```

Dictionary<int, int> fused = temp.Item2; // Creating a dictionary to
store the tiles that fused due to the movement.

// Performing the movement of each tile in the table:
for (int i = 0; i < table.GetLength(0); i++)
{
    for (int j = 0; j < table.GetLength(1); j++)
    {
        if (table[i, j] != -1)
        {
            switch (key)
            {
                // Movement Up - Moving to the current tile's indexes
                // in the table:
                case Keys.Up:
                    _tilesList[table[i, j]].to = new Point(j, i);
                    break;
                // Movement Down - Moving to the current tile's
                // indexes in the table with the exception of 3-j instead j:
                case Keys.Down:
                    _tilesList[table[i, j]].to = new Point(3 - j, 3 -
i);
                    break;
                // Movement Left - Moving to the current tile's
                // indexes in the table:
                case Keys.Left:
                    _tilesList[table[i, j]].to = new Point(i, j);
                    break;
                // Movement Right - Moving to the current tile's
                // indexes in the table with the exceptions
                // of 3-i instead i and 3-j instead j:
                case Keys.Right:
                    _tilesList[table[i, j]].to = new Point(i, 3 - j);
                    break;
            }
        }
    }
}

// Fusing the tiles in the fused tiles dictionary:
foreach (KeyValuePair<int, int> pair in fused)
{
    // Setting the 2nd tile's TO attribute to a new point with a
    // matching tile from _tilesList at the
    // Key (the 1st tile's value) TO attribute:
    _tilesList[pair.Value].to = new Point(_tilesList[pair.Key].to.X,
    _tilesList[pair.Key].to.Y);
}

// Letting the animation begin again if the board has changed
if (_boardChanged)
{
    // Base amount of ticks for the tile:
    Tile.ticks = 0;
}
}

#endifregion

```

```
#region Checking If the user lost:
    /// <summary>
    /// General: The main function that is in charge of checking whether the
    using lost or not at a given time.
        /// Process: Checking the adjacent columns and rows for every cell by each
    time checking 3 adjacent cells.
            /// If at the end the value of the bool "alive" remains false,
    then updating the relevant things
                /// in order to end the game.
    /// </summary>
    private void CheckIfLost()
    {
        bool alive = false;

        // Converting the tiles list to an array:
        var table = ListToMatrix();

        // Checking for adjacent columns and rows for every cell:
        for (int x = 0; x < 4 && !alive; x++)
        {
            for (int y = 0; y < 3 && !alive; y++)
            {
                // Each time checking 3 cells. For example, take the following
                board:
                    // 2 2 0 0
                    // 2 0 0 0
                    // 0 0 0 0
                    // 0 0 0 0
                // In the first run, we will check the 3 tiles that their
                value isn't 0.
                    // { x=0, y=0 -> (x, y), (x, y+1), (y+1, x) }
                // In this way we will cover the whole board each time
                // and if we won't find any possible "fusing", then we will
                know that the game has ended.
                    alive = (table[x, y] == table[x, y + 1]) || (table[y, x] ==
                table[y + 1, x]);
            }
        }
        // If same is still false then the user lost therefore drawing the
        DrawGameState.Lose:
            if (!alive)
            {
                // Drawing the Losing screen:
                toDraw = DrawGameState.Lose;
            }
        }

        /// <summary>
        /// General: Converting the list to a matrix of ints on which each cell
    represents a tile on the board and it's value represents it's matching
            /// tile from the tiles list value.
        /// Process: Using a foreach loop to iterate through the tiles on the list
    and storing the value of each tile in the matrix.
        /// </summary>
        /// <returns> Returning a matrix of ints representing the values of the
    tiles in the tiles list. </returns>
    }
```

```

private int[,] ListToMatrix()
{
    int[,] table = new int[4, 4];
    foreach (Tile curTile in _tilesList)
    {
        table[curTile.to.X, curTile.to.Y] = curTile.Value;
    }
    return table;
}
#endregion
#region Drawing & Generating Tiles:
/// <summary>
/// General: The main function on this class which is in charge of drawing
the tiles and
///           the game state when the game ends (either on a win or a
loss).
/// Process: Drawing the tiles from the tiles list and animating their
movement by
///           incrementing Tile.ticks value each time this function is
called. Also, every
///           time this function is called, checking whether the user won
or not and
///           accordingly deciding whether to display a win/loss screen.
/// </summary>
public void DrawTiles()
{
    // Drawing every tile in the tiles list:
    _tilesList.ToList().ForEach(block => block.Draw());

    // Updating the counter of the animation until reaching max ticks:
    if (Tile.ticks < Tile.maxTicks)
    {
        // Adding 1 to the counter of the animation:
        Tile.ticks++;
    }

    // Finishing the movement of the tile - updating each of the tiles'
.from to .to value,
    // and if anything happened then also adding a new tile to the game:
    else
    {
        if (boardAI != AIState.None)
        {
            _lockKeyboard = false; // CHECK WITH AI STUFF
        }
        //_lockKeyboard = false; // CHECK WITH AI STUFF

        if (_boardChanged)
            _turnsCounter++; // Counting how many moves have been made by
the user/ai.

        // Setting the "from" property of every tile to it's "to" property
        // since we finished animating it's movement:
        _tilesList.ForEach(block => block.from = new Point(block.to.X,
block.to.Y));

        // Removing from the list any block that has the state "Delete":
    }
}

```

```

        _tilesList.RemoveAll((Tile x) => { return x.TileState == TileState.Delete; });

        // Adding the value of any block.upgrade to the score variable:
        _tilesList.Where(block => block.TileState == TileState.Upgrade)
            .ToList()
            .ForEach(block => Score += block.Upgrade());

        // Checking if the PLAYER won (Won't display "YOU WIN" in an AI run):
        if (!_won && _tilesList.Any(block => block.Value == 2048) &&
boardAI == AISTate.None || _tilesList.Any(block => block.Value == 32768))
        {
            _won = true;
            toDraw = DrawGameState.Win;
        }

        // Checking if the player lost:
        if (_tilesList.Count == 16)
            CheckIfLost();

        // If the board has changed, then it means that we should generate
a new tile
        // and set the _boardChanged bool to false again (so that in the
next turn we
        // will be able to determine whether the board chagned or not):
        if (_boardChanged)
        {
            _boardChanged = false;
            GenerateRandomTile();
        }
    }

    /// <summary>
    /// General: Generating and adding a new tile to the board.
    /// Process: Picking a random position on the board to put the new tile in
and adding it. While adding it,
    ///           a random value (90% -> 2 || 10% -> 4) will be generated for
the tile through the Tile's constructor.
    /// </summary>
    public void GenerateRandomTile()
    {
        // A random generator to generate random numbers:
        Random rnd = new Random();

        // Using a Sorted Set to maintain an ascending order of elements in
the range 0-15.
        SortedSet<int> available = new SortedSet<int>(Enumerable.Range(0,
16));

        // Removing all the taken slots from the available sorted set:
        foreach (var block in _tilesList)
        {
            available.Remove(block.to.X * 4 + block.to.Y);
        }
    }
}

```

```

        // Generating a random value between 0 to the count of the available
slots:
    var num = available.ElementAt(rnd.Next(0, available.Count()));

        // Adding a tile that receives a point [(num/4, num%4)]:
        _tilesList.Add(new Tile(new Point(num >> 2, num % 4)));
    }
#endregion

```

:Solver.cs – מחלקה פיתר – 2.

```

public class Solver
{
    public int[,] grid; // A variable that resembles the Solver's class board.

    #region Constructors:
    /// <summary>
    /// General: A Constructor to the Solver class.
    /// </summary>
    public Solver()
    {
        this.grid = new int[4, 4];
    }

    /// <summary>
    /// General: A copy Constructor to the Solver class.
    /// </summary>
    /// <param name="s"> A Solver to copy. </param>
    public Solver(Solver s)
    {
        this.grid = new int[4, 4];
        // Copying all the values from the received Solver's grid to the
        current state's grid:
        for (int r = 0; r < 4; r++)
            for (int c = 0; c < 4; c++)
                this.grid[r, c] = s.grid[r, c];
    }
#endregion

    #region Random AI Algorithm:
    /// <summary>
    /// General: A Random AI.
    /// Process: Randomly picking a move to perform.
    /// </summary>
    /// <returns> A Key representing a move to perform. </returns>
    public static Keys RandomAI()
    {
        // Variables Definition:
        Random rnd = new Random();
        int val = rnd.Next(0, 4); // Randomly picking a number between 0 to 3,
each value represents a Key.
        Keys key = Keys.None;

        // Using a switch case to determine what key was randomly picked:
        switch (val)
        {

```

```

        case 0:
            key = Keys.Left;
            break;
        case 1:
            key = Keys.Right;
            break;
        case 2:
            key = Keys.Up;
            break;
        case 3:
            key = Keys.Down;
            break;
    }

    // Returning the chosen key:
    return key;
}
#endifregion

#region Performing Movement:
/// <summary>
/// General: The main function which is in charge of performing a move.
/// Process: "Pushing" the board towards the direction represented by the
received key.
///           Using an outer loop which runs 4 times which resembles every
row/column in
///           the board, each time collecting the values of a specific
row/column, then
///           performing the movement on them and eventually inserting them
back to the
///           table which represents the board.
/// </summary>
/// <param name="key"> A Key representing a direction to move the board
towards. </param>
public void pushBoard(Keys key)
{
    // An outer loop that runs 4 times (the amount of rows/cols in the
board):
    for (int t = 0; t < 4; t++)
    {
        // Defining a list of ints (max - 4 items) to contain all the
values which
        // are different than 0 in the current row.
        List<int> items = new List<int>(4);
        switch (key)
        {
            #region Movement Left:
            case Keys.Left:
                for (int c = 0; c < 4; c++)
                    if (this.grid[t, c] != 0)
                        items.Add(this.grid[t, c]);

                // Fusing tiles:
                for (int i = 0; i < items.Count - 1; i++)
                {
                    if (items[i] == items[i + 1])
                    {

```

```

        items[i]++;
        items.RemoveAt(i + 1);
    }
}

// Writing the fused tiles back to the row:
for (int c = 0; c < 4; c++)
    this.grid[t, c] = (c < items.Count ? items[c] : 0);
break;
#endregion

#region Movement Right:
case Keys.Right:
    for (int c = 0; c < 4; c++)
        if (this.grid[t, c] != 0)
            items.Add(this.grid[t, c]);

    //Consolidate duplicates
    for (int i = items.Count - 1; i > 0; i--)
    {
        if (items[i] == items[i - 1])
        {
            items[i]++;
            items.RemoveAt(i - 1);
            i--;
        }
    }

    //Write the data back to the row
    for (int i = 0; i < 4; i++)
        this.grid[t, 4 - 1 - i] = (items.Count - 1 - i >= 0 ?
items[items.Count - 1 - i] : 0);
    break;
#endregion

#region Movement Up:
case Keys.Up:
    for (int r = 0; r < 4; r++)
        if (this.grid[r, t] != 0)
            items.Add(this.grid[r, t]);

    //Consolidate duplicates
    for (int i = 0; i < items.Count - 1; i++)
    {
        if (items[i] == items[i + 1])
        {
            items[i]++;
            items.RemoveAt(i + 1);
        }
    }

    //Write the data back to the row
    for (int r = 0; r < 4; r++)
        this.grid[r, t] = (r < items.Count ? items[r] : 0);
    break;
#endregion

```

```

#region Movement Down:
case Keys.Down:
    for (int r = 0; r < 4; r++)
        if (this.grid[r, t] != 0)
            items.Add(this.grid[r, t]);

    //Consolidate duplicates
    for (int i = items.Count - 1; i > 0; i--)
    {
        if (items[i] == items[i - 1])
        {
            items[i]++;
            items.RemoveAt(i - 1);
            i--;
        }
    }

    //Write the data back to the row
    for (int i = 0; i < 4; i++)
        this.grid[3 - i, t] = (items.Count - 1 - i >= 0 ?
items[items.Count - 1 - i] : 0);
    break;
#endregion

#region ExpectiMax Algorithm & Calculations:
/// <summary>
/// General: The main AI function in the project. This function is in
charge of performing all the necessary
/// calculations and using all the relevant functions in order to
return a score representing the
/// score that the original received Solver deserves.
/// Process: This is a recursive function which is based on the ExpectiMax
Algorithm.
/// For full explanation in regards to how this Algorithm works,
please check my project portfolio.
/// To keep it simple, what basically happens in this recursive
function is that we create a Tree.
/// The "Tree" we create contains nodes, while each of the nodes
in the tree is a Solver representing
/// a possible outcome based on the original Solver. We calculate
the rating for each node in the tree and
/// then choose the AVERAGE of the childs of every node as the
node's value. We continue doing this
/// all the way up until reaching the root, which is the original
Solver that the function received.
/// For the root, we pick the MAX of it's childs, and then we
return it as the "bestValue" which
/// represents the BEST outcome based on the received Solver's
board.
/// </summary>
/// <param name="root"> A Solver that we want to get an ExpectiMax rating
for. </param>
/// <param name="depth"> The depth to which we want the ExpectiMax
algorithm to reach. </param>

```

```

    /// <param name="player"> A bool to determine whether it's the "Player's"
(AI's) turn or the opponent (Computer). </param>
    /// <returns> Returning the rating for the original received Solver.
</returns>
    public static double ExpectiMax(Solver root, int depth, bool player)
    {
        // Variables Definition:
        double bestValue = 0, val = 0;

        // If depth is 0 then we can stop the recursion and return the root's:
        if (depth == 0)
            return root.GetRating();

        // The algorithm's turn:
        if (player)
        {
            // Setting bestValue as the min value of double at the start of
each calculation:
            bestValue = double.MinValue;

            // Initializing moves with all the possible moves for the current
state:
            Dictionary<Keys, Solver> moves = root.getAllMoveStates();

            // If we can't move then the game is over - WORST CASE:
            if (moves.Count == 0)
                return double.MinValue;

            // Calculating the bestValue for every possible state in the
board:
            foreach (KeyValuePair<Keys, Solver> st in moves)
            {
                // Recursion - Explained:
                // First, recursively calling the ExpectiMax algorithm to
continue it's run
                // until reaching depth = 0. Then, when we get a result,
choosing the better
                // option between the 2: val, bestValue. val represents the
current value
                // received from the recursive call to ExpectiMax and
bestValue represents
                // the highest value received so far. This, combined with the
call to the
                // getAllMoveStates function which returns a list of all the
possible moves
                // for the current state, allows us to pick the best move to
perform in the
                // current Solver's state.

                // Summary:
                // Basically, what we do here is to get the best possible move
from a list of
                // moves that contains all the possible moves for the current
state.
                val = ExpectiMax(st.Value, depth - 1, false);
                bestValue = Math.Max(val, bestValue);
            }
        }
    }
}

```

```

        // "Computer" turn:
    else
    {
        // Getting all the possible states based on a given move:
        List<Solver> moves = root.getAllRandom();
        int i = 0;
        foreach (Solver st in moves)
        {
            // Math - Explained:
            // We use the variable i to help us determine the INDEX of the
            child in the "tree" we created.
            // Since we built the tree in such way that we have both a
            Solver with the value 2 and a Solver with
            // the value 4 initialized for every FREE CELL in the board,
            we know that childs with even index
            // will be initialized with the value 1 (representing the
            value 2), while those with an odd index
            // will be initialized with the value 2 (representing the
            value 4). We know that in the game itself,
            // the chances of getting 2 are 90% while the chances of
            getting 4 are 10%, therefore we can determine
            // by the index what we should multiply by (0.9 OR 0.1). We
            also multiply by 1 divided by half of the
            // possible moves (which represents the amount of even/odd
            nodes). At last, we multiply by the result
            // of the recursive call to Expectimax with depth-1 and the
            player being set to true.

            // Summary:
            // To sum it up, this part is in charge of getting the average
            of every level in the tree:
            bestValue += (1.0 / (moves.Count / 2) * ((i % 2 == 0) ? 0.9 :
            0.1)) * ExpectiMax(st, depth - 1, true);
            i++; // Index of child in the "tree".
        }
    }

    // Returning the bestValue:
    return bestValue;
}

/// <summary>
/// General: Calculating the rating for a given "node" in the "tree" we
created.
/// Process: Using 2 for loops to scan the board and setting "weight" to
each node to keep the most
///           significant node, the node with the highest valued tile, at
the top right corner of the board.
/// Math: Basically, we rate a "node", which represents a possible
board as a result of a movement here.
///           In order to rate the "node", we use the following
calculation: score = log2(x)*(1/4)^y,
///           having x = the value of every tile in the board, and y
representing the tile's index in the board.
///           This way, we can assure that on MOST cases, we will have the
most signifant tile (the tile with
///           the highest value) at the TOP RIGHT corner of the board.
/// </summary>

```

```

/// <returns> Returns a double representing the score that the current
Solver instance received. </returns>
public double GetRating()
{
    // Variables Definition:
    double score = 0, weight = 1;

    // Using 2 for loops to iterate through the board.
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            score += Math.Pow(2, this.grid[i, j]) * weight;
            weight *= 0.25;
        }
    }

    // Output - Returning the score that the "node" received:
    return score;
}
#endregion

```

ExpectiMax Helping Functions:

```

#region ExpectiMax Helping Functions:
/// <summary>
/// General: Finding all the possible moves which will affect the board.
/// Process: Performing a movement on a copy of the current Solver, and
/// checking whether
///           the copy of the Solver's board (after the movement) is
/// different from the original
///           Solver's board. If it's different, then adding it to a
/// Dictionary which contains
///           all the possible moves that will lead to a different Solver's
/// board from the current
///           Solver's board.
/// </summary>
/// <returns> Returns a Dictionary containing all the possible moves &
/// their matching Solvers. </returns>
public Dictionary<Keys, Solver> getAllMoveStates()
{
    // Creating a dictionary of up to 4 Keys-Solver pairs which we will
    return at the end of the function:
    Dictionary<Keys, Solver> allMoves = new Dictionary<Keys, Solver>(4);
    // Defining every possible move that can be performed:
    Keys[] possibleMoves = { Keys.Left, Keys.Right, Keys.Up, Keys.Down };

    // Creating an instance which will be used to replicate the current
    Solver:
    Solver next;

    // Generating and storing the board's state after "movement" to the
    direction that key points
    // towards. Checking the board's state after performing "movement" to
    every direction:
    foreach (Keys key in possibleMoves)
    {
        next = new Solver(this);
        next.pushBoard(key);
    }
}

```

```

        if (!this.equalTo(next))
            allMoves.Add(key, next);
    }

    // Returning the dictionary with up to 4 Keys. The Dictionary contains
the state of
    // the board after performing every possible movement at a given time,
if the movement
        // causes any changed to the board:
        return allMoves;
    }

    /// <summary>
    /// General: Finding all the FREE cells in the current Solver's board.
    /// Process: Creating a list of points to which we add values by using 2
for loops
        // to scan the current Solver's board, each time checking if the
current
        // value in the Solver's board isn't initialized (it's value is
0). If it
        // isn't initialized, then adding a point representing the cell
to the list
        // of points we created.
    /// </summary>
    /// <returns> Returning a list of points representing the indexes of the
uninitialized cells in the current Solver's board. </returns>
    private List<Point> GetFreeCells()
    {
        // Variables Definition:
        List<Point> free = new List<Point>(); // A list of points to store the
indexes of the "free" cells.

        // Using 2 for loops to add every empty cell from the grid to the
list:
        for (int r = 0; r < 4; r++)
            for (int c = 0; c < 4; c++)
                if (this.grid[r, c] == 0)
                    free.Add(new Point(r, c));

        // Output - a list of the "free" cells:
        return free;
    }

    /// <summary>
    /// General: Generating a list of Solvers with boards representing each
possible outcome as
        // a result of a new tile being added to the current Solver's
board.
        // Process: First, getting all the free cells in the board using the
GetFreeCells we created
        // above. Then, for each free cell in the board, generating 2
COPY Solvers on which
        // at the value in the index of the free cell is initialized to
"1" (representing 2)
        // or initialized to "2" (representing 4).
    /// </summary>
    /// <returns> Returning a list of Solvers containing all the Solvers for
every free cell. </returns>

```

```

public List<Solver> getAllRandom()
{
    // Variables Definition:
    List<Solver> res = new List<Solver>(); // A new list of Solver.
    List<Point> free = this.GetFreeCells(); // A list of all the free
cells in the board.
    Solver next = new Solver();

    // Generating the board's state twice:
    // 1. In case the current cell's value will be 2.
    // 2. In case the current cell's value will be 4.
    foreach (Point curPoint in free)
    {
        next = new Solver(this);
        next.grid[curPoint.X, curPoint.Y] = 1;
        res.Add(next);

        next = new Solver(this);
        next.grid[curPoint.X, curPoint.Y] = 2;
        res.Add(next);
    }

    // Returning the list of states generated for each free cell in the
board:
    return res;
}

/// <summary>
/// General: Checking if the current Solver is equal to another Solver
instance.
/// Process: Checking whether ALL the cells in the current Solver's grid
are equal
///           to the other Solver's grid.
/// </summary>
/// <param name="another"> Another Solver to check whether it's equal to
the current Solver or not. </param>
/// <returns> Returns True if both Solvers grids are equal or False if
not. </returns>
public bool equalTo(Solver another)
{
    for (int r = 0; r < 4; r++)
        for (int c = 0; c < 4; c++)
            if (this.grid[r, c] != another.grid[r, c])
                return false;

    return true;
}
#endregion

/// <summary>
/// General: A public ToString to help the debugging process be easier.
/// </summary>
/// <returns></returns>
public override string ToString()
{
    StringBuilder sb = new StringBuilder();
    for (int r = 0; r < 4; r++)
    {

```

```

        for (int c = 0; c < 4; c++)
    {
        string number = "";
        if (this.grid[r, c] != 0)
        {
            number = ((int)Math.Pow(2, this.grid[r, c])).ToString();
        }
        sb.Append(number.PadLeft(5, ' '));
    }
    sb.AppendLine();
}

sb.AppendLine(" ----- ----- ----- -----");
return sb.ToString();
}

```

3. מחלקה אריה :Tile.cs – מחלקה אריה

```

public enum TileState
{
    Normal,
    Upgrade,
    Delete
}
public class Tile
{
    #region Variables Definition:
    // _size = Size of the tile's "Physical Body", _gap = Distance between
    each tile on the board,
    // _boardGap = Distance between board boundaries to tiles.
    private Point _size = new Point(80, 80), _gap = new Point(11, 12),
    _boardGap = new Point(12, 13);
    private Rectangle _tileRect; // A variable to represent the "physical
    body" of the tile.
    private int _value = 0; // A variable to store the tile's value.
    private TileState _tileState; // A variable to represent the tile's state.
    public Point from, to; // Defining the points from where we move the tile
    and to where we move it.

    public static SpriteBatch spriteBatch; // Drawing with the same
    spriteBatch from the SceneManager.
    public static int ticks = 0; // A counter to "animate" every tile's
    movement by moving each tile maxTicks times.
    public static int maxTicks = 8; // The amount of times we want to
    "animate" the tile's movement.
    private static Dictionary<int, Texture2D> _tilesTextures; // A dictionary
    to store tiles textures.
    #endregion

#region Getters & Setters:
// Defining necessary Getters & Setters:
public int Value { get => _value; set => _value = value; }

```

```

    public TileState TileState { get => _tileState; set => _tileState = value;
}

// A Setter to the textures of the tiles:
public static void SetTextures(Dictionary<int, Texture2D> dict)
{
    _tilesTextures = dict;
}
#endregion

#region Constructor:
/// <summary>
/// General: A Constructor to the Tile class.
/// </summary>
/// <param name="point"> A point by which we will initialize the Tile
class. </param>
public Tile(Point point)
{
    Random rnd = new Random(); // Defining a random instance to randomly
choose a value for a new tile.

    // Choosing a random value for the new tile (90% - 2, 10% - 4):
    _value = rnd.Next(0, 10) < 9 ? 2 : 4;

    // Initializing the to & from instances
    from = new Point(point.X, point.Y);
    to = new Point(point.X, point.Y);

    _tileRect = new Rectangle(Board.offset, this._size);

    // Setting the state of the block to normal:
    _tileState = TileState.Normal;

    // Creating a "physical body" for the tile:
    GetRect();
}
#endregion

#region Helping Functions:
/// <summary>
/// General: Upgrading a tile.
/// Process: Changing the tile's state to normal & it's value to double
it's previous value.
/// </summary>
/// <returns> Returns the value of the upgraded tile. </returns>
public int Upgrade()
{
    Value = Value << 1; // value = value * 2
    TileState = TileState.Normal; // Setting the tile state to normal.
    return Value; // Returning the value from the upgrade function (the
value of the tile).
}

/// <summary>
/// General: Initializing the positions of the tile's X and Y values.
/// Process: Deciding where to "put" the tile based on the _gap, _size,
_boardGap and tiles animated move.

```

```

/// </summary>
/// <returns> Returns the rectangle created for the new tile. </returns>
public Rectangle GetRect()
{
    // _boardGap = the gap from the board sides.
    // Board.offset.x / Board.offset.y = The locations of the board in the
background.
    // The rest = Calculations to get the position of the tile in the
board:
    _tileRect.X = _boardGap.X + Board.offset.X + (from.Y * (maxTicks -
ticks) + to.Y * ticks) * (_size.X + _gap.X) / maxTicks;
    _tileRect.Y = _boardGap.Y + Board.offset.Y + (from.X * (maxTicks -
ticks) + to.X * ticks) * (_size.Y + _gap.Y) / maxTicks;
    return _tileRect;
}

/// <summary>
/// General: Drawing the image of the tile which matches the value of the
current tile.
/// </summary>
public void Draw()
{
    spriteBatch.Draw(_tilesTextures[Value], GetRect(), Color.White);
}

/// <summary>
/// General: Checking whether a given tile equals to the current tile.
/// Process: Checking whether both of the tiles values are equal to each
other's value and whether their state is equal to normal.
/// </summary>
/// <param name="other"> Another tile to check if it's equal to the
current tile or not. </param>
/// <returns></returns>
public bool Equals(Tile other)
{
    return this.Value == other.Value && this.TileState == other.TileState
        && this.TileState == TileState.Normal;
}
#endifregion

```