

עבודת גמר

לקבלת תואר טכנאי תוכנה



נושא: שימוש בבינה מלאכותית לפתירת המשחק 2048.

מגיש: ארז דרוטין

שמות המנחים: מיכאל צ'רנובלסקי ואלון חיימוביץ

תש"ף

אפריל 2020

תוכן עניינים

- הצעת נושא לפרויקט.....3
- לוחות זמנים פרויקט י"ג.....6
- מבוא.....7
 - מטרה.....7
 - תיאור המערכת.....7
 - שפת תכנות ופירוט סביבת העבודה והכלים.....8
- מפרטי תוכנה.....9
 - תיאור כללי.....9
 - ניקוד במשחק.....10
 - תזוזה במשחק.....10
 - מהלך במשחק.....10
 - ניסוח וניתוח הבעיה האלגוריתמית.....11
 - פיתוח הפתרון ויישומו.....13
 - תיאור אלגוריתמים.....19
 - מבנה נתונים.....25
- תרשים מחלקות – UML Class Diagram.....26
- תכנון.....27
- מדריך למשתמש.....35
- ביבליוגרפיה.....41
- נספחים.....42

הצעת נושא לפרויקט

נושא הפרויקט: משחק מחשב אשר מדמה את המשחק 2048, שמטרתו היא "להצמיד" כמה שיותר זוגות מספרים ובעקבות כך להגיע לתוצאה הגבוהה ביותר תוך כדי ניסיון להשיג משבצת עם הערך 2048, אשר קובעת את סיום המשחק וניצחון המשתמש. בנוסף, המשחק יגמר כאשר לא יהיה ניתן להצמיד עוד מספרים, והמשתמש יפסיד. המשחק יאפשר למשתמש לשחק במשחק בגרסתו הרגילה (כשחקן יחיד) או מול המחשב.

גודל הלוח: 4x4.

תזוזה במשחק – ביצוע תור:

חץ למעלה: מעלה את המספרים בלוח כלפי מעלה.

חץ למטה: מוריד את המספרים בלוח כלפי מטה.

חץ שמאלה: מזיז את המספרים בלוח כלפי צד שמאלה.

חץ ימינה: מזיז את המספרים בלוח כלפי ימינה.

התרחשות המשחק:

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

מצב ניצחון – מצב שבו הושגה משבצת שערכה הוא 2048.

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

ניקוד במשחק:

הניקוד במשחק מתקבל באמצעות הצמדת זוגות של מספרים זהים. בעת תזוזה, כלומר לחיצה על אחד ממקשי החיצים, יכולים להיווצר מספר זוגות באותה העת. בנוסף, בעת היווצרות זוג מספרים, תופיע תוצאת החיבור של המספרים במקום זוג המספרים (במיקום שאליו החץ סימן), ונוסף על כך הניקוד של המשתמש יעלה בתוצאת החיבור שלהם.

המצבים השונים במשחק:

אחרי הצמדה:

חץ שמאלה + חץ מטה + חץ מטה

			2
8			
8	2		2

לפני הצמדת מספרים:

אמצע משחק – לפני הצמדה

4			
2			
2			
4	4		

מצב התחלתי:

לפני ביצוע מהלך כלשהוא

2		2	

מצב הפסד:

אין מהלכים תקינים לביצוע

2	4	16	2
8	128	256	8
2	32	64	2
16	4	2	8

מצב ניצחון:

הגעה לאריח 2048

	4	4	2048
	2	8	
4		2	

אסטרטגיות פעולה:

1. **לנסות להוריד כמה שיותר צמדים באותו תור:** לדוגמא, אם ניתן להוריד 2 צמדים בבת אחת באמצעות לחיצה על החץ למטה ואילו ניתן להוריד צמד אחד באמצעות לחיצה על החץ שמאלה, עדיף ללחוץ על החץ מטה, שכן כך נוכל "לנקות" את המסך, מה שיאפשר לנו "לשרוד" יותר זמן.

2	2		
2	4		
2	4	2	
8	64	8	4

2. **עדיף להוריד צמד מספרים גדולים על פני צמד מספרים קטנים:** אם ניתן להוריד צמד של מספרים גדולים (כגון 128) לעומת צמד של מספרים קטנים יותר (כגון 2), לרוב נעדיף להוריד את צמד המספרים הגדול יותר שכן קשה יותר להוריד אותו באופן כללי מאשר להוריד צמד מספרים קטנים (כי צריך לבצע יותר פעולות על מנת להגיע לכל מספר גבוה).

2	4	2	16
2	16	128	8
		128	2
	8	2	4

3. **כדאי להיצמד לפינות הלוח:** כיוון שמספרים יופיעו באופן אוטומטי במקומות רנדומליים ברחבי המסך לאחר ביצוע כל פעולה, מומלץ לבחור באחת מפינות הלוח ולהיצמד אליה.

8	8		
128	2		
4			2

תיאור המערכת:

- מימוש GUI:
 - הצגת לוח המשחק בצורה ויזואלית אשר יתעדכן לאחר כל תור.
 - יצירת אנימציה בעת הצגת תורות המשתמש.
 - הצגת ניקוד המשתמש בכל עת והדפסת הודעה מתאימה כאשר המשחק מסתיים.
 - אפשרות למשחק חוזר, או לחלופין חזרה למסך הפתיחה של המשחק.
- יישום bot שישתמש באלגוריתם מורכב לצורך מימוש שחקן 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 משתמש ממוחשב.
3. מימוש גרסה "חדשה" של המשחק 2048, אשר תאפשר למשתמש לשחק מול bot שיהווה שחקן ממוחשב.

עמדת פיתוח:

- מערכת הפעלה - Windows.
- סביבת עבודה – Visual Studio 2017.

דרישות מהמשתמש:

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

לוחות זמנים פרויקט י"ג

מועד ביצוע	משימה	
09/11/2019	בחירת פרויקט הגשת נוסח ראשוני (First Draft) – הצעת ותיאור הפרויקט (Use & PRD Cases)	1
17/11/2019	הגשת נוסח סופי (final draft) - הצעת ותיאור הפרויקט (Use & PRD Cases) אישור ראשוני	2
01/12/2019	הגשה למשרד החינוך ואישור	3
20/12/2019	מימוש GUI מרכזי של המשחק – מימוש לוח המשחק, מימוש פעולות במשחק, מימוש חוקי המשחק ועיצוב מסך המשחק (כולל אנימציות רלוונטיות).	4
10/01/2020	תחילת פיתוח האלגוריתם – הגדרת האלגוריתם החכם שנשתמש בו לצורך יצירת bot שיהווה השחקן הממוחשב.	5
26/01/2020	דו"ח	6
01/02/2019	המשך פיתוח האלגוריתם – תחילת מימוש האלגוריתם החכם במשחק שישמש בbots.	7
23/02/2020	הגשת דו"ח ביניים – התקדמות הפיתוח עד כה	8
סוף מרץ 2020	שדרוג כללי של התוכנה – בדיקת מצבי קצה, תיקון בעיות, שיפור חווית משתמש, ועוד.	9
אפריל 2020	סיום פיתוח (Code Freeze)	10
אפריל 2020	בחינת מתכונת כולל תיק פרויקט	11
אפריל 2020	סיום תיקון שגיאות (Bug Freeze) - סופי	12
אפריל 2020	הגשת תיק מלווה של הפרויקט סופי	13
אפריל-מאי 2020	הגנה על עבודת הגמר – פנימית - סופית	14
אפריל-מאי 2020	הגנה על עבודת הגמר - חיצונית	15

מבוא:

מטרה:

פיתוח המשחק 2048 בשפת התכנות C# באמצעות framework שפותח ע"י מייקרוסופט עבור פיתוח משחקים לסביבת NET. הנקרא MonoGame ופיתוח בינה מלאכותית לשחקן ממוחשב תוך שימוש באלגוריתמים ובמבני נתונים מורכבים ויעילים. אחת המטרות המרכזיות שלי היו ללמוד כלים לשיפור חווית המשתמש אשר ישפרו את יכולותי כמפתח ויעזרו לי בעתיד.

תיאור המערכת:

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

המשחק בנוי מלוח משבצות בגודל 4X4 עם אריחים ממוספרים בצבעים שונים. על מנת להזיז את האריחים, השחקן משתמש בחצי המקלדת (או בטלפון – 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 **אקראית** על גבי הלוח.

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

במערכת שלי יש 3 אפשרויות למשתמש והן:

1. משחק רגיל, אשר פועל בצורה זהה למשחק המקורי 2048.
2. משחק מול מחשב, אשר מאפשר למשתמש לשחק מול שחקן ממוחשב.
3. משחק ממוחשב, אשר מאפשר לשחקן לצפות במחשב משחק ובכך להפיק לקחים ואסטרטגיות ממשחק המחשב.

מראה לוח המשחק:

64	32	4	2
8			
4			

שפת התכנות ופירוט סביבת העבודה והכלים:

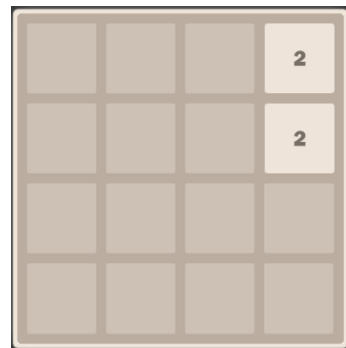
פיתוח המערכת נעשה בשפת התכונה C# באמצעות framework שפותח ע"י מיקרוסופט עבור פיתוח משחקים בסביבת NET. הנקרא MonoGame (קיים ב.NET framework 4.X ומעלה). הפיתוח התבצע על גבי Microsoft Visual Studio 2017, ובמהלכו התבצע שימוש בכלים שסביבת העבודה NET. מציעה.

מפרטי תוכנה:

תיאור כללי:

המשחק 2048 הינו משחק חשיבה המשוחק על לוח בגודל 4x4 והוא מיועד לשחקן יחיד כאשר מטרת השחקן היא להגיע לאריח בעל הערך 2048. המשחק מתחיל כאשר 2 אריחים בעלי הערך 2 או 4 (כאשר יש 90% לכל אריח להיות 2 ו-10% להיות 4) מופיעים במיקום רנדומלי על גבי לוח המשחק. המשחק מסתיים כאשר השחקן מגיע לאריח עם הערך 2048 (ניצחון) או כאשר לשחקן אין עוד מהלכים שהוא יכול לבצע (הפסד).

מראה הלוח בתחילת המשחק:



מראה הלוח בסיום המשחק (ניצחון):



מראה הלוח בסיום המשחק (הפסד):



ניקוד במשחק:

הניקוד במשחק מתקבל באמצעות הצמדת זוגות של מספרים זהים. בעת תזוזה, כלומר לחיצה על אחד ממקשי החיצים, יכולים להיווצר מספר זוגות באותה העת. בנוסף, בעת היווצרות זוג מספרים, תופיע תוצאת החיבור של המספרים במקום זוג המספרים (במיקום שאליו החץ סימן), ונוסף על כך הניקוד של המשתמש יעלה בתוצאת החיבור שלהם.

תזוזה במשחק:

התזוזה מתבצעת באמצעות חיצ המקלדת באופן הבא:

חץ מעלה: מזיז את האריחים בלוח כלפי מעלה.

חץ מטה: מזיז את האריחים בלוח כלפי מטה.

חץ שמאלה: מזיז את האריחים בלוח כלפי צד שמאל.

חץ ימינה: מזיז את האריחים בלוח כלפי צד ימין.

מהלך במשחק:

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

בעת ביצוע תזוזה, 2 אריחים עשויים להתמזג (Fuse) לאריח אחד (שיופיע בהתאם לכיוון החץ שגרם להתמזגות האריחים) במידה וערכם זהה והם צמודים זה לזה. ערך האריח שיתקבל מהמתזגות שני האריחים יהיה סכום 2 האריחים שהתמזגו.

מצב ניצחון – מצב שבו הושג אריח עם הערך 2048.

מצב הפסד – מצב שבו לא ניתן לבצע מהלכים נוספים, כלומר הלוח מלא באריחים ואין זוג אריחים שיכולים להתמזג.

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

ניסוח וניתוח הבעיות האלגוריתמיות:

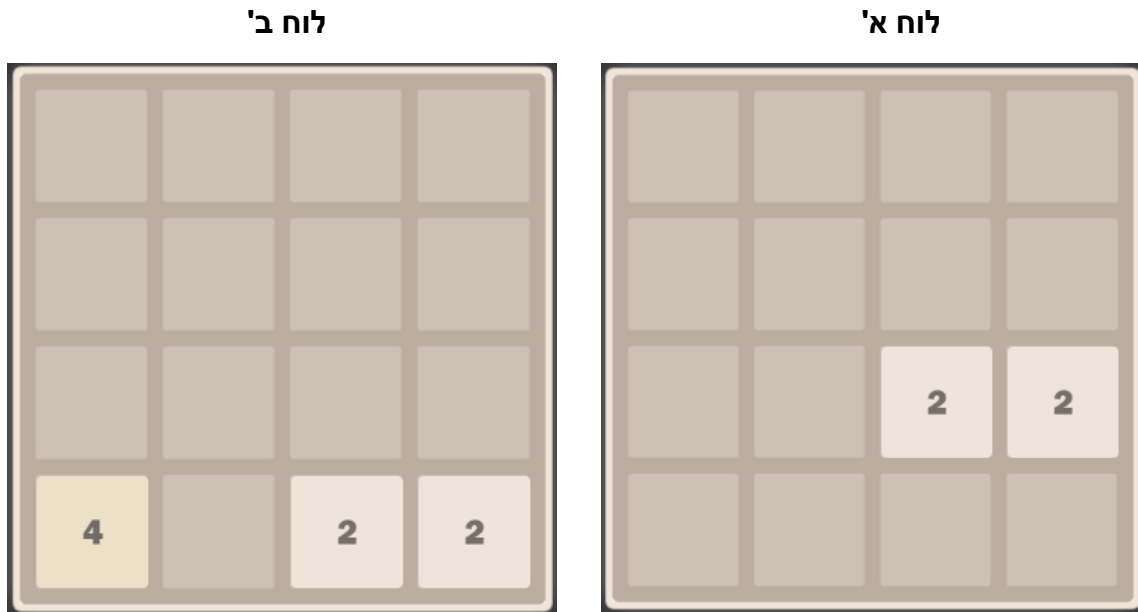
בעיה אלגוריתמית: תיאום מערכת לבחירות השחקן

הבעיה האלגוריתמית שעומדת מאחורי תיאום מערכת לבחירת השחקן נובעת מהצורך לאפשר למשתמש לבחור דרך מבין מספר דרכים שונות לשחק במשחק: משחק רגיל, משחק של שחקן ממוחשב ומשחק של שחקן אנושי נגד שחקן ממוחשב. בהתאם לבחירת המשתמש, עלינו להפעיל פעולות שונות אשר יתאימו בין בחירת המשתמש לבין מה שצריך להציג לו. לדוגמא, אם השחקן בחר לשחק נגד שחקן ממוחשב, אז על המערכת ראשית לפעול כשחקן ממוחשב ולאחר מכן לאפשר לשחקן האנושי לשחק ולנסות להביס את השחקן הממוחשב.

מעבר לסוג המשחק שנבחר ע"י השחקן, המערכת צריכה להתחשב בבחירותיו בנוגע לרמת הקושי של האלגוריתמים ולסוג האלגוריתם הנבחר. לדוגמא, כאשר השחקן מעוניין לצפות באלגוריתם (AI) משחק, עליו לבחור את סוג האלגוריתם (Random AI / ExpectiMax 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. השחקן יכול לזוז אך ורק באמצעות חיצ'י המקלדת כאשר כל חץ מסמל תזוזה לכיוון שונה אשר משפיעה על הזזת כל האריחים על הלוח לכיוון החץ. במשחק, כל מהלך נחשב לתקין פרט למקרים בהם התזוזה לא משפיעה על הלוח המשחק. לדוגמא, ניקח את 2 לוחות המשחק הבאים:



ניתן לראות כי **בלוח א'** השחקן יכול לנוע לכל כיוון (מעלה, מטה, ימינה או שמאלה), וזאת מכיוון שכל תזוזה תגרום לשינוי בלוח, אשר בעקבותיו גם יתווסף אריח חדש למשחק.

מנגד, **בלוח ב'** השחקן יכול לנוע לכל הכיוונים פרט למטה (מעלה, ימינה או שמאלה). זאת מכיוון שתזוזה למטה לא תשפיע על הלוח, משום שכל האריחים בזמן הנתון כבר צמודים למטה ולכן תזוזה מטה לא תשפיע עליהם.

בעיה אלגוריתמית: התמזגות אריחים

כיוון שבמשחק 2048, אריחים בעלי ערך זהה אשר נפגשים זה עם זה יכולים להתמזג לאריח אחד, הייתי צריך לחשוב על דרך שתאפשר לי לזהות מתי זוג אריחים צריכים להתמזג לאריח אחד שערכו שווה לסכום 2 האריחים. אם נתבונן שוב בלוחות א' וב' שהצגתי בעמוד הקודם, נראה כי בשניהם יש 2 זוגות אריחים בעלי ערך זהה (2 הזוגות בעלי הערך 2) אשר יכולים **וצריכים** להתמזג לאריח אחד שערכו יהיה 4 במידה והמשתמש יבחר לנוע לצד ימין או שמאל (כיוון שתנועה לאחד מהכיוונים הללו תגרום להתמזגות האריחים). על מנת להתמודד בעיה זו, הייתי צריך לבדוק מתי והאם כל זוג אריחים בלוח יכולים להתמזג, ומתי לא. בעקבות כל מהלך מספר זוגות אריחים יכולים להתמזג במקביל.

בעיה אלגוריתמית: אלגוריתם לשחקן ממוחשב

אחת הדרכים הנפוצות ביותר לחישוב מהלך במשחק לוח עם "מזל" (כלומר, בעקבות כל מהלך יש הסתברות לקבל תוצאה רנדומלית ולא חד משמעית) הוא באמצעות האלגוריתם מונטה קרלו. עם זאת, לאחר מחקר מעמיק וניתוח תוצאות שפורסמו ברחבי האינטרנט, הגעתי למסקנה שהאלגוריתם היעיל והטוב ביותר לצורך מימוש שחקן ממוחשב במשחק 2048 הוא האלגוריתם ExpectiMax. אלגוריתם זה מתבסס על האלגוריתם המוכר MiniMax, אך מעט מסובך יותר ומותאם למשחקים עם "מזל", כמו המשחק 2048. בעוד שבMiniMax פעם בוחרים את הצומת בעלת הערך המינימלי ופעם את הצומת בעלת הערך המקסימלי עד שמגיעים לשורש, באלגוריתם ExpectiMax בוחרים בכל פעם את ממוצע הבנים של כל צומת, ובסוף כאשר מגיעים לשורש בוחרים את הבר ששערכו הוא הגבוה ביותר.

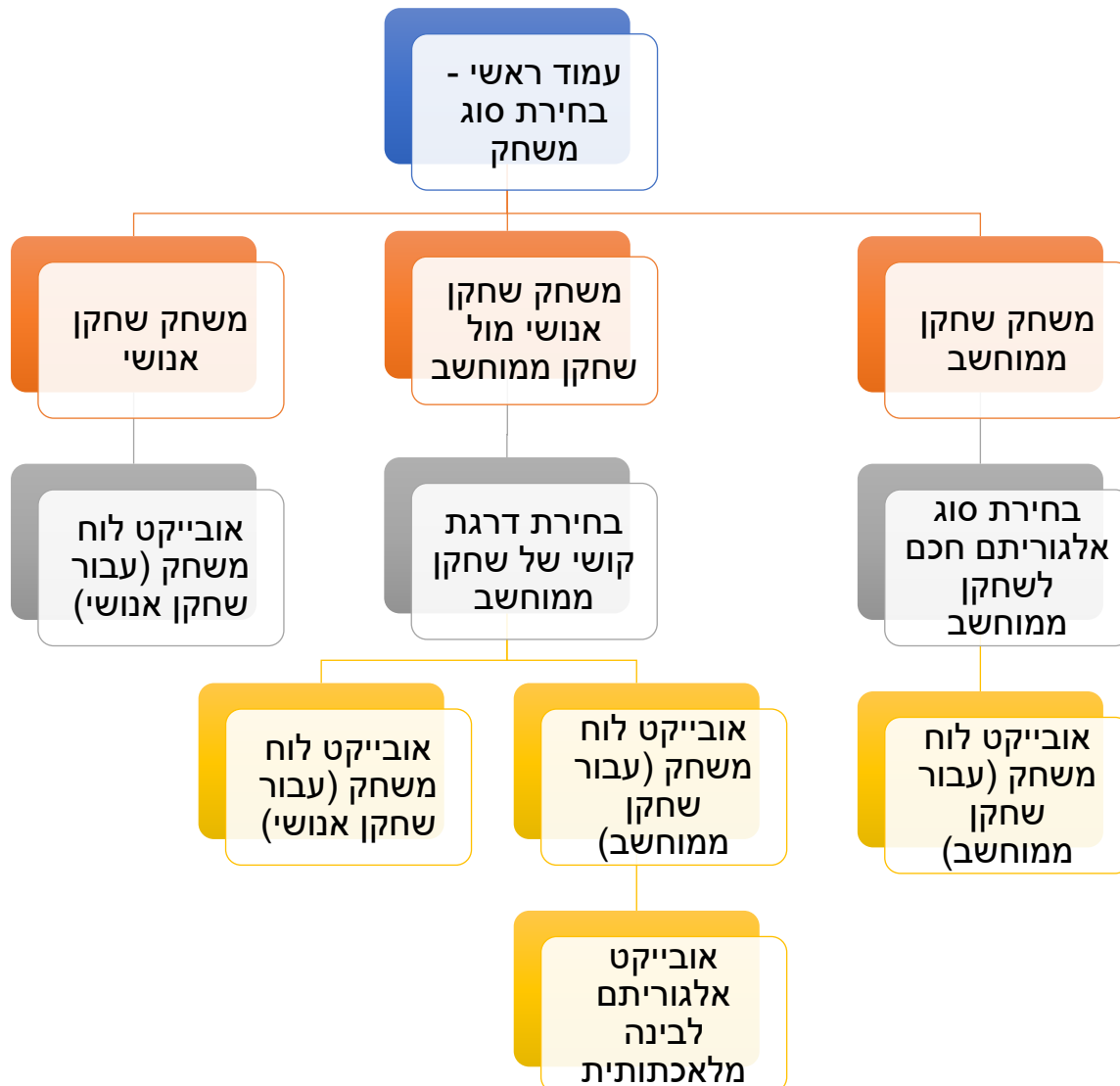
פיתוח הפתרון ויישומו:

פיתוח פתרון לבעיה אלגוריתמית: תיאום מערכת לבחירות השחקן

במערכת המשחק ישנן 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 הגבוהה ביותר (ובכך מאפשר לשחקן האנושי ללמוד ולסגל לעצמו אסטרטגיות משחק ע"י צפייה בו).

על המערכת ליצור תיאום בין בחירות המשתמש לבין המתרחש במערכת, לכן לשם פתרון בעיה זו החלטתי ליצור מחלקה אחת ראשית אשר מטרתה תהיה לנהל את המעברים בין "הסצנות" השונות במשחק. למחלקה זו קוראים SceneManager, והיא בנויה בצורה גנרית מאוד בכדי שניתן יהיה להוסיף סצנות למשחק בקלות. התרשים הבא מתאר את צורת פעילות מחלקת SceneManager באופן כללי:



בזכות המחלקה SceneManager אשר עוזרת לנהל את הכל, ניתן לראות כיצד בעקבות כל אחת מבחירות המשתמש נוצר אובייקט / נטען עמוד בהתאם לבחירה. בהתאם לבחירות המשתמש המשחק הרלוונטי יתחיל. לדוגמא, אם המשתמש בחר במשחק רגיל – יוצר לוח משחק ויטען עמוד המשחק הרגיל.

פיתוח פתרון לבעיה אלגוריתמית: בדיקת תקינות מהלך

כאשר המשתמש לוחץ על אחד מחיצי המקלדת בזמן משחק, מתבצעת בדיקת תקינות למהלך שביצע. במידה והמהלך שביצע עתיד להשפיע על הלוח, המהלך יחשב לתקין. אחרת, המהלך יחשב ללא תקין. בעקבות זאת, כתבתי אלגוריתם שבודק את כל השפעת התזוזה שנבחרה ע"י המשתמש על לוח המשחק, ובהתאם לתזוזה שיבצע המשתמש האלגוריתם יחזיר האם הלוח השתנה או לא. על מנת לזהות אם המהלך שביצע המשתמש תקין או לא, נעזרתי בלולאות ובמשתנה מסוג bool שמאוחל עם הערך false לבדיקה האם בעקבות המהלך שהמשתמש ביצע הלוח השתנה או לא. באמצעות הלולאות עברתי על גבי כל האריחים בלוח ובדקתי כיצד הלוח מושפע מתזוזת המשתמש. במידה והלוח השתנה כתוצאה מתזוזת המשתמש, ערכו של המשתנה מסוג bool הפך לtrue. במידה ובסיום המעבר באמצעות הלולאות על הלוח ערכו של המשתנה מסוג bool לא השתנה, אז הלוח לא הושפע כתוצאה מהמהלך והמהלך נחשב ללא תקין (ולכן גם לא נספר כמהלך).

פיתוח פתרון לבעיה אלגוריתמית: התמזגות אריחים

על מנת להתמודד עם בעיה אלגוריתמית זו, הייתי צריך לחשוב על דרך לזהות מתי זוג אריחים מתמזג, ולהשתמש בדרך זו על גבי כל הלוח כיוון שמספר זוגות אריחים יכולים להתמזג בעקבות מהלך אחד של המשתמש. לדוגמא, נתבונן בלוח הבא:

		4	2
			2
2		2	4
		8	4

בעקבות תזוזה מטה/מעלה, נקבל ש2 זוגות אריחים יתמזגו. זוג האריחים שערכם 2 בצד ימין למעלה יתמזג לאריח שערכו 4 וזוג האריחים שערכם 4 בצד ימין למטה יתמזג לאריח שערכו 8.

בעקבות זאת, פיתחתי דרך ראשית לזהות מתי זוג אריחים מתמזג לאריח אחד, ואז נעזרתי בלולאה לזהות את כל זוגות האריחים שמתמזגים בעקבות התזוזה.

פיתוח פתרון לבעיה אלגוריתמית: אלגוריתם לשחקן הממוחשב

הבעיה האלגוריתמית המרכזית בפרויקט היא פיתוח הבינה המלאכותית לצורך מימוש שחקן ממוחשב, ועל כן היא גם הייתה המסובכת ביותר. על מנת להתמודד איתה, ראשית קראתי מאמרים שעוסקים באסטרטגיות למשחק 2048. הגעתי למסקנה שהאסטרטגיה הטובה ביותר במשחק 2048 היא לשמור על האריח בעל הערך הגבוה ביותר בצד שמאל למעלה של הלוח, ובכל פעם להצמיד אליו את שאר האריחים בעלי הערכים הגבוהים ביותר.

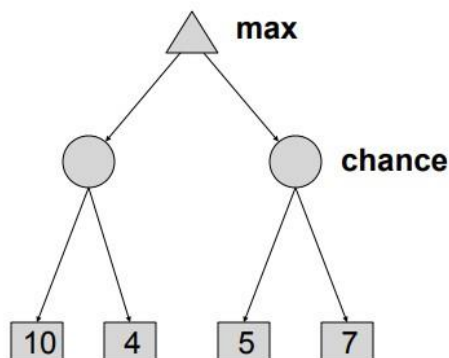
לדוגמא – ניתן לראות כיצד בלוח הבא הערך 256 (הגבוה ביותר) ממוקם בצד שמאל למעלה כאשר לצידו ממוקם המספר הגבוה ביותר הבא בתור:

256	128	32	4
		8	2
		8	2
			4

באמצעות חישובים מתמטיים, הגעתי לשיטה שבאמצעות מימושה עם האלגוריתם ExpectiMax ניתן יהיה לשמור על האריח בעל הערך 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 ראשית מחשב את ממוצע האופציות האפשריות לפעולה מסוימת, לאחר מכן עולה "שלב" בעץ ומבצע את אותו הדבר עד שהוא מגיע לשורש, כאשר עבורו הוא בוחר את הבן שערכו גבוה יותר. כלומר, בעצם בוחרים בכל פעם את הממוצע (chance) ובשורש בוחרים את המקסימום (max).

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



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

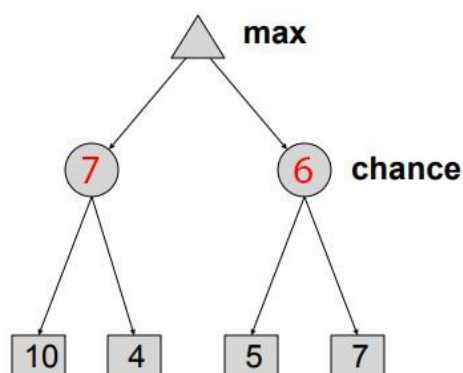
אופציה א: החלקת המסך ימינה שבעקבותיה זוג מספרים יתמזג ויתווסף אריח עם הערך 2.

אופציה ב: החלקת המסך ימינה שבעקבותיה זוג מספרים יתמזג ויתווסף אריח עם הערך 4.

- על מנת לא לסבך את הדוגמא, לא אתייחס פה לכמות המקומות הרנדומליים האופציונליים למיקום הערך 2 או לכמות המקומות הרנדומליים האופציונליים למיקום הערך 4 ונניח שיש 2 אופציות בלבד למקם כל אחד מהם.

כעת, כאשר יש לנו את הערכים (10, 4) ו(5, 7), אנחנו יכולים לחשב את ערך ההורים 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.

כעת, העץ יראה באופן הבא:

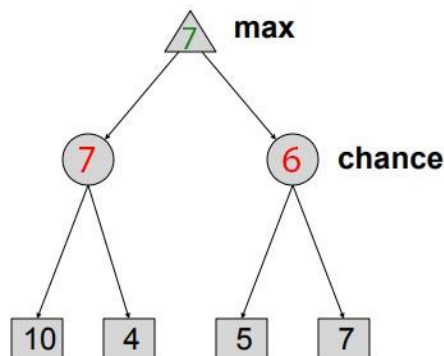


כעת, הגענו לשלב האחרון בבניית העץ, והוא בחירת הבן בעל הערך המקסימלי.

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

מהרצת האלגוריתם ExpectiMax עבור העץ שלעיל, נקבל שערכו של שורש העץ יהיה 7, כיוון שלשורש יש שני בנים, וערכם הוא 6 ו7, ו7 גדול מ6 (כאמור, בוחרים את הבן בעל הערך המקסימלי).

לכן, בסיום ריצת האלגוריתם, נקבל את העץ הבא:



אז איך ניתן ליישם את האלגוריתם ExpectiMax בפרויקט?

ראשית, צריך להבין מהן האופציות שעומדות בפנינו בכל מהלך ולהחליט איך כדאי לממש את האלגוריתם כך שבסופו של דבר, השחקן הממוחשב יגיע לביצועים מקסימליים. כפי שהסברתי מוקדם יותר, האסטרטגיה הטובה ביותר למשחק 2048 היא לשמור את האריח בעל הערך המקסימלי בצד שמאל למעלה של הלוח. על מנת לשמור על כך, בעוד שאנו עוברים על כמות אופציות גדולה מאוד, ניעזר בחישובים מתמטיים שיאפשרו לנו לשמור על האריח בעל הערך המקסימלי תמיד בצד שמאל למעלה.

איך ניתן הניקוד למהלכים במשחק?

על מנת לשמור על האריח בעל הערך הגבוה ביותר בלוח בצד שמאל למעלה של הלוח, החלטתי למשקל את האריחים ע"פ "קדימותם" בלוח וערכם. על מנת לבצע זאת, השתמשתי בנוסחה הבאה: $\log_2 x * \left(\frac{1}{4}\right)^y$, כאשר: $x = \text{Tile's Value}$, $y = \text{Tile's Index}$. האינדקס של האריח נקבע ע"פ מיקומו בלוח והוא נע בין 0 ל-15, וערכו של האריח הוא הערך שהאריח מחזיק באותו הזמן. לדוגמה, אם אריח הוא בעל הערך 8 והוא ממוקם באינדקס 0,0 (האריח בצד שמאל למעלה של הלוח), אז נתייחס אליו כאל $\log_2 8 * \left(\frac{1}{4}\right)^0$, כלומר, כאל הערך 3.

באמצעות שיטה זו, ניתן להבטיח שהאריח בעל הערך הגדול ביותר יהיה בצד שמאל למעלה **ברוב המוחלט** של המצבים, כאשר יהיו צמודים אליו (מאחד הכיוונים) שאר האריחים בעלי הערך הגבוה ביותר, כיוון שבכל תור הערך שהכי משפיע על התוצאה הסופית הוא ערך האריח באינדקס 0,0, כלומר בצד שמאל למעלה בלוח.

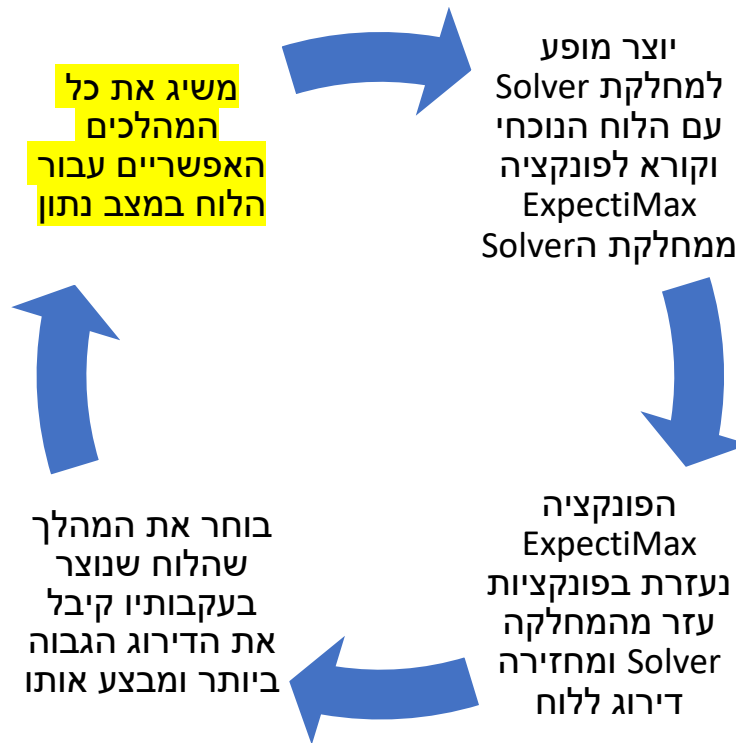
איך נקבע המהלך שצריך לבצע?

על מנת לקבוע איזה מהלך הוא הטוב ביותר ברגע נתון, צריך לדעת מה השפעת המהלך על הלוח במצב הנוכחי (לפני כניסה לעומק). במידה והמהלך לא משפיע על הלוח, המהלך מוסר מרשימת המהלכים האפשריים בעבור התור הנוכחי. במידה ובזמן ריצת האלגוריתם (עם עומק) מתגלה שהשחקן הממוחשב עשוי להפסיד, אותו מהלך יקבל את העונש הגרוע ביותר – והוא את הערך `double.MinValue`, כלומר הערך הנמוך ביותר שניתן לייצג באמצעות משתנה מאותו 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, עלינו ליצור סך של $2 * Z - 32$ לוחות כאשר Z מייצג את כמות האריחים המאותחלים בלוח. הסיבה נובעת מכך שאנו לא יודעים האם לאחר מהלך יתווסף אריח שערכו 2 או שיתווסף אריח שערכו 4, ומעבר לכך אנחנו לא יודעים באיזה מיקום יתווסף האריח. לכן, במקרה הגרוע ביותר, ניאלץ ליצור 30 לוחות (אם יש אריח 1 בלבד על הלוח לאחר ביצוע המהלך, כלומר היה מיזוג של 2 אריחים בעקבות המהלך הנוכחי), כאשר 15 מהלוחות יכילו אריח עם הערך 2 במיקום רנדומלי ו-15 מהלוחות יכילו אריח עם הערך 4 במיקום רנדומלי על הלוח. את יצירת הלוחות ניאלץ לבצע בעבור כל מהלך על מנת לקבוע את דירוגו. כמות הלוחות שיווצרו היא כמובן שונה בהתאם לעומק שייבחר בעבור האלגוריתם, כאשר האלגוריתם פועל בצורה היעילה ביותר במידת האפשר מבלי לפגוע בביצועיו. בהתאם לדיאגרמת העץ שמוסברת לעיל בנוגע לפעילות האלגוריתם ExpectiMax, כך גם האלגוריתם לשחקן ממוחשב פועל. בהתאם לעומק הנבחר, האלגוריתם יבחר את ממוצע הבנים של כל צומת ולבסוף עבור שורש העץ הוא יבחר את הבן שערכו הוא הגבוה ביותר, ויבצע את המהלך מבין 4 המהלכים שערכו הוא הגבוה ביותר.

התרשים הבא מייצג את האופן שבו הAI פועל בפרויקט:

מסומן בצהוב – נקודת פתיחה לתרשים הזרימה.



הסבר מילולי לתרשים הזרימה:

בפועל, בעבור כל פעולה שהAI מבצע, הAI ראשית מייצר עד 4 לוחות כאשר כל לוח מייצג את הלוח שיווצר בעקבות ביצוע תזוזה לכיוון כלשהוא. הלוחות אליהם הAI יתייחס הם הלוחות ששונים מהלוח הנוכחי **בלבד**, וזאת לפני שיתווסף אליהם אריח במיקום רנדומלי. לאחר מכן, בעבור כל אחד מהלוחות, הAI יקרא לפונקציה ExpectiMax שמוגדרת במחלקת Solver על מנת לדרג את כל אחד מלוחות המשחק שנוצרו, ולבסוף הAI **בוחר ומבצע** את המהלך שהלוח שנוצר בעקבותיו קיבל את הדירוג הגבוה ביותר, כיוון שמהלך זה הוא הכדאי ביותר.

תיאור אלגוריתמים:

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

האלגוריתם הוא מחלקה הנקראת SceneManager ותפקידה הוא לנהל את "הסצנות" השונות במשחק. המחלקה ממשת פונקציות מה framework שנקרא MonoGame שפותח ע"י מייקרוסופט לצורך פיתוח משחקים בסביבת .NET. והפונקציות הללו אחראיות לטעינת, הצגת וקליטת מידע.

- קיים enum הנקרא DrawState, ומטרתו היא לעזור למחלקה לקבוע איזה "סצנה" צריך לצייר ברגע נתון.
- חלק מהפונקציות נקראות באופן אוטומטי כחלק מהגדרות framework וחלק נקראות באופן אינדיבידואלי כאשר יש בהן צורך. הפונקציות Draw, Update נקראות פעם אחת ע"י הגדרת framework ולאחר מכן הן קוראות לעצמן באופן רקורסיבי לאורך ריצת המשחק.

להלן אציג את האלגוריתם לתיאום מערכת לבחירות השחקן מהמחלקה SceneManager:

1. אתחל סצנה – InitializeScene:

a. צור "ריבועים" (Rectangles) רלוונטים לסצנה.

2. טען סצנה – LoadContent:

a. טען מידע רלוונטי לסצנה הנוכחית.

3. עדכון מסך – Update:

a. אם הסצנה לציור כעת היא Menu אז:

i. אם הכפתור לבחירת משחק שחקן אנושי נלחץ אז:

1. בטל סצנה.

2. עדכן משתנה לקביעת סצנה נוכחית לסצינת שחקן אנושי.

3. עדכן ערך דגל לאמת.

ii. אם הכפתור לבחירת משחק שחקן מול מחשב נלחץ אז:

1. בטל סצנה.

2. עדכן משתנה לקביעת סצנה נוכחית לסצינת שחקן מול מחשב.

3. עדכן ערך דגל לאמת.

iii. בדוק אם הכפתור לבחירת משחק שחקן מחשב נלחץ.

1. בטל סצנה.

2. עדכן משתנה לקביעת סצנה נוכחית לסצינת שחקן ממוחשב.

3. עדכן ערך דגל לאמת.

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

1. אתחל סצנה.

2. טען סצנה.

3. נעל את העכבר.

v. אחרת:

1. בטל את נעילת העכבר.

b. אם הסצנה לציור כעת היא PlayerMode אז:

- i. בדוק כמה delay נותר עד שהמשתמש יכול לבצע את התור הבא.
- ii. אם הכפתור לתחילת משחק מחדש נלחץ אז:
 1. אם העכבר לא נעול אז:
 - a. צור לוח משחק חדש.
 - b. נעל את העכבר.
 - iii. אם הכפתור לחזרה נלחץ אז:
 1. אם העכבר לא נעול אז:
 - a. בטל סצינה.
 - b. עדכן משתנה לקביעת סצינה נוכחית לסצינת Menu.
 - c. אתחל סצינה.
 - d. טען סצינה.
 - e. נעל את העכבר.
 - iv. אחרת:
 1. בטל את נעילת העכבר.
 - c. אם הסצינה לציור כעת היא ChooseDifficulty אז:
 - i. אם הכפתור למצב קל נבחר אז:
 1. תן למשתנה depth את הערך 0.
 2. עדכן ערך דגל לאמת.
 - ii. אם הכפתור למצב קשה נבחר אז:
 1. תן למשתנה depth את הערך 2.
 2. עדכן ערך דגל לאמת.
 - iii. אם הכפתור למצב מקצוען נבחר אז:
 1. תן למשתנה depth את הערך 4.
 2. עדכן ערך דגל לאמת.
 - iv. אם אחד הכפתורים נלחץ (דגל = אמת) אז:
 1. בטל סצינה.
 2. צור לוח משחק חדש (עבור השחקן האנושי).
 3. צור לוח משחק חדש עם AIState.Exi depth (עבור השחקן הממוחשב).
 4. עדכן משתנה לקביעת סצינה נוכחית לסצינת PlayVsAi.
 5. אתחל סצינה.
 6. טען סצינה.
 7. נעל את העכבר.
 - v. אחרת:
 - vi. בטל את נעילת העכבר.
 - d. אם הסצינה לציור כעת היא PlayVsAi אז:
 - i. אם המצב לציור ע"פ לוח המשחק של המחשב הוא הפסד אז:
 1. בדוק כמה delay נותר עד שהמשתמש יכול לבצע את התור הבא.
 2. אם הכפתור לתחילת משחק מחדש נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - b. צור לוח משחק חדש (עבור השחקן).

- c. צור לוח משחק חדש עם AIStateI depth זהים לאלו שיש ללוח משחק של המחשב כרגע.
3. אם הכפתור לחזרה נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - i. בטל סצינה.
 - ii. עדכן משתנה לקביעת סצינה נוכחית לסצינת Menu.
 - iii. אתחל סצינה.
 - iv. טען סצינה.
 - v. נעל את העכבר.
 - b. אחרת:
 - i. בטל את נעילת העכבר.
 - ii. אחרת:
1. אם עברו 0.02 שניות מאז התור הקודם שבוצע ע"י ai אז:
 - a. אם ai של לוח שחקן המחשב הוא Randi אז:
 - i. קרא לפונקציה RandomAI() של לוח שחקן המחשב.
 - b. אם ai של לוח שחקן המחשב הוא Ex אז:
 - i. קרא לפונקציה ExpectiMaxAI() של לוח שחקן המחשב.
2. בדוק כמה delay נותר עד שהמחשב יכול להציג את האנימציות תזוזה של האריחים.
3. אם הכפתור לתחילת משחק מחדש נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - b. צור לוח משחק חדש (עבור השחקן).
 - c. צור לוח משחק חדש עם AIStateI depth זהים לאלו שיש ללוח משחק של המחשב כרגע.
4. אם הכפתור לחזרה נלחץ אז:
 - a. אם העכבר לא נעול אז:
 - i. בטל סצינה.
 - ii. עדכן משתנה לקביעת סצינה נוכחית לסצינת Menu.
 - iii. אתחל סצינה.
 - iv. טען סצינה.
 - v. נעל את העכבר.
 - b. אחרת:
 - i. בטל את נעילת העכבר.
 - e. אם הסצינה לציור כרגע היא ChooseAi אז:
 - i. אם הכפתור לבחירת Randi נלחץ אז:
 1. בטל סצינה.
 2. עדכן משתנה לקביעת סצינה נוכחית לסצינת WatchAi.
 3. צור לוח משחק חדש עם ai Randi.

4. עדכן ערך דגל לאמת.
- ii. אם הכפתור לבחירת Ex נלחץ אז:
 1. בטל סצינה.
 2. עדכן משתנה לקביעת סצינה נוכחית לסצינת WatchAi.
 3. צור לוח משחק חדש עם ה-Ex Ai ועומק 4.
 4. עדכן ערך דגל לאמת.
- iii. אם אחד הכפתורים נלחץ (דגל = אמת) אז:
 1. אתחל סצינה.
 2. טען סצינה.
 3. נעל את העכבר.
- iv. אחרת:
 1. בטל את נעילת העכבר.
- f. אם עברו 0.02 שניות מאז התור הקודם שבוצע ע"י ai אז:
 - i. אם ai של לוח שחקן המחשב הוא Randi אז:
 1. קרא לפונקציה RandomAi() של לוח שחקן המחשב.
 - ii. אם ai של לוח שחקן המחשב הוא Ex אז:
 1. קרא לפונקציה ExpectiMaxAi() של לוח שחקן המחשב.
 - iii. בדוק כמה delay נותר עד שהמחשב יכול להציג את האנימציות תזוזה של האריחים.
 - iv. אם הכפתור לתחילת משחק מחדש נלחץ אז:
 1. אם העכבר לא נעול אז:
 - a. צור לוח משחק חדש עם אותו AIState וdepth זהים לאלו שיש ללוח המשחק כרגע.
 - b. נעל את העכבר.
 - v. אם הכפתור לחזרה נלחץ אז:
 1. אם העכבר לא נעול אז:
 - a. בטל סצינה.
 - b. עדכן משתנה לקביעת סצינה נוכחית לסצינת Menu.
 - c. אתחל סצינה.
 - d. טען סצינה.
 - e. נעל את העכבר.
 - vi. אחרת:
 1. בטל את נעילת העכבר.

4. צייר סצינה – Draw:

- a. אם הסצינה לציור כרגע היא Menu אז:
 - i. צייר את כל הtextures והrectangles הרלוונטיים לסצינה.
- b. אם הסצינה לציור כרגע היא PlayerMode אז:
 - i. צייר את כל הtextures והrectangles הרלוונטיים לסצינה.
 - ii. אם מצב לוח המשחק מורה על ניצחון אז:
 1. צייר את הtexutre והrectangle הרלוונטיים למצב ניצחון.
 - iii. אם מצב לוח המשחק מורה על הפסד אז:

1. צייר את texturen והrectangles הרלוונטיים למצב הפסד.
 - c. אם הסצינה לציור כרגע היא ChooseDifficulty אז:
 - i. צייר את הרקע לסצינה (שמכיל בתוכו את כל מראה הסצינה).
 - d. אם הסצינה לציור כרגע היא PlayVsAi אז:
 - i. צייר את כל textures והrectangles הרלוונטיים לסצינה.
 - ii. אם מצב הלוח של השחקן הממוחשב שונה מהפסד אז:
 1. צייר את Tiles של לוח השחקן הממוחשב.
 - iii. אחרת:
 1. צייר את Tiles של לוח השחקן האנושי.
 - iv. אם מצב לוח השחקן הוא הפסד אז:
 1. אם ניקוד השחקן האנושי < ניקוד השחקן הממוחשב אז:
 - a. צייר את texturen והrectangles הרלוונטיים למצב ניצחון שחקן על שחקן ממוחשב.
 2. אחרת:
 - a. צייר את texturen והrectangles הרלוונטיים למצב הפסד שחקן מול שחקן ממוחשב.
 - e. אם הסצינה כרגע היא ChooseAi אז:
 - i. צייר את הרקע לסצינה (שמכיל בתוכו את כל מראה הסצינה).
 - f. אם הסצינה כרגע היא WatchAi אז:
 - i. צייר את כל textures והrectangles הרלוונטיים לסצינה.
 - ii. אם מצב הלוח הוא הפסד אז:
 1. צייר את texturen והrectangles הרלוונטיים למצב הפסד של שחקן ממוחשב.
5. בטל סצינה – UnloadScene:
- a. עדכן את רשימת texturesn לרשימה חדשה וריקה.

תיאור אלגוריתם: בדיקת תקינות מהלך + התמזגות אריחים | $O(n)$

- B מתייחס לחלק התמזגות האריחים, בעוד ששני החלקים ביחד מתייחסים לחלק בדיקת תקינות המהלך (אם בסיום הפונקציה ערך המשתנה boardChanged הוא אמת אז המהלך הוא תקין, אחרת המהלך לא תקין).
- 1. עבור i כאשר $i=0...4$ בצע:
 - 1.1. בצע כל עוד (do-while) ערך המשתנה change הוא אמת:
 - a. עבור j כאשר $j=1...4$ בצע:
 - b. אם ערך התא במיקום $i, j-1$ מהטבלה שונה מ-1, וגם ערך התא במיקום i, j מהטבלה שונה מ-1, וגם האריח באינדקס שהוא הערך של התא במיקום $i, j-1$, מהטבלה ברשימת האריחים שווה ערך לאריח באינדקס שהוא הערך של התא במיקום i, j מהטבלה { $TilesList[table[i, j]].Equals(TilesList[i, j-1])$ }, אז:
 - i. הוסף את האריח לרשימת האריחים שהתמזגו.
 - ii. הצב עבור האריח הראשון (i, j) ברשימת האריחים את המצב "שדרג".

- III. הצב עבור האריח השני ($i, j-1$) ברשימת האריחים את המצב "הרוס".
- IV. הצב עבור התא באינדקס $i, j-1$ בטבלה את הערך 1-.
- V. הצב למשתנה change את הערך true.
- VI. עצור לולאה.
- c. אם ערך התא במיקום ה- $i, j-1$ מהטבלה שווה ל-1 וערך התא במיקום j, i שונה מ-1, אז:
 - I. עדכן את הערך בטבלה באינדקס $i, j-1$ לערך שנמצא באינדקס j, i .
 - II. עדכן את הערך בטבלה באינדקס j, i 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 שהתמזגו}.

תיאור אלגוריתם: אלגוריתם לשחקן ממוחשב | $O(n)$

1. אם המקלדת לא נעולה וגם מצב הציוור במשחק הוא לא הפסד אז:
 - a. אתחל את טבלת לוח המשחק באמצעות קריאה לפונקציה ListToMatrixAI().
 - b. אתחל את המשתנה bestScore עם המינימלי ביותר שניתן לאחסן בו, ואת המשתנה moveRating עם 0.
 - c. אתחל את מילון המצבים הטובים ביותר כמילון ריק.
 - d. אתחל את מילון המצבים האפשריים באמצעות קריאה לפונקציה getAllMoveStates() של ה-AI.
 - e. בעבור כל זוג מפתח-ערך move מהמילון moves:
 - i. הצב במשתנה moveRating את הערך שמתקבל מקריאה לאלגוריתם ExpectiMax מהמחלקה של ה-AI.
 - ii. אם moveRating גדול מ bestScore אז:
 1. עדכן את bestScore להיות שווה ערך ל moveRating.
 2. מחק את הערכים במילון movesBest.
 - iii. אם moveRating שווה ל bestScore אז:
 1. הוסף לרשימת movesBest את move.
 - f. אם כמות הערכים במovesBest גדולה מ 0 אז:
 - i. קרא לפונקציה לביצוע פעולה עם המפתח הראשון במovesBest.
 - g. נעל את המקלדת.

מבני נתונים:

רשימה (List):

מבנה הנתונים העיקרי בפרויקט הוא רשימה. מבנה זה נלקח מהספרייה `System.Collections.Generic` של .NET. והוא מאפשר לשמור ערכים באופן דינאמי ומסודר. אחד השימושים המרכזיים שהיה לי במבנה נתונים זה הוא ייצוג לוח המשחק. את לוח המשחק ייצגתי באמצעות רשימת אריחים (`List<Tile>`) אשר איפשרה לי לשמור רק את האריחים שמתבצע בהם שימוש בלוח באותו הזמן, כאשר כל אריח ברשימה הכיל תכונות רבות, וביניהן גם 2 נקודות (Points) שנלקחו מהספרייה `Microsoft.Xna.framework` וייצגו את המיקום של כל אריח בלוח לפני ואחרי מהלך, כאשר מתייחסים ללוח כאל מטריצה דו מימדית בגודל 4x4. שימוש מרכזי נוסף שביצעתי עם מבנה נתונים זה היה בחלק של הבינה המלאכותית, כאשר הייתי צריך לחשב את השפעת מהלך כלשהו על המשחק. באמצעות רשימה של מצבים, ייצגתי את הלוחות שעשויים להיווצר בעקבות מהלך מסויים, ולאחר מכן נעזרתי בכך על מנת לקבוע איזה מהלך הוא הטוב ביותר לביצוע ע"פ האלגוריתם ExpectiMax.

מטריצה (Matrix):

בפרויקט, ובמיוחד בחלק הבינה המלאכותית בפרויקט, נעזרתי במטריצה, שהיא בעצם מערך דו מימדי ריבועי, כלומר כמות השורות והטורים בו זהה, כיוון שהיה נוח יותר לייצג ולעבוד עם לוחק המשחק באופן זה, כאשר מתייחסים לכל אריח (גם אם ערכו אינו מאותחל) בחישובים לקביעת המהלך הטוב ביותר עבור מצב נתון.

מילון (Dictionary):

בפרויקט, ובמיוחד בחלק הבינה המלאכותית בפרויקט, נעזרתי במילון לצורך שיוך בין המהלך שמבצעים לבין הלוח שמתקבל בעקבות ביצוע המהלך. מילון הוא מבנה נתונים שמוגדר בספרייה `System.Collections.Generic` של .NET. ומטרתו היא לאפשר שמירה של מפתחות לצד ערכים. אחד השימושים המרכזיים במבנה נתונים זה בפרויקט בא לידי ביטוי בחלק הבינה המלאכותית בו האלגוריתם צריך לדרג כל מהלך, ולצורך ביצוע החישובים המתאימים שיובילו לתוצאות מתאימות, השתמשתי במילון על מנת לקשר בין המפתחות – המהלך שמתבצע (ימינה, שמאלה, מעלה, מטה), לבין הערכים – הלוחות שמתקבלים בעקבות ביצוע המהלך.

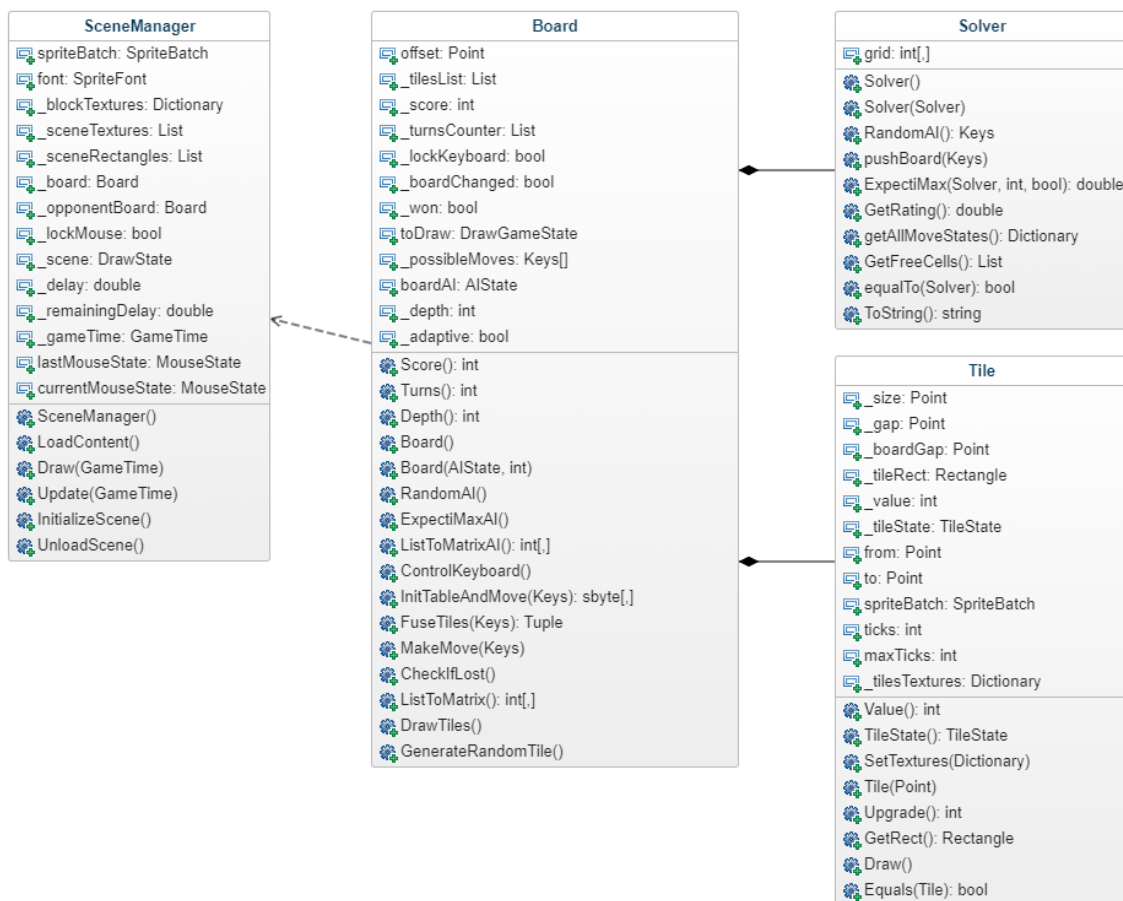
מערך (Array):

בפרויקט, השתמשתי במערך לצורך אחסון וייצוג המהלכים המותרים במשחק (ימינה, שמאלה, מעלה, מטה).

סט ממוין (Sorted Set):

סט ממוין הינו מבנה נתונים המאפשר לנו לאחסן ערכים בעודם **ממויינים**. כלומר, אם ניקח סט ממוין שיכיל את הערכים בתחום 0-15, האיבר הראשון בסט יהיה 0, השני יהיה 1, וכך הלאה עד האיבר האחרון שערכו יהיה 15. התכונה החשובה ביותר של מבנה נתונים זה היא שלא משנה איזו פעולה נבצע, הוא תמיד ישאר ממוין. כלומר, אנו יכולים להוסיף או להסיר מספר מהסט מבלי לחשוב על ההשפעות של הפעולה שביצענו על סידור הערכים, כיוון שהם תמיד ישארו ממויינים. אני מבצע שימוש במבנה נתונים זה בפרויקט על מנת לאחסן את אינדקסי האריחים הפנויים בלוח **ביעילות** רבה בזכות תכונה זו.

תרשים מחלקות – UML Class Diagram:



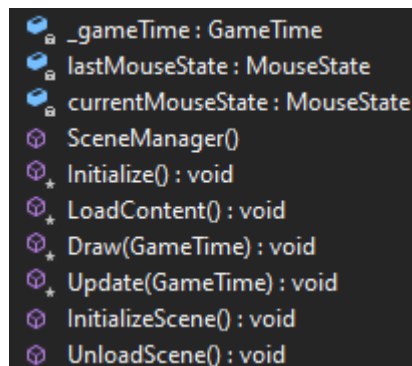
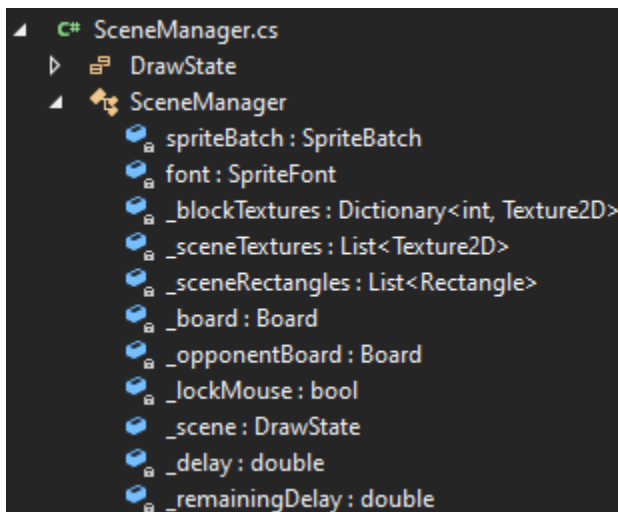
הסבר כללי על המחלקות:

- המחלקה **SceneManager** היא המחלקה שאחראית על ה GUI של המשחק והיא מבצעת זאת באמצעות שימוש בפונקציות של *MonoGame*.
- המחלקה **Board** היא המחלקה שאחראית על לוח המשחק ועל כל הלוגיקה שפועלת מאחוריו.
- המחלקה **Tile** מייצגת אריח בלוח המשחק, ומכילה מידע אשר עוזר לנו למקם את האריחים על גבי לוח המשחק ולעבוד עם כל אריח בצורה יעילה יותר.
- המחלקה **Solver** אחראית על פתירת לוח המשחק ומכילה 2 אלגוריתמים לפתירת לוח בתוכה.

תכנון:

מחלקת SceneManager:

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



טבלת נתונים:

| שם הנתון | פירוט הנתון |
|--|--|
| <code>private SpriteBatch spriteBatch</code> | אחראי על ציור המשחק. |
| <code>private SpriteFont font</code> | אחראי לאחסון font בשימוש בעת ציור המשחק. |
| <code>private Dictionary<int, Texture2D> _blockTextures</code> | מילון שיכיל ערכים לצד משתנים מסוג Texture2D שיכילו תמונות שייצגו את האריח שערכו הוא המפתח. |
| <code>private List<Texture2D> _sceneTextures</code> | רשימה שמכילה Texture2D לצורך אחסון התמונות שנציג במשחק בכל זמן נתון. |
| <code>private List<Rectangle> _sceneRectangles</code> | רשימה שמכילה Rectangle לצורך נתינת "גוף פיזי" שבאמצעותו נוכל לזהות לחיצות על כפתורים ועוד. |
| <code>private Board _board</code> | מופע לוח משחק אשר בעזרתו נממש לוח משחק עבור שחקן אנושי/ממוחשב. |
| <code>private Board _opponentBoard</code> | מופע לוח משחק אשר בעזרתו נממש לוח משחק עבור שחקן ממוחשב. |
| <code>private bool _lockMouse</code> | משתנה שמאפשר לנו להגביל את מספר הלחיצות על העכבר שהמשתמש יכול לבצע בבת אחת. |
| <code>public static DrawState _scene</code> | מופע של הenum DrawState אשר מאפשר לזהות איזו סצינה צריך לצייר בכל זמן. |

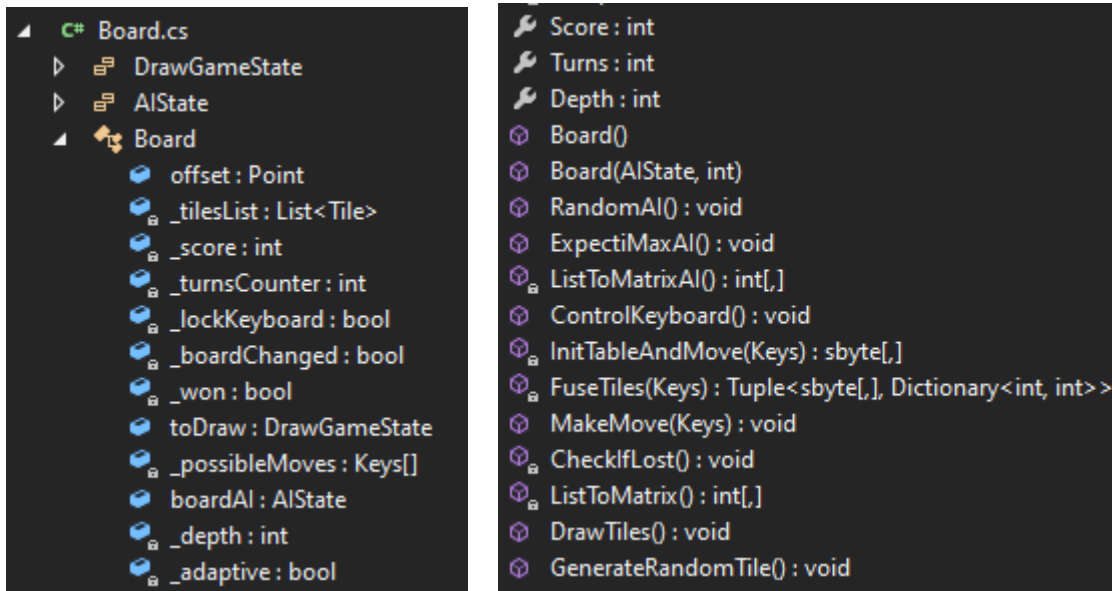
| | |
|---|---|
| קבוע שמטרתו היא לקבוע דילאיי בין מהלכים שהמשתמש מבצע. | <code>private const double _delay</code> |
| משתנה עזר למעקב אחר כמות הזמן שנותרה עד שהמשתמש יכול לבצע את המהלך הבא שלו. | <code>private double _remainingDelay</code> |
| משתנה עזר למעקב אחר כמות הזמן בין הפעולות שהמחשב. | <code>private 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 פעולות:

| שם הפעולה | תיעוד הפעולה |
|--|---|
| <code>public SceneManager()</code> | בנאי המחלקה, יוצר עצם מטיפוס SceneManager שינהל את הסצנות שמוצגות למשתמש בכל עת. |
| <code>protected override void Initialize()</code> | מאפשר למשחק לבצע אתחול כלשהוא אשר נדרש לפני ריצת התוכנית. זוהי פונקציה מובנית כחלק מהframework שנקרא MonoGame. |
| <code>protected override void LoadContent()</code> | טוען את התוכן הרלוונטי מהRootDirectory של התוכנית (אשר ערכו מאותחל בבנאי) בהתאם לסצינה הנוכחית. זוהי פונקציה מובנית כחלק מהframework שנקרא MonoGame. |
| <code>protected override void Draw(gameTime gameTime)</code> | נקראת כאשר המשחק צריך לצייר את עצמו, ותפקידה הוא לצייר את הסצנות השונות במשחק בהתאם למצב. זוהי פונקציה מובנית כחלק מהframework שנקרא MonoGame. |
| <code>protected override void Update(gameTime gameTime)</code> | פונקציה זו נקראת בכל פעם שframe של המשחק עושה render ומוצג על המסך. בפונקציה זו קיימת כל הלוגיקה בסצינה כגון – זיהוי לחיצות משתמש על כפתורים, זיהוי מצבי ניצחון/הפסד של שחקן/AI, ועוד. זוהי פונקציה מובנית כחלק מהframework שנקרא MonoGame. |
| <code>public void InitializeScene()</code> | פונקציה שמטרתה היא לאתחל את הסצנות השונות בהתבסס על הסצינה הנוכחית. |
| <code>public void UnloadScene()</code> | פונקציה שמטרתה היא לאפס את הסצינה הנוכחית על מנת שניתן יהיה לטעון סצינה חדשה. |

מחלקת Board:

מחלקה זו מייצגת את לוח המשחק.



טבלת נתונים:

| שם הנתון | פירוט הנתון |
|--|---|
| <code>public static Point offset</code> | האינדקסים שמהם לוח המשחק מתחיל (על מנת שנוכל לצייר את האריחים). |
| <code>private List<Tile> _tilesList</code> | רשימת האריחים שנמצאים על הלוח בכל זמן נתון. |
| <code>private int _score</code> | מחזיק את תוצאת המשחק בכל זמן נתון. |
| <code>private int _turnsCounter</code> | מחזיק את כמות המהלכים שבוצעו במשחק בכל זמן נתון. |
| <code>private bool _lockKeyboard</code> | מאפשר לנו לנהל את השימוש במקלדת בזמן המשחק. |
| <code>private bool _boardChanged</code> | מאפשר לנו לזהות האם לוח המשחק השתנה בעקבות ביצוע פעולה או לא. |
| <code>private bool _won</code> | מאפשר לנו לזהות האם השחקן ניצח את המשחק או לא. |
| <code>public DrawGameState toDraw</code> | מאפשר לנו לזהות את מצב הלוח שצריך לצייר בכל זמן נתון. |
| <code>Keys[] _possibleMoves</code> | מערך שמכיל את הפעולות שניתן לבצע במשחק (מעלה, מטה, ימינה ושמאלה). |
| <code>public AIState boardAI</code> | מאפשר לנו לקבוע איזה סוג AI צריך להפעיל על לוח המשחק. |
| <code>private int _depth</code> | מאפשר לנו לקבוע את עומק הAI שיהיה בשימוש במשחק. |
| <code>private bool _adaptive</code> | משתנה עזר לקביעה האם הAI צריך לפעול בצורה אדפטיבית או לא. |

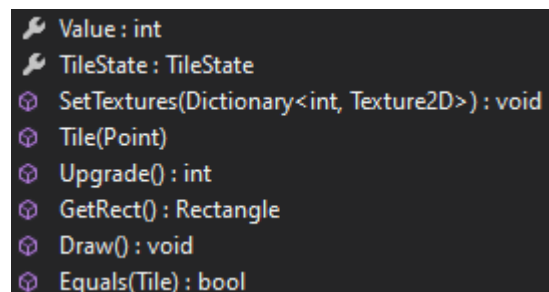
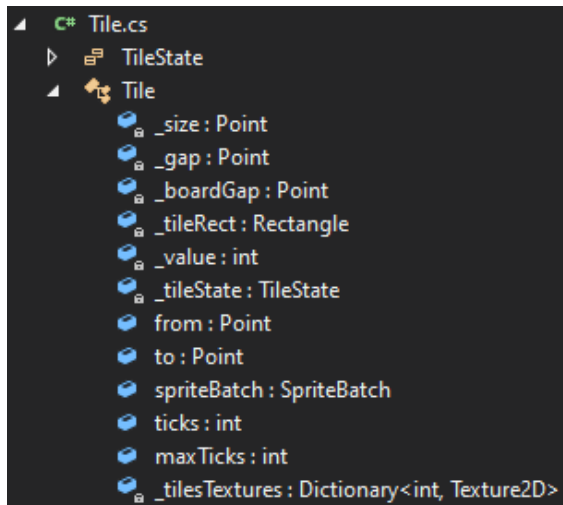
טבלת פעולות:

| שם הפעולה | תיעוד הפעולה |
|--|---|
| <code>public Board()</code> | בנאי המחלקה, יוצר עצם מטיפוס Board ומזמן את הפונקציה להוספת אריח באופן רנדומלי פעמיים . |
| <code>public Board(AIState state, int depth)</code> | בנאי המחלקה, יוצר עצם מטיפוס Board, מאתחל את תכונותיו הרלוונטיות בהתאם לערכים המתקבלים ומזמן את הפונקציה להוספת אריח באופן רנדומלי פעמיים . |
| <code>public void RandomAI()</code> | אחראית על קישור בין בחירה רנדומלית של מהלכים ללוח המשחק אשר מוגדרת במחלקת הAI ובכך מאפשרת לנו ליצור שחקן ממוחשב שבוחר מהלכים באופן רנדומלי. |
| <code>public void ExpectiMaxAI()</code> | אחראית על קישור בין בינה מלאכותית שפועל על פי האלגוריתם ExpectiMax שמוגדר במחלקת הAI לבין לוח המשחק ובכך מאפשרת לאלגוריתם לממש שחקן ממוחשב. |
| <code>private int[,] ListToMatrixAI()</code> | ממירה בין רשימת האריחים למטריצה שמאותחלת בערכי האריחים ע"פ הרשימה כאשר הערך של כל אריח מהרשימה יוכנס כlog2 שלו (לדוגמא, עבור 8 יוכנס 3). המטריצה היא בגודל 4x4 כאשר התאים שלה יאותחלו בהתאם לערכי האריחים (על פי תכונות האריחים שמעידים על האינדקס שלהם במטריצה). |
| <code>public void ControlKeyboard()</code> | מאפשרת ללוח המשחק להיות מושפע מלחיצות מקשי המקלדת. פונקציה זו תזהה לחיצות על מקשי מקלדת ותפעל בהתאם אליהם. |
| <code>private sbyte[,] InitTableAndMove(Keys key)</code> | מאתחל מטריצה של בתים מסומנים אשר תעזור לנו לזהות אלו אריחים הושפעו כתוצאה מביצוע תזוזה. |
| <code>private Tuple<sbyte[,], Dictionary<int, int>> FuseTiles(sbyte[,] table)</code> | אחראית על זיהוי האם הלוח מושפע/לא מושפע כתוצאה ממהלך של משתמש ועל בדיקה אילו אריחים התמזגו כתוצאה מהתזוזה במידה ויש כאלו. |
| <code>public void MakeMove(Keys key)</code> | הפעולה המרכזית שאחראית על תזוזת המשתמש. פעולה זו נעזרת בשתי הפעולות InitTable, FuseTiles ומבצעת את תזוזת האריחים. |
| <code>private void CheckIfLost()</code> | בודקת האם בזמן שהיא נקראה למשתמש לא נותרו מהלכים לבצע, מה שמעיד על כך שהוא הפסיד, או לא. במידה והמשתמש הפסיד, הפונקציה תשנה את ערך המשתנה |

| | |
|---|---|
| שמיצג מה צריך לצייר כרגע בלוח למצב הפסד. | |
| ממירה בין רשימת האריחים למטריצה שמאותחלת בערכי האריחים ע"פ הרשימה. המטריצה היא בגודל 4x4 כאשר התאים שלה יאותחלו בהתאם לערכי האריחים (על פי תכונות האריחים שמעידים על האינדקס שלהם במטריצה). | <code>private int[,] ListToMatrix()</code> |
| אחראית על ציור האריחים בלוח המשחק בכל רגע. פונקציה זו תזהה אילו אריחים צריך "לשדרג" ואילו אריחים צריך להזיז או להשאיר כפי שהם, ובהתאם למצבם תפעל. | <code>public void DrawTiles()</code> |
| אחראית על זימון אריח חדש ללוח המשחק. פונקציה זו בוחרת באופן רנדומלי את ערך האריח כאשר יש 90% שערך יהיה 2 ו-10% שערך יהיה 4, ואז בוחרת באופן רנדומלי מיקום על לוח המשחק שאליו יתווסף האריח החדש. | <code>public void GenerateRandomTile()</code> |

מחלקת Tile:

מחלקה זו מייצגת אריח בלוח המשחק.



טבלת נתונים:

| שם הנתון | פירוט הנתון |
|--------------------------------------|-------------------------------------|
| <code>private Point _size</code> | גודל האריח שמוצג למשתמש בזמן המשחק. |
| <code>private Point _gap</code> | המרחק בין אריחים צמודים בלוח המשחק. |
| <code>private Point _boardGap</code> | גודל השוליים של לוח המשחק. |

| | |
|--|--|
| private Rectangle _tileRect | משתנה לייצוג "הגוף הפיזי" של האריח. באמצעות משתנה זה נוכל לייצג את האריחים השונים של הלוח במקומות שונים. |
| private int _value | משתנה לאחסון ערך האריח. |
| private TileState _tileState | משתנה לייצוג מצב האריח ("הריסה"/"שדרוג"/"רגיל"). |
| public Point from | משתנה לייצוג אינדקסי האריח בלוח לפני ביצוע המהלך הנוכחי. |
| public Point to | משתנה לייצוג אינדקסי האריח בלוח אחרי ביצוע המהלך הנוכחי. |
| public static SpriteBatch spriteBatch | משתנה שמטרתו היא לקשר בין spriteBatch מה SceneManager לבין מחלקת האריח, כך שנוכל לצייר את האריחים השונים דרך מחלקתם. |
| public static int ticks | מאפשר לנו לעקוב אחר כמות ההזזות שביצענו עבור כל אריח בעקבות מהלך מסוים. |
| public static int maxTicks | מאפשר לנו לקבוע כמה הזזות אנחנו מעוניינים לבצע לאריחים עבור כל מהלך שמתבצע. |
| private static Dictionary<int, Texture2D> _tilesTextures | מאפשר לנו לשמור ולאחר מכן להציג את התמונות שמייצגות כל אריח בהתאם לערכו. |

טבלת פעולות:

| שם הפעולה | תיעוד הפעולה |
|---|---|
| public static void SetTextures(Dictionary<int, Texture2D> dict) | משווה את מילון ה_tilesTextures למילון המתקבל. |
| public Tile(Point point) | בנאי המחלקה, יוצר עצם מטיפוס Tile ושם עבורו את ערכי הX והY של הנקודה שהתקבלה כערכי הfrom והto שלו. |
| public int Upgrade() | פעולה זו מעדכנת את ערך האריח הנוכחי לפי 2 מערכו, מעדכנת את מצבו ל"רגיל" ומחזירה את הערך החדש של האריח. |
| public Rectangle GetRect() | פעולה זו מחזירה את ה"ריבוע" שעל פי אינדקסי הX והY שלו לאחר מכן נצייר את האריח הנוכחי על המסך. |
| public void Draw() | פעולה זו אחראית לצייר כל אריח ע"פ ה"ריבוע" שמתאים לו, אשר מופק ע"י הפעולה GetRect(). |
| public bool Equals(Tile other) | מחזירה אמת אם האריח הנוכחי שווה לאריח אחר. אריח נחשב לזהה לאריח אחר אם ערכו ומצבו זהים לאלו של האחר ומצבו "רגיל". |

מחלקת Solver:

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

```
C# Solver.cs
Solver
  grid : int[,]
  Solver()
  Solver(Solver)
  RandomAI() : Keys
  pushBoard(Keys) : void
  ExpectiMax(Solver, int, bool) : double
  GetRating() : double
  getAllMoveStates() : Dictionary<Keys, Solver>
  GetFreeCells() : List<Point>
  getAllRandom() : List<Solver>
  equalTo(Solver) : bool
  ToString() : string
```

טבלת נתונים:

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

טבלת פעולות:

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

| | |
|--|---|
| ובשורש את הבן שערכו הוא הגבוה ביותר.
עבור כל צומת "בעץ" שנוצר, הפעולה מחשבת את הלוח שיווצר, ובסיום מחזירה את דירוג המהלך האופטימלי לביצוע. | |
| פעולה זו מחשבת את הניקוד שמהלך מסוים צריך לקבל. פעולה זו מחזירה את הניקוד הגבוה מבין הניקודים ששתי הרשימות ScanGridm קיבלו. | <code>public double GetRating()</code> |
| מחזירה מילון שכולל את התזוזה (מפתח) לצד מצב הלוח (ערך) בעקבות ביצוע כל אחת מהתזוזות האפשריות, כאשר במילון יהיו רק הפעולות שביצען ישפיע על לוח המשחק. | <code>public Dictionary<Keys, Solver> getAllMoveStates()</code> |
| מחזירה רשימה של נקודות שמכילה את האינדקסים של כל התאים בgrid שערכם אינו מאותחל. | <code>private List<Point> GetFreeCells()</code> |
| מחזירה רשימה של מצבים שמייצגים את כל הלוחות שעשויים להיווצר בעקבות התווספות אריח חדש ללוח המשחק. | <code>public List<Solver> getAllRandom()</code> |
| מחזירה אמת אם grid של מופע המחלקה שעליו הפונקציה נקראת זהה לgrid של מופע המחלקה שהפונקציה מקבלת. | <code>public bool equalTo(Solver alt)</code> |
| מאפשרת לנו לראות את לוח המשחק בצורה פשוטה וברורה יותר בזמן debugging. | <code>public override string ToString()</code> |

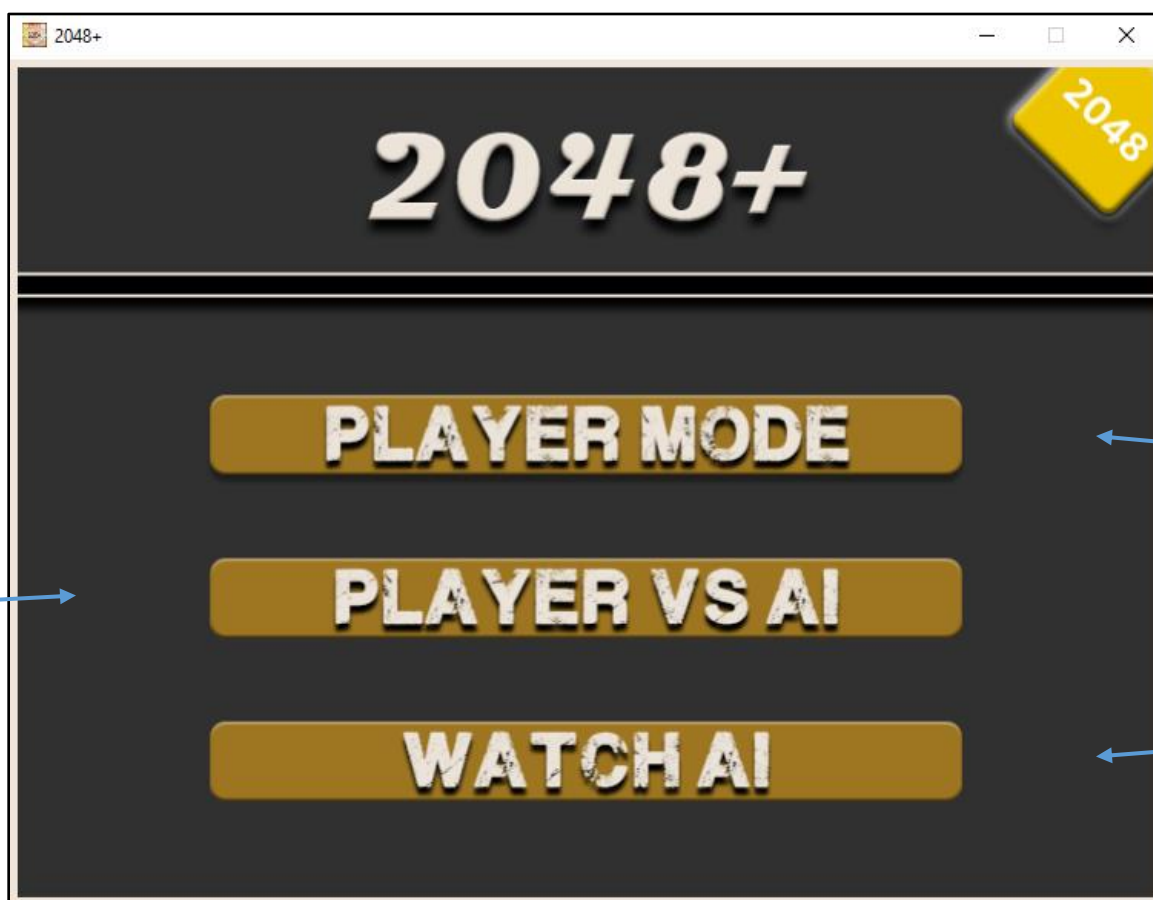
מדריך למשתמש

כעת אסביר בקצרה כיצד לתפעל את המערכת כראוי.

מסך הפתיחה:

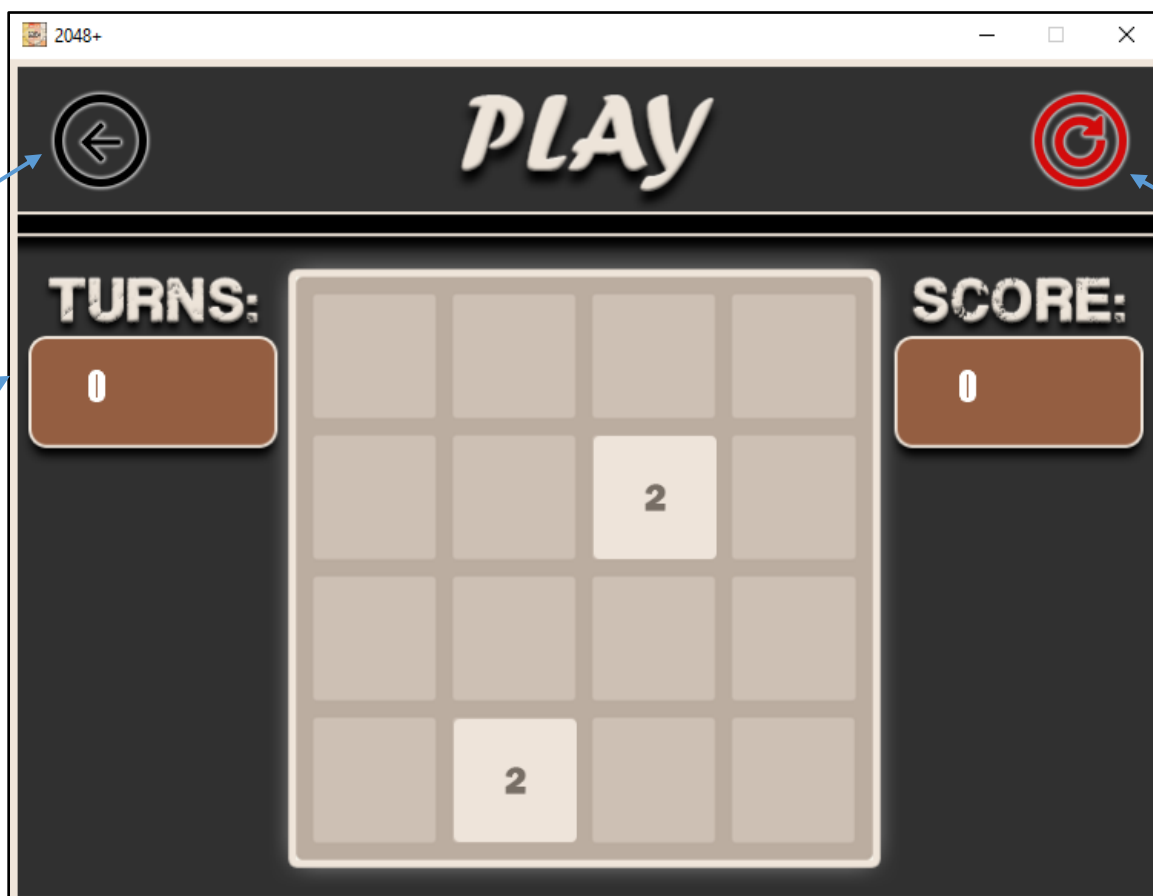
במסך זה המשתמש צריך לבחור את האופן שבו הוא מעוניין לשחק את המשחק. את המשחק ניתן לשחק בשלושת האופנים הבאים:

1. משחק שחקן אנושי (בגרסתו הרגילה של 2048).
2. שחקן אנושי מול שחקן ממוחשב (מתחרים להגיע לתוצאה הגבוהה ביותר).
3. משחק שחקן ממוחשב (השחקן הממוחשב מנסה להגיע לתוצאה הגבוהה ביותר).



מסך משחק שחקן אנושי:

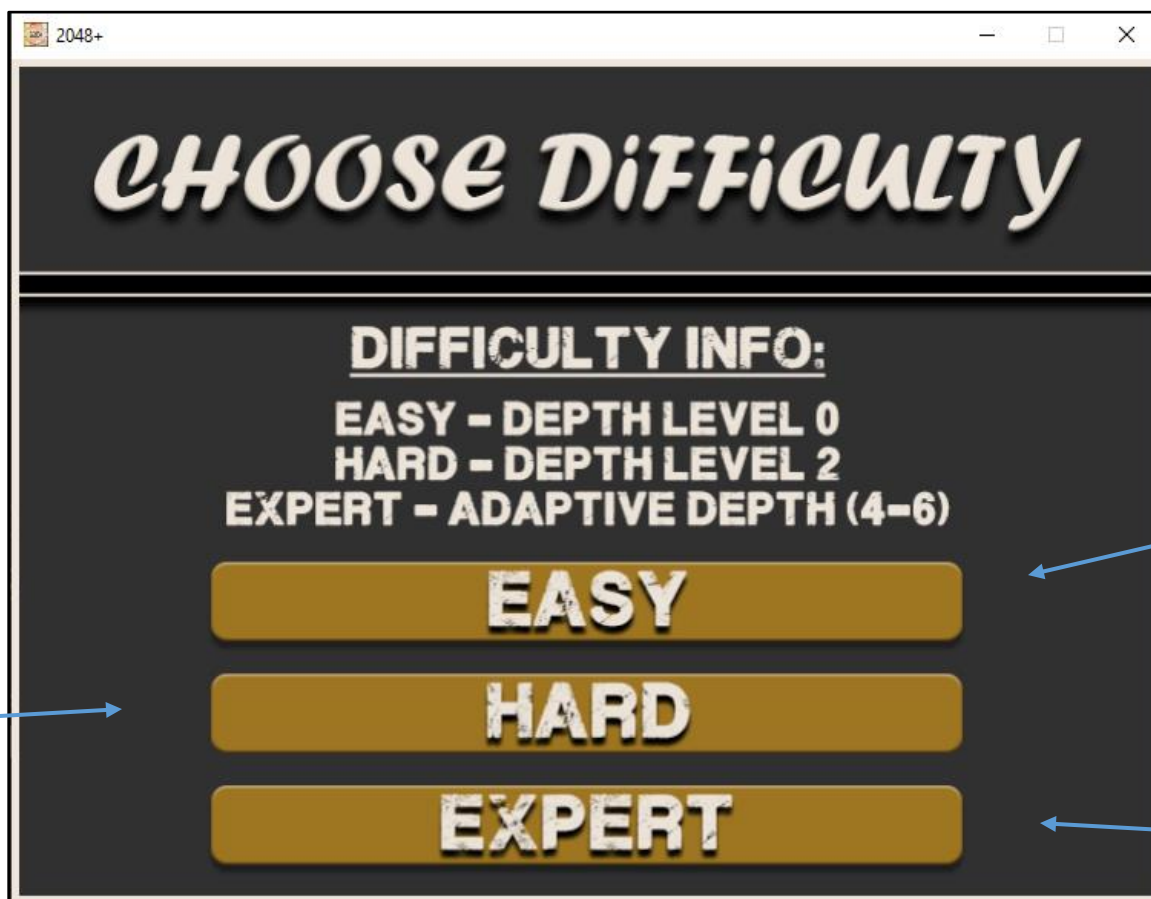
כאשר השחקן נמצא במסך זה, הוא יכול לשחק במשחק 2048 בגרסתו הרגילה, כאשר מטרתו של השחקן היא להגיע לאריח בעל הערך 2048 על מנת לנצח.



מסך בחירת 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 מהי 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 שבסופו של דבר הסיכויים שלו להיות טוב יותר יהיו גבוהים יותר.



כפתור
לבחירת מצב
קל – עומק 0

כפתור
לבחירת מצב
קשה – עומק 2

כפתור
לבחירת מצב
מקצוען –
עומק 4

מסך משחק שחקן אנושי מול שחקן ממוחשב:

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

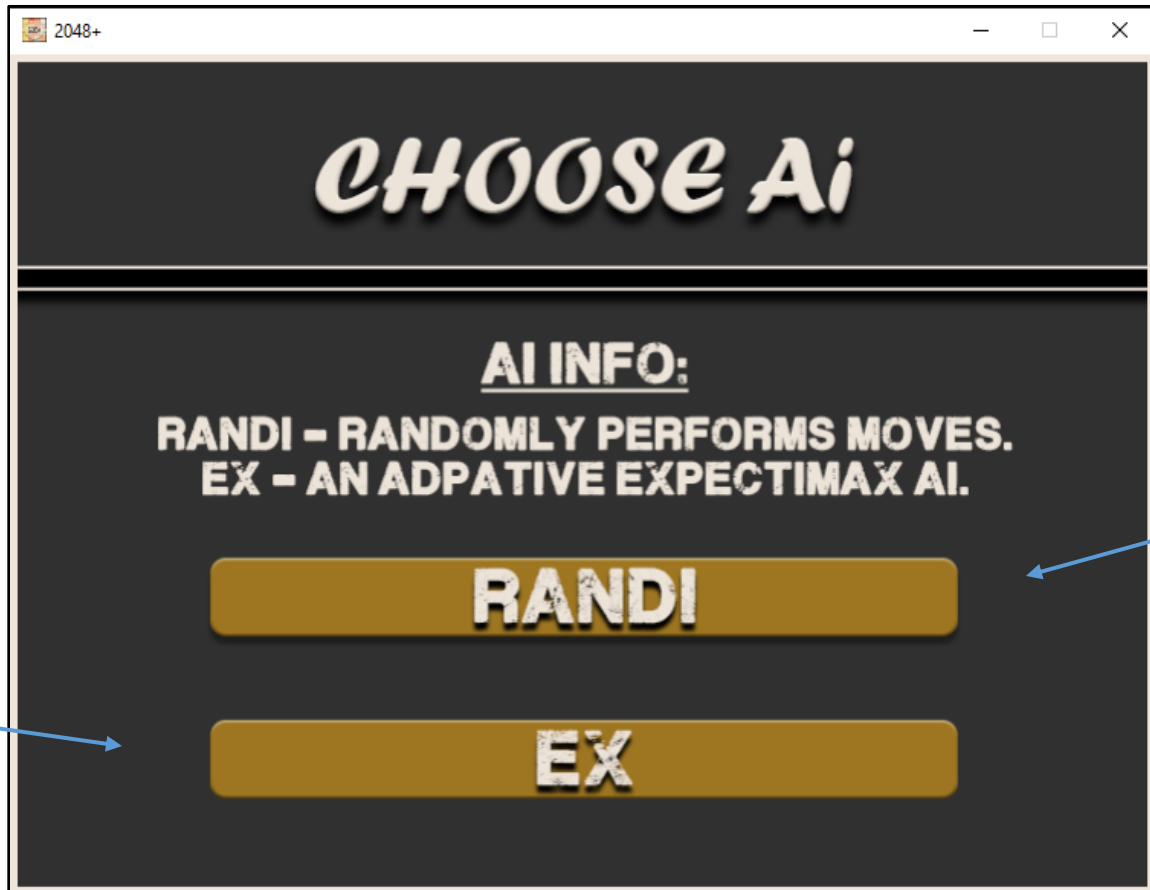
המנצח יקבע בסיום המשחק, כאשר שני השחקנים נפסלו, וזה שהגיע לתוצאה הגבוהה יותר מבין השניים – ינצח.



מסך בחירת 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 משחק.

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



מסך משחק שחקן ממוחשב:

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

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



ביבליוגרפיה

תודות:

מיכאל צ'רנובלסקי

אלון חיימוביץ

אתרי אינטרנט:

1. <https://www.youtube.com>
2. <http://en.wikipedia.org>
3. <https://stackoverflow.com>
4. <http://www.monogame.net/>

מאמרים / מחקרים / אסטרטגיות:

1. מאמר אקדמי שמדבר 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/Alfall14/vignesh_gayas_2048_project.pdf
3. אסטרטגיות למשחק 2048:
<https://www.imore.com/2048-tips-and-tricks>

נספחים – קטעי קוד מרכזיים בתוכנית:

1. מחלקת 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.
#endregion
#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 ( $\log_2(8) = 3$ )
/// 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 occurred therefore setting
change's value to true.        break; // Ending the run of the for since a fuse has
occurred.                    }

                                // Checking whether the 2nd tile was moved as a result of
the movement.                // If it was moved, we would like to switch between it's
value in the table            // to the first tile's value in the table in order to move
a tile by 1 to the            // matching direction (based on the movement direction)
each time.                    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;
}
}
#endregion

```

```
#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
checking 3 adjacent tiles
                // 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

```

2. מחלקת פותר – Solver.cs

```

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;
}
#endregion

#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
    }
}
#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

#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);
        }
    }
#endregion

```

```

        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. מחלקת אריח – *Tile.cs*

```

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;
}
#endregion

```