agar.py

תוכן עניינים

מבוא

על מה מדובר?

מאותו הרגע שהחלטתי על הרעיון לפרויקט שלי - agar.py, הדבר הראשון שעשיתי זה התחלתי לשחק את המשחק הקיים agar.io.

למי שלא מכיר, agar.io הוא משחק רשת מרובה משתתפים מסוגת אסטרטגיה-פעולה, שבו השחקן שולט על תא שנמצא במפה המזכירה צלחת פטרי. מטרת המשחק היא לצבור את כמות המסה הגדולה ביותר על ידי אכילת תאים קטנים יותר ומניעת אכילת השחקן על ידי תאים גדולים יותר.

בכל מקרה, משחק מגניב, ממליץ. באותו הרגע התחלתי לפתח תמונה אידיאלית בראש של נראות המשחק. כאשר אתם באים לפתח משחק, תתחילו בלנתח אותו. תנסו לשאול שתי שאלות מאוד פשוטות - **איך? למה?** עם השתי שאלות האלו אתם תגיעו מאוד רחוק מאוד מהר.

המסמך הזה מיועד כדי לתת תמונה כללית לגבי איך אני החלטתי לתכנת agar.io.

גרפיקה

התחלתי עם דף ועט, הגדרתי לעצמי דרישה ברורה שכהתחלה אני רוצה שהשחקן יהיה מסוגל לזוז לפי העכבר.

פתחתי python, וציירתי עיגול עם רקע כחול בהיר (pygame). לאחר מכן על הדף, התחלתי לחשוב על איך לשנות את המיקום של השחקן לפי העכבר.

תזוזת השחקן לפי עכבר

נפרק את זה לשני דברים קטנים:

1. לתת לעיגול מהירות ולפיה הוא יזוז (גודל)

הרי אחרי הכל, מהירות מוגדרת כהשינוי המיידי במיקום בכל רגע נתון, למי שמכיר - נגזרת.

לכן מן הסתם המיקום ייקבע לפי המהירות.

אבל אנחנו כבר נתקלים בבעיה מאוד ברורה, איך אנחנו יודעים אם להחסיר או להוריד מיקום? התשובה היא לפי המיקום של העכבר. זה מוביל אותנו לדבר הבא:

2. לדעת לאיזה כיוון להזיז את העיגול (לפי העכבר)

אוקיי, כעת נגדיר כלשהי מהירות לעיגול, לשם הגנריות אקרא לזה - v.

עכשיו, pygame יודע להביא לנו את מיקום העכבר ביחס ל - (0,0) של המסך. אינטואיטיבית אפשר להבין שאנחנו צריכים להזיז את השחקן בוקטור, שמוגדר לפי גודל (במקרה שלנו זו תהיה המהירות - v), וכיוון שבמקרה הזה יהיה זווית העכבר ביחס לקו שעובר דרך השחקן.

מכאן, אתחיל לקרוא לעיגול - שחקן, מכיוון שעומדת להיות לו את היכולת לזוז.

המהירות היא משהו שהגדרנו, אבל את הזווית צריך לחשב, אז נעבור על זה צעד, צעד:

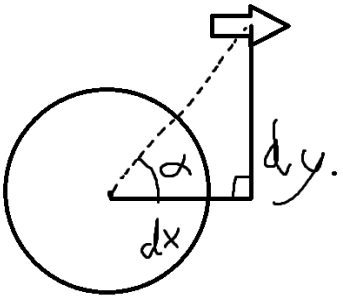
נניח שהשחקן נמצא בכלשהו מיקום - (x_{player}, y_{player}), ונניח שהעכבר נמצא במיקום - (x_{mouse}, y_{mouse}).

נקח את הפרש המיקומים:

$$dx = x_{mouse} - x_{player}$$

$$dy = y_{mouse} - y_{player}$$

כעת, השגנו את ההפרשים במיקומים. אשרטט את המצב כרגע על מנת להבהיר מה אנחנו מנסים להשיג:



ניתן להבחין כאן במשולש ישר זווית, שבו הצלע ממול הינה - dy והצלע ליד הינה - dx. מכאן אפשר למצוא את הזווית אלפא ע"י שיקולים טריגונומטריים:

$$tan(\alpha) = \frac{dy}{dx}$$

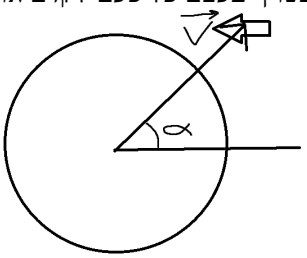
מכאן, מצאנו את הזווית הרצויה שלנו.

דגש חשוב - חובה להשתמש בפונ' atan2 **ולא** atan מכיוון שהיא לא מבדילה בין ארבעת הרביעים. לפירוט,

קראו

עכשיו יש לנו וקטור של המהירות, ונצטרך בעצם כל פעם לקדם את השחקן בכיוון הנכון לפי המהירות.

תמונת מצב:



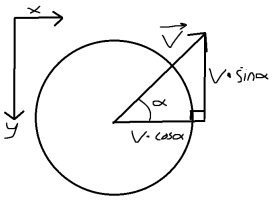
בסופו של דבר, אנחנו צריכים לקדם את השחקן בכל אחד בשני הצירים לפי הזווית והמהירות שקיבלנו כאן.

אז בעצם, אפשר ליצור עוד משולש ישר זווית רק עכשיו שהיתר הוא גודל המהירות.

מכאן נמצא את הצלע ליד ואת הצלע ממול ואלו יהיו הגדלים שאנחנו צריכים להשתמש בהם על מנת לקדם את השחקן:

$$x \mathrel{+}= v * cos(\alpha)$$

$$y \mathrel{+}= v * sin(a)$$



ככה אנחנו מפרקים את וקטור המהירות לרכיבים הקרטזיים שלו ומוסיפים את הרכיבים למיקומים.

והנה, יש לנו שחקן שיודע לזוז לפי העכבר במסך ריק לחלוטין, איזה יופי 🙂

אנחנו רק מגרדים עכשיו את ההתחלה, אז אם אתם כבר עייפים, אני ממליץ לברוח כל עוד נפשכם בידיכם.

אז עשינו קצת מתמטיקה מגניבה והצלחנו לגרום למשהו לעבוד, אבל השחקן כרגע רק זז במפה שהגודל שלה הוא כרזולוציית המסך, ואנחנו רוצים מפה מאוד גדולה שיהיו בה המון חלקי משחק ושחקנים אחרים.

לכן, נצטרך להבין איך נעשה את זה.

אכילה וגדילה

נתחיל בבסיס, המטרה ב - agar.io היא להיות המסה **הכי גדולה.** זאת אומרת שצריך לגדול.

לכן, נתחיל למקם פריטים אכילים על המפה. במשחק הסופי, השרת יחזיק את כל הפריטים, אבל כרגע בשביל הפשטות תניחו שאנחנו יודעים איפה כל אחד נמצא.

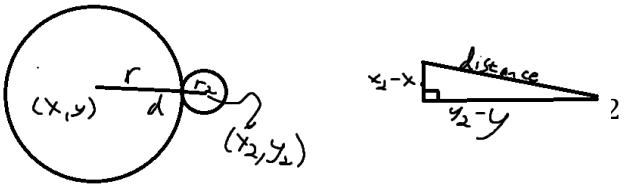
אז העיקרון כאן די פשוט, ברגע שהרדיוסים של השחקן והפריט האכיל נוגעים (שחקן נוגע בפריט האכיל) אז המסה של הפריט תתווסף לשחקן (השחקן אוכל את הפריט).

ניתן לתאר את זה מתמטית כך:

נניח מיקום השחקן ורדיוס - (x,y,r)

מיקום הפריט האכיל ורדיוס - (x2,y2,r2)

ניתן לזהות התנגשות בחלק משחק כך:



התיאור הגרפי הימני הוא מתאר את המרחק בין שני המעגלים, והשמאלי הוא מבט כללי על מנת להדגיש שהמרחק וסכום הרדיוסים שווה.

עכשיו נותר רק לאכול את הפריט, לשטח של השחקן מתווסף השטח של הפריט, ומשם אפשר לחלץ את הרדיוס:

$$area_{edible} = \pi * r2^2 \quad \Rightarrow \quad area_{new_{player}} = \pi * r^2 + area_{edible} = \pi * radius_{new_{player}}{}^2$$

$$radius_{new_{player}} = \sqrt{\frac{area_{new_{player}}}{\pi}}$$

השחקן שלנו כעת יכול לאכול חלקי משחק על המסך, והוא יכול לזוז לפי העכבר.

אנחנו נתקלים כרגע בבעיה מאוד גדולה - לאחר שאנחנו אוכלים כמה פריטים, השחקן שלנו גדל וגודלו הוא כגודל המסך.

מה שאמור לקרות זה שהמשתמש מתחיל לראות יותר ויותר מהמפה, ובאותו הזמן השחקן שלו גדל. לשם כך נפתח מספר דברים אשר יעזרו לנו להשיג את המטרה הזו.

קודם כל, נגדיר מה המשתמש רואה. המשתמש רואה דרך מסך המחשב שלו, שגודל חלון המשחק יהיה - 1920x1080. לכן, בכל רגע נתון, המשתמש רואה 1920 פיקלסים על פני הציר הרוחבי (ציר ה - x), ו - 1080 פיקסלים על הציר האנכי (ציר ה - y).

זאת אומרת שלא משנה מה יקרה לשחקן במשחק, הכמות פיקסלים האבסולוטית שהוא באמת יכול לראות היא 1920x1080. מה שמשתנה בגדילה זה טווח הראייה שלו.

אז עכשיו נשאלת השאלה - **איך אנחנו מגדילים את המסך מבלי להגדיל את הכמות פיקסלים האמיתית שלו?**

לשם כך, נצטרך להגדיר מחלקה שנקראת - **player_camera.**

<u>מצלמת השחקן</u>

מצלמת השחקן עקרונה הוא להגדיר את הכמות שהשחקן אמור לראות בכל רגע נתון (אותו עיקרון כמו FOV). **היא עוקבת אחרי השחקן.**

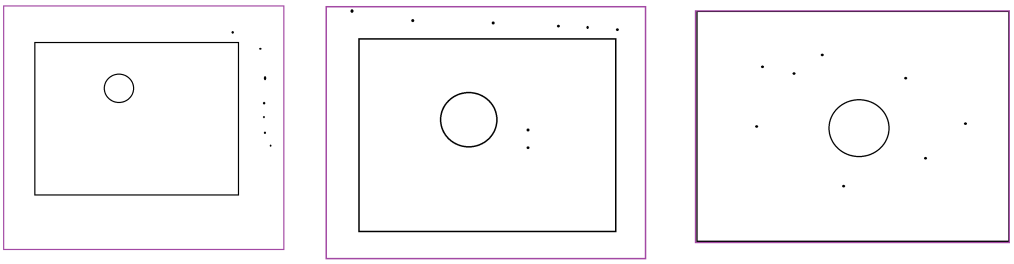
מכיוון שטווח הראייה של השחקן הוא המסך, שצורתו היא מלבן, אז מצלמת השחקן תהיה גם מלבן.

היא תהיה מלבן דמיוני שיגדל בכמות פיקסלים כלשהי כל פעם שפריט ייאכל על ידי השחקן.

<u>מקרא</u>

מלבן שחור - מסך

מלבן סגול - מצלמה



שלושת השרטוטים מתארים תהליך (מימין לשמאל) שבו מצלמת השחקן גודלת כל פעם שהשחקן אוכל פריט. בהתחלה מצלמת השחקן היא כגודל המסך, אבל לאחר כל אכילה של פריט אז היא גודלת וכך גם טווח הראייה (השחקן רואה את הטווח במלבן הסגול, **לא השחור**).

כעת סיימנו עם התיאוריה, והבנו איך אנחנו אמורים לעשות את זה. נתחיל:

מערכות צירים

המרת נקודה על המפה לנקודה על המסך

המטרה בתת נושא הזה היא להבין איך לוקחים כלשהי נקודה על המפה וממירים אותה לנקודה על המסך. נתחיל מהבסיס:

אז בעצם יש לנו שלושה "טווחים", או מערכות צירים: המסך, מצלמת השחקן והמפה כולה

אנחנו כבר מכירים את כל הדברים האלו, המפה מייצגת את מפת המשחק, והרחבנו כבר מצלמת השחקן.

המטרה שלנו היא בעצם לבצע "כיווץ" של טווח הראייה של השחקן על מנת שהכל יתאים לתוך המסך.

בואו נקח דוגמה עם מספרים יבשים:

נגדיר את גודל המסך כ - (1920,1080)

ואת גודל מצלמת השחקן כ - (2020 ,1180) (השחקן אכל כמה פריטים והמצלמת גדלה ב - 100 פיקסלים).

למען הפשטות, **נניח** שיש פריט הנמצא על גבול מצלמת השחקן (2020,1180)

עכשיו אנחנו צריכים להמיר את הקואורדינטה הזו על מנת שתתאים **למסך של השחקן.** מכיוון שהדוגמה כל כך פשוטה אז קל מאוד להבין שבגלל שהפריט נמצא בדיוק על קצה מצלמת השחקן אז הוא אמור להיות **בקצע המסך.**

זאת אומרת, צריך להפוך את הקואורדינטה הזו - (2020,1180) לקצה המסך - (1920,1080)

מכאן אנחנו מקבלים משוואה אלגברית מאוד פשוטה

$$1180 * x = 1080 \quad \Rightarrow \quad x = \frac{1080}{1180}$$

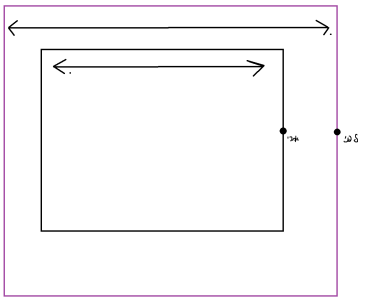
זאת אומרת שלכל מקרה ספציפי צריך למצוא את המספר המוזר הזה?

לא, מכיוון שהמשוואה האלגברית הזו נשארת קבועה אז ניתן לפתח ביטוי עבור כל מקרה:

$$transformation = \left(\frac{x_{screen}}{x_{camera}}, \frac{y_{screen}}{y_{camera}}\right)$$

עכשיו אנחנו נתקלים בבעיה, כי כל הקטע הוא לנסות למצוא את הנקודה על המסך שבה צריך לשים את הפריט, זאת אומרת שעדיין לא מצאנו את x. הטריק הוא שהמספר הזה הוא לא באמת ספציפי עבור כל מקרה, זאת אומרת שיש כלשהו magic constant שאיתו אנחנו ממירים.

כדי לפתור את הבעיה נתאר אותה גרפית -



אוקיי אז אתם כנראה שואלים את עצמכם - **מה זה עוזר לי?** אם תבחינו בחצים תבינו שהם מנסים להסביר שאפשר להשיג את המיקום שצריך לשים בו את בנקודה לפי המימדים של המסך והמצלמה!

נניח שמימדי המסך - $(width_{screen}, height_{screen})$ ומימדי מצלמת השחקן - $(width_{camera}, height_{camera})$

עכשיו מה עלינו לעשות עם המימדים האלו? נחזור למילת המפתח שציינתי בהתחלה - **לכווץ.** זאת אומרת נכווץ את כל מצלמת השחקן לתוך המסך.

בשביל זה נצטרך את **היחסים של המימדים שלהם,** כיוון שקבוע הכיווץ שאנחנו מחפשים פרופורציונאלי למימדים האלו. זאת אומרת עבור למשל ערך x כלשהו הנמצא בגבולות מצלמת השחקן, כדי להמיר למסך נכפיל ביחס בין רוחב המסך לרוחב מצלמת השחקן.ש

ובינגו! מצאנו שיטה שתמצא לנו איפה למקם נקודה על המסך לפי מיקומה במצלמת השחקן:

$$camera_{point} = (x, y) \quad \Rightarrow \quad screen_{point} = (x * \frac{width_{screen}}{width_{camera}}, y * \frac{height_{screen}}{height_{camera}})$$

זאת אומרת, עכשיו אנחנו יודעים איך לכווץ את טווח הראייה לתוך המסך.

כל זה היה בהנחה שיש לנו את הנקודה כבר ביחס למצלמת השחקן.

האמת היא שמיקום הפריטים נשמר על המפה, זאת אומרת מערכת צירים שונה לחלוטין, לכן נצטרך לבצע המרה. ההמרה מהמערכת הצירים של המסך למערכת הצירים של מצלמת מאוד פשוטה, והיא פשוט כוללת לקחת את ההפרש בין הנקודה ל - (0,0) של מצלמת השחקן.

לכן, כאשר יש לנו קואורדינטה על המפה שברצוננו להמיר אותה לנקודה על המסך, נערבב את כל מה שלמדנו כך:

$$point_{map} = (x, y) \Rightarrow point_{camera} = (x - x_{0_{camera}}, y - y_{0_{camera}})$$

$$point_{screen} = (point_{camera_x} * \frac{width_{screen}}{width_{camera}}, point_{camera_y} * \frac{height_{screen}}{height_{camera}})$$

המרת גודל רדיוס על המפה לגודל רדיוס על המסך

העיקרון כאן זהה לעיקרון בתת נושא הקודם, אבל הפעם אין צורך לבצע המרה בין מערכת הצירים של המפה למערכת הצירים של מצלמת השחקן. הרי מדובר כאן בגודל, שהוא כל הזמן יחסי למיקום, זאת אומרת שנשנה אותו רק שאנחנו רוצים להקטין אותו. מכאן, זה אותו הדבר כמו המרת נקודה:

$$camera_{radius} = r \Rightarrow screen_{radius} = camera_{radius} * \frac{width_{screen}}{width_{camera}}$$

הרחבה חלקה של טווח הראייה (אינטרפולציה)

עכשיו אנחנו יודעים איך להרחיב את הטווח ראייה של השחקן. הבעיה היא שאם למשל אנחנו אוכלים פריט שהוא מאוד גדול, אז הטווח ראייה גדל ביחס לרדיוס החדש של השחקן:

$$width_{camera} = width_{camera} * \frac{radius_{player}}{radius_{start_{player}}}$$

המכנה בשבר הוא הרדיוס שהשחקן התחיל בו.

נניח ואכלנו שחקן שהוא קצת פחות מהגודל שלנו (נזניח ונניח שהוא בערך אותו הגודל), אז הרדיוס של השחקן גדל פי $\sqrt{2}$ לפי הפיתוח הזה:

$$radius_{new_{player}} = \sqrt{\frac{area_{new_{player}}}{\pi}} = \sqrt{\frac{2\pi * radius_{player}^{2}}{\pi}} = \sqrt{2} * radius_{player}$$

(מתווספת למסת השחקן את גודל המסה שלו, זאת אומרת היא גדלה פי שתיים).

במקרה הזה, פתאום השחקן יראה שהכל נהיה יותר קטן, אבל הוא יראה את זה בקפיצה. זה מכיוון שאנחנו ישר מקטינים את כל המסך באיטרציה אחת אז זה נראה כמו באג לא תהליך אכילה.

זאת אומרת, צריך להגדיל את טווח הראייה לא בפעם אחת, אלא בכמות כלשהי של פעמים.

לכן נשתמש ב**אינטרפולציה ליניארית (lerp).**

הכלי המתמטי הזה בעצם יכול לבנות ערכים חדשים בין שתי קצוות של ערכים.

לדוגמא, נניח אנחנו רוצים להגיע מנקודה אחת לאחרת, במקום לקפוץ את הכל בפעם אחת, הכלי המתמטי הזה יודע לעשות את זה בכמות פעמים שנגיד לו.

המשוואה המתמטית היא מאוד פשוטה, נניח אנחנו רוצים להגיע מהמספר 10 למספר 20 בשתי קפיצות. אז אנחנו נגיד לכלי הזה להתקרב במכפלה של חצי.

$$10 \;\; -> \;\; 15 \;\; -> \;\; 20$$

עכשיו השאלה היא איך נעשה את זה. אפשר להבין אינטואיטיבית שצריך לעשות את זה בקפיצות של חמש, אבל איך נגיע למספר? אולי פשוט נקח את נקודת ההתחלה ונכפיל בחצי?

זה לא יעבוד, כי נניח שני המספרים הם 20 ו- 25 אז אם נעשה את מה שאמרנו אנחנו נבצע קפיצות של עשר יחידות.

לכן תדמיינו את עצמכם הולכים מ- 20 ל- 25 בשתי קפיצות, אתם תבנו לעצמכם משוואה פשוטה בראש:

$$20 \;+\; 2x \;=\; 25 \quad \Rightarrow \quad 2x \;=\; 5 \quad \Rightarrow \quad x \;=\; 2.5$$

כאן בעצם שאלנו את עצמנו כמה צריך להוסיף כל פעם כדי להגיע מ- 20 ל- 25 בשתי קפיצות, וקיבלנו את התשובה הנכונה.

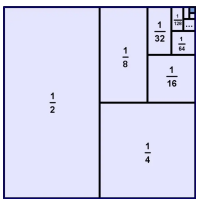
טעות נפוצה, היא לעשות ככה:

$$\begin{aligned}
&value \;=\; min \\
&step \;=\; 0.5 \\
&while\ value\ != \;\; max: \\
&\qquad value \;=\; lerp(value,\ max,\ step)
\end{aligned}$$

כאן בעצם אמרנו לעצמנו כל עוד לא הגענו למקסימום, תעשה אינטרפולציה בין הערך שלך עד לערך המקסימלי.

הטעות כאן היא אולי קצת קשה לזהות, אבל הלולאה בחיים לא תפסיק. נניח אנחנו רוצים להגיע מאפס לאחד בקפיצות של חצי, אז מה שיקרה זה כך:

$$0,\ 0.5,\ 0.75, ...$$



למרות שאנחנו שואפים לאחד, אנחנו לא מגיעים אליו. אנחנו צריכים בעצם להכפיל את הערך בקפיצה שאנחנו נמצאים בה עכשיו:

$$\begin{aligned}
&value \;=\; min \\
&step \;=\; 0.5 \\
&current_{step} \;=\; 0 \\
&while\ value\ != \;\; max: \\
&\qquad value \;=\; lerp(min,\ max,\ step\ *\ current_{step}) \\
&\qquad current_{step} = current_{step} + \;\; step
\end{aligned}$$

עכשיו, כאשר אנחנו שואפים מאפס לאחד יקרה הדבר הבא:

$$0,\ 0.5,\ 1$$

שימו לב לשינוי הכי קריטי, אנחנו עכשיו מבצעים אינטרפולציה בין המינימוס למקסימום, וזוהי אינטרפולציה נכונה.

הדבר האחרון הוא, מה lerp מחזיר? ממה שלמדנו עד עכשיו ניתן לפתח את הפונקציה הבאה:

$$\begin{aligned}
&lerp(min, max, step): \\
&\qquad return\ min \;+\; (max \;-\; min) \;*\; step
\end{aligned}$$

עוד דרך לרשום את הפונקציה מתקבלת מהפיתוח האלגברי הבא:

$$min + step(max - min) = min + step * max - step * min =$$

$$min - step * min + step * max = min * (1 - step) + max * step$$

לכן:

$$lerp(min, max, step):$$
$$\quad return\ min * (1 - step) + max * step$$

וזהו, כך פיתחתי את המחלקה interpolator בפרויקט שלי. היא טיפה שונה, אבל פועלת על אותו עיקרון מתמטי.

ארכיטקטורת שרת לקוח

המשחק agar.io הוא Real Time Multiplayer, זאת אומרת שבכל רגע נתון יש כמה שחקנים על המפה וכל אחד צריך לראות אחד את השני.

זאת אומרת, שאם נעשה את הפרויקט בשרת לקוח, אז השרת צריך לדעת איפה כל שחקן נמצא וככה הוא יוכל להעביר את המידע שנדרש לכל שחקן בנפרד.

אבל צריך לעשות את זה צעד צעד, לכן נתחיל עם לקוח אחד שמתקשר עם שרת שאומר לו כל הזמן איפה נמצאים חלקי משחק.

שרת

אתחול

השרת בתחילת המשחק יוצר חלקי משחק בגבולות שהוגדרו לו בצורה אקראית. לאחר מכן, הוא שומר את המידע הזה בתוך אובייקט שנקרא World, אשר מייצג את הפריטים שנמצאים על המפה.

לאחר מכן, השרת פותח תקשורת בשימוש באובייקט סוקט של פייתון. התקשורת מתבצעת על ידי TCP, מכיוון שכפי שנבין בהמשך, אסור לנו לאבד הודעות במהלך המשחק לכן לא נשתמש ב- UDP, למרות שהוא יותר מהיר.

ישנה מחלקת Protocol (הסבר כאן), אשר מכילה פעולות סטטיות לשליחת הודעות בין השרת ללקוח. היא מתפקדת כפרוקסי לפירוק ההודעות בין השרת ללקוח, כדי שהם לא יהיו צריכים להתייחס לזה. לאחר שלקוח התחבר, מאותחל thread עם thread_id משלו אשר משמש כדי שהשרת יזהה את ה- thread, נשלחת הודעה על גבי הפרטוקול שנקראת - server_initiate_world. לאחר מכן, נכנס השימוש של thread_id.

עדכוני לקוח

לכל thread, יש לולאה (כל עוד הלקוח מחובר) שבה הוא שולח ללקוח server_status_update שמכיל מידע חשוב על שאר השחקנים ושינוי בפריטים על המפה. לאחר שהלקוח מקבל את ההודעה המפורשת ממחלקת Protocol, הוא מעבד את המידע, מציג למסך, ומחזיר לשרת client_status_update.

עכשיו נשאלת שאלה מאוד חשובה, איך כל thread יכול לדעת איפה נמצא כל שחקן ושינויים בפריטים (שחקן אכל פריט)? הרי כל thread רץ בתוך הלולאה שלו ללא תקשורת עם ה- threadים האחרים.

משאבים משותפים

הפיתרון לבעיה הזו הוא אכן משאבים משותפים.

השימוש של המספר thread_id בא לידי ביטוי כאשר כל thread כלשהו בשרת רוצה לגשת למידע משותף של השרת. נניח שאנחנו כלשהו thread על השרת שצריך לדווח ללקוח את המידע הנדרש. חלק גדול מאוד מהמידע הזה, הוא מידע מלקוחות אחרים (איפה כל שחקן נמצא, האם שחקן אחר אכל פריט וכו'). ישנם מספר מנגנונים לטפל במקרים האלו.

מנהל עדכוני השחקנים- player_update_handler

המשאב המשותף הזה, הוא מנגנון שנועד לכך שכל thread ידע את המידע על כל שחקן. לכל שחקן יש UUID, שזה בקצרה מספר מאוד ארוך בהקסה (לא חייב להיות, אבל כך השתמשתי בו) שנותן לכל שחקן ID אחיד ומבטיח שלא יהיה לשני שחקנים את אותו UUID (ההסתברות לקבל אחד זהה כל כך נמוכה שהיא זניחה). מכיוון שעבור כל ID של שחקן יש את המידע שלו, אז אפשר להסיק שהמנגנון שלנו צריך להיות key-value. מבנה נתונים אשר מבצע את זה ביעילות הוא מילון.

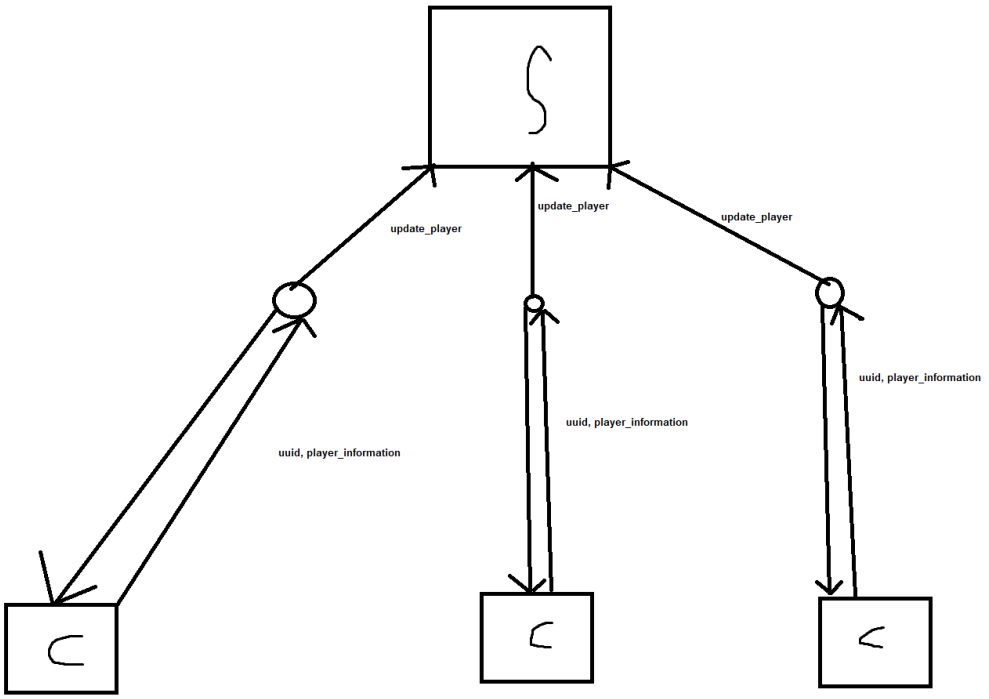
כאשר אנחנו פותחים את ה- player_update_handler, נפתח מילון שהוא ריק שכל זוג יכיל את ה- ID של השחקן, ואת המידע עליו.

כאשר אנחנו רוצים לעדכן את המידע של שחקן, ישנה פונקציה אשר נקראת update_player שמקבלת player_information שהיא מכניסה למילון (player_information מכיל גם ID). השורה שנמצאת בפעולה היא:

$$self.players\_dict[player\_information.id] = player\_information$$

הפעולה הזו נקראת כל איטרציה של כל thread, על מנת לעדכן תמיד איפה נמצא כל שחקן.

כך, נוצרת מערכת כזו:

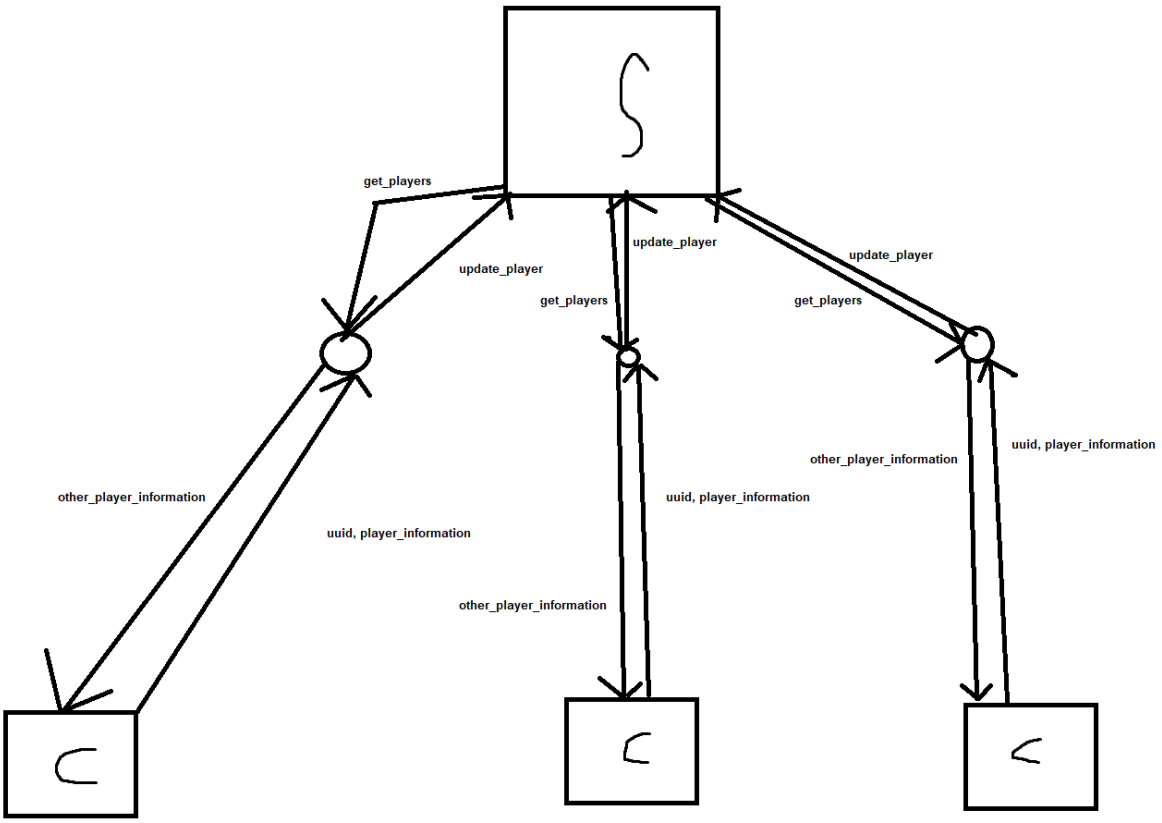


כל לקוח מעדכן כל thread איפה נמצא השחקן שלו, וה- thread מעדכן את player_update_handler על ידי update_player. עכשיו רק נשאר לתת ל- threadים את היכולת לשלוח לשחקן את המיקומים של שאר השחקנים.

לכן, player_update_handler מכיל עוד פעולה אשר נקראת get_player. הפעולה הזאת מקבלת player_information שהוא מיועד כדי שה- thread לא יגיד לשחקן לצייר את עצמו. ישנה גם אופציה לשלוח את כל השחקנים, כי בברירת מחדל הפעולה מאתחלת את player_information כ- None. השורה החשובה בפעולה, נראית כך:

$return\ [v\ for\ k,\ v\ in\ self.players\_dict.items()\ if\ k\ !=\ player\_information.id\ and\ not\ isinstance(v,\ str)]$

שורה זו מחזירה רשימה של המידע של כל השחקנים חוץ מהמידע שקיבלנו כפרמטר. נשלים את הסקיצה:



עכשיו, כל thread יודע איפה כל שחקן אחר נמצא, והוא יכול לשלוח את זה ללקוח המתאים.

מנהל עדכוני הפריטים- edible_update_handler

נשאר לנו עכשיו לעדכן את כל הלקוחות מתי נאכל פריט משחק, ומתי נוסף פריט משחק.ב- client_status_update, הלקוח שולח לשרת את המידע על השחקן שלו, **אבל בנוסף שולח את כל הפריטים שנאכלו מאז האיטרציה הקודמת.** לאחר שהשרת מקבל את ההודעה, הוא מעדכן את שאר ה- threadים שנאכל פריט משחק (חוץ מה- thread המדווח), יוצר פריט חדש בהתאמה, ומעדכן את **כל** ה- threadים לקיומו.

זה מתבצע על ידי- edible_update_handler.

כאשר נבנה אובייקט מסוג זה, נוצרת רשימה מסוג ThreadUpdateHelper. כל פעם שמאותחל thread חדש, נקראת הפעולה make_space_for_new_thread אשר מוסיפה מקום ברשימה ל- thread החדש:

$self.edible\_updates.append(ThreadUpdateHelper())$

זה נותן לכל thread אינדקס ברשימה, ופה נכנס השימוש של thread_id אשר מבצע זאת.

בכל איטרציה, בהנחה שנאכל פריט, ה- thread קודם כל מוציא את הפריט מ- World. לאחר מכן, הוא מודיע לכל ה-

threadים האחרים שהוא עשה כך, על ידי קריאה לפעולה:

```
self.edible_update_handler.notify_threads_changing_edible_status(new_edibles, edibles_eaten, thread_id)
```

הפעולה הזו, עוברת על כל אינדקס ברשימה ומבצעת את הפעולה המובנית של ThreadUpdateHelper שנקראת

update_edible_statuses:

```
for i in range(len(self.edible_updates)):
    if i != thread_id:
        self.edible_updates[i].update_edible_statuses(edibles_created, edibles_removed)
```

חשוב לציין כי ה- thread הזה אינו מודיע לעצמו על השינויים שנעשו בפריטים, מכיוון שהוא מודע להם כבר כי הוא קרא

לפעולה המעדכנת. לאחר מכן, ה- thread קורא לפעולה: fetch_thread_specific_edible_updates:

```
self.edible_updates[thread_id].fetch_edibles_removed(), self.edible_updates[thread_id].fetch_edibles_created()
```

לאחר מכן ה- thread מחזיר ללקוח ב- server_status_update את השינויים שחלו בכל פריט מאז הסטטוס האחרון. חשוב לציין כי

פעולת fetch נקראת כל איטרציה בכל מקרה, מכיוון שייתכן ש- thread אחר עדכן את השאר.

כעת נכנס יותר פנימה, ונדון ב- ThreadUpdateHelper.

עוזר עדכוני פריטים - ThreadUpdateHelper

המשאב EdibleUpdateHandler משתמש בעוזר זה, על מנת לתת לכל thread את העדכונים שלו לגבי הפריטים.

בבניית אובייקט, נוצרות שתי רשימות שמסמלות פריטים שנאכלו ופריטים שנוספו.

למחלקה זו יש שני סוגי פעולות: update ו- fetch. לכל סוג יש פעולה לפריטים שנאכלו ולפריטים שנוספו. הלוגיקות זהות,

לכן לא משנה איזה סט פעולות נציג. נקח את edibles_created כדוגמא:

הפעולה הראשונה, update_edibles_created פשוט מוסיפה לרשימה את כל הפריטים שהועברו כפרמטר (מועברת רשימה):

```
for edible in edibles_created:
    self.edibles_created.append(edible)
```

הפעולה השניה, fetch_edibles_created, מוציאה מהרשימה את כל הפריטים שנוצרו:

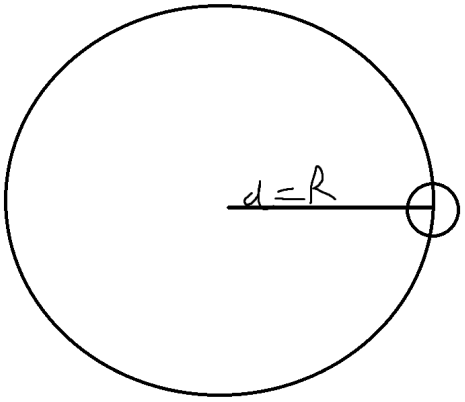
```
edibles_created = self.edibles_created.copy()
self.edibles_created.clear()
return edibles_created
```

זוהי מטרת המחלקה.

אוקיי, אז עכשיו סיימנו נכון? עדיין לא, צריך עכשיו רק לזהות התנגשויות.

<u>זיהוי ההתנגשויות</u>

כדי לזהות התנגשויות, ניצור עוד thread שמטרתו היחידה היא לזהות התנגשויות בין שחקנים. כדי לבדוק אם שני שחקנים מתנגשים, נבדוק אם חצי או יותר מהשחקן עם הרדיוס היותר קטן נמצא בתוך השחקן הגדול:



ניתן לראות בסקיצה מימין בקלות שזה קורא כאשר המרחק קטן או שווה לרדיוס של השחקן שיותר גדול.

לכן ניתן לפתח את התנאי, שכאשר הוא מחזיר אמת אז השחקן היותר קטן נאכל:

$$distance(\ (x_1, y_1),\ (x_2,\ y_2)\ )\ <=\ max(r_1, r_2)$$

כדי לבצע את זה בקוד, צריך לבדוק **עבור כל שחקן**, אם הוא **מתנגש בכל שחקן אחר.** ניתן להסיק מניסוח הזה שמדובר בלולאה מקוננת פשוטה שבודקת אם קיימות התנגשויות. נקרא לפעולה הזו: detect_collisions.

ה- thread של ההתנגשויות צריך לקרוא לפעולה הזו כל איטרציה כדי לבדוק אם קיימות התנגשויות.

יופי! אז סיימנו!

לא, יש כאן עוד בעיה די גדולה. הבעיה היא בתזמונים בין threads. יכול מאוד להיות שה- thread שמנהל התנגשויות הכניס את אותה ההתנגשות כמה פעמים, מכיוון שה- thread שאחראי על הלקוח שנאכל, לא הספיק להוציא אותו. על מנת לפתור את התקלה הזו, נשתמש בטיפוס מובנה של פייתון שנקרא set. מבנה זה הוא בעצם קבוצה, והגדרתה של קבוצה היא שכל איבר יכול להתקיים בה אך ורק פעם אחת.

לכן, ברגע שאנו מזהים התנגשות, נוסיף את זה לקבוצה שיצרנו.

עכשיו נותר רק לדווח ל- threadים הרלוונטים על התנגשות.

עוזר זיהוי ההתנגשויות - players_eaten_helper

מחלקה זו מכילה רשימה עם אינדקס עבור כל thread שמאתחל מקום משלו באתחולו (כמו edible_update_handler), מטיפוס PlayersEatenInformation.

בכל פעם שיש התנגשות, נקראות שתי פעולות- killed, עבור ה- thread שהלקוח שלו נהרג, ו- ate_player עבור ה- thread שהלקוח שלו אכל את הנהרג. שתיהן מקבלות thread_id כאינדקס. הפעולה killed בעצם שמה באינדקס את ההודעה- KILLED. הפעולה ate_player, קוראת לפעולה בתוך PlayersEatenInformation אשר מחשבת את הרדיוס החדש של השחקן לפי הוספת מסה (כפי שחישבנו באכילה וגדילה).

ישנו משאב משותף מסוג מחלקת PlayersEatenHelper, שכל thread מבקש ממנו עדכון כל איטרציה לגבי סטטוס התנגשות ופועל בהתאם לפי פרוטוקול ההריגה.

נעילות

אז עכשיו, יש לנו משאבים משותפים כדי שכל ה- threadים יוכלו לתקשר אחד עם השני. מה יקרה אם שניים או יותר ינסו לגשת למשאב משותף באותו הזמן? יכול לקרות race condition, אשר ישנה ערכים בצורה אקראית. לכן יש לשרת lock שכל thread מפעיל בתחילת השימוש במשאבים המשותפים ומכבה בסוף השימוש.

פרוטוקול הריגה

כל thread פועל עד שלקוח המיועד אליו נאכל. כאשר הוא נהיה מודע לזה, הוא מדליק דגל שמפסיק את הלולאת עדכונים. מיד לאחר מכן, הוא מודיע לפרוטוקול לעשות override להודעה שתישלח מכיוון שהשחקן נהרג. לבסוף, הוא מוציא את השחקן מ- player_update_handler ומשמיד את עצמו.

לקוח

הלקוח מקיים תקשורת עם השרת שלמדנו עליה כבר חלקית, אבל עכשיו נעבור עליה לעומק. יש לו גם את החלק של העדכונים המחזוריים של הגרפים, אבל עליהם אין צורך לעבור מכיוון שעברנו על העקרונות בפרק הגרפיקה. לכן, נדבר פה רק על הפן התקשורתי, וכל דבר שקשור לשניהם, נציין אך לא ננמק עד הסוף.

אתחול

בפעולה אשר בונה את תקשורת הלקוח, אנו מאתחלים שני דברים: world_information ו- edible_eaten_list.

האובייקט world_information מכיל מידע על כל הפריטים כרגע, ו- edible_eaten_list היא בעצם הדבר היחיד שהחלק הגרפי (שעושה את החלקים הגרפיים, כגון לבדוק האם נאכל פריט ולפעול בהתאם) מדווח לחלק התקשורתי.

לאחר אתחולם, נוצר thread נפרד אשר נועד לתקשורת בין השרת ללקוח. הרי זה חייב להיות אסינכרוני מכיוון שאין אפשרות לחכות להודעות מהשרת שהשחקן צריך לשחק כל הזמן.

עדכונים

ה- thread הזה כל איטרציה שולח לשרת client_status_update לגבי המידע עליו וכל אכילת פריט (כאן נעשה השימוש ב- edible_eaten_list). מייד לאחר מכן ה- thread מנקה את הרשימה מכיוון שרגע לפני כן דיווח לשרת. אחרי זה הלקוח מקבל הודעת server_status_update, ובמידה וחל שינוי בפריטים, הוא משתמש ב- world_information כדי לעדכן כי world_information הוא האובייקט המגשר בין הפן הגרפי של הלקוח לתקשורתי.

הריגה

במידה והלקוח נהרג (קבלת ההודעה ב- server_update), אז הוא מנתק את התקשורת מהשרת ומכבה את עצמו.

הפרוטוקול

הפרוטוקול של השרת והלקוח הוא אסינכרוני, מכיוון שהמשחק הינו Real Time (הנושאים עד לכאן הוכיחו עד כמה צריך את זה).

בנושא הזה, נפרט על הפן הטכני של איך הודעות נשלחות.

הפרוטוקול ממומש על ידי מחלקה עוטפת שנקראת game_protocol. מחלקה זאת מכילה פעולות סטטיות כדי שהשרת והלקוח יוכלו להשתמש בהן. המחלקה מכילה שש פעולות, שתיים עבור אתחול וארבע עבור עדכוני סטטוסים.

לאתחול העולם יש רק generate ו- parse, אבל לעדכונים יש generate ו- parse עבור השרת והלקוח.

אתחול העולם

מייד לאחר שהלקוח התחבר לשרת, הוא מחכה להודעה ממנו לגבי כל הפריטים הנמצאים בעולם. הלקוח שולח את ההודעה דרך הפעולה- server_initiate_world. הפעולה מקבלת מספר פרמטרים ולפיהם מרכיבה הודעה מהסוג הבא:

~world_size_x,world_size_y~edible_x,edible_y,edible_color,edible_radius~

הלקוח מפענח את ההודעה בעזרת הפעולה- parse_server_initiate_world

עדכונים

שרת

בפעולת ה- generate של השרת, הפרטוקול יוצר לשרת הודעה לשלוח ללקוח לפי הפורמט הבא:

special seperator for this type of message - ~!"# -> signifies transition between data

message -> [size]~5,5,(2,2,2),6~...~!"#6,5,4...~23,45,(..),5~!"....~

המבדיל המיוחד נועד להבדיל בין השחקנים, לפריטים שנוצרו ולפריטים שנאכלו.

פעולת ה- parse מחזירה ללקוח את המידע בצורה נוחה (במבני נתונים).

במידה והשחקן נהרג, תוחזר ההודעה המיוחדת - EATEN.

לקוח

בפעולת ה- generate של הלקוח, הוא שולח לשרת הודעה מהסוג הבא:

message - ~name,x,y,radius~...~

... - Send edible information if it was eaten, so server can remove it from the playing field

פעולת ה- parse של הלקוח מחזירה לשרת את המידע בצורה נוחה.

טבלאות מסכמות

טבלאת פעולות

לפעולות parse אין מבנה הודעה.

פעולה	תיאור	מבנה הודעה
server_initiate_world	שולחת ללקוח הודעה של כל הפריטים בעולם, עם גודל העולם.	~world_size_x,world_size_y~edible_x,edible_y,edible_color,edible_radius~.....~ ..... - מייצג כל פריט בפורמט הבא: edible.platform_x,edible.platform_y,tup,edible.radius~
parse_server_initiate_world	קבלת ההודעה server_initiate_world.	
generate_client_status_update	יוצרת הודעה ששולחת לצד השרת מידע עבור הלקוח.	~name,x,y,radius~...~ ... - מסמל פריטים שנאכלו (צ'אנק עבור כל פריט)
parse_client_status_update	קבלת ההודעה generate_client_status_update.	
generate_server_status_update	שולחת ללקוח מידע עבור כל שחקן אחר, כל הפריטים שנוספו ונאכלו, ובמידת הצורך, רדיוס חדש. במידה והשחקן נהרג, נשלחת הודעה מיוחדת	~....#"!~5,(..),45,23~...~4,5,6#"!~...~6,(2,2,2),5,5 המבדיל המיוחד נועד כדי להבדיל בין צ'אנקים גדולים של מידע, כגון בין מידע על שחקנים למידע על פריטים שנוספו. במידה והשחקן נהרג - EATEN
parse_server_status_update	קבלת ההודעה generate_server_status_update	

<u>דוגמא לתקשורת בין שרת ללקוח</u>

תיאור	הודעה	כיוון
גודל העולם הוא - 20000,20000 נוצרו מספר פריטים שכולם עם הצבע השחור (0:0:0), עם אותו הרדיוס- 20, ונמצאים בנקודות שונות על המפה	~20000,20000~5078,6825,(0: 0: 0),20~16730,19666,(0: 0: 0),20~12141,16381,(0: 0: 0),20~16694,18747,(0: 0: 0),20~3132,4637,(0: 0: 0),20~10102,6828,(0: 0: 0),20~	שרת -> לקוח server_initiate_world
ה- ID של השחקן הוא - 8b2f225a31794879812767ac156b4357 שמו- Niran מיקומו- 8463.883, 3724.16 גודלו- 101.98 מידע על פריט שנאכל- 3647,8547,(0: 0: 0),20	~8b2f225a31794879812767ac156b4357,Niran,3724.16,8463.883,101.98~3647,8547,(0: 0: 0),20~	לקוח -> שרת generate_client_status_update
נוצר פריט- 3921,11498,(0: 0: 0),20 מידע על שחקן אחר שנמצא על המפה- Johnny, 5164.2, 7399.41, 131.160 פריט שנמחק- 5266,7496,(0: 0: 0),20 הפרמטר האחרון שערכו 0, מסמל שאין שינוי רדיוס לשחקן מכיוון שערכו אפס.	~3921,11498,(0: 0: 0),20~!"#Johnny,5164.278940742695,7399.41039514396 4,131.14877048604004~!"#5266,7496,(0: 0: 0),20~0~	שרת -> לקוח generate_server_status_update
עוד תקשורת.....	עוד תקשורת.....	עוד תקשורת.....
השחקן נאכל על ידי שחקן אחר	EATEN	שרת -> לקוח generate_server_status_update

קוד

constants.py

```
import pygame


class PlayerCameraConstants:
    SCREEN_WIDTH = 1920
    SCREEN_HEIGHT = 1080
    GRID_LINE_COLOR = 180, 180, 180
    BACKGROUND_COLOR = 173, 216, 230
    WINDOW_GRID_SPACING = 50  # pixels


class PlayerConstants:
    PLAYER_VELOCITY = 7.5  # pixels per 1/ FPS seconds
    PLAYER_COLOR = (0, 200, 200)
    PLAYER_OUTLINE_COLOR = 115, 147, 179
    PLAYER_STARTING_OUTLINE_THICKNESS = 1
    PLAYER_LOCATION_CAMERA = (PlayerCameraConstants.SCREEN_WIDTH / 2,
PlayerCameraConstants.SCREEN_HEIGHT / 2)
    PLAYER_STARTING_RADIUS = 100
    DISTANCE_THRESHOLD = PLAYER_VELOCITY

class EdibleConstants:
    EDIBLE_RADIUS = 20
    EDIBLE_COLOR = (0,0,0)
    AMOUNT_OF_EDIBLES = 4000


class PlatformConstants:
    PLATFORM_HEIGHT = 20000
```

```
    PLATFORM_WIDTH = 20000

class GameSettings:
    GAME_NAME = "Agar.py"
    FPS = 60
```

coordinate_system.py

```
from src.constants import *
import math

"""
    This class is a helper that transforms platform coords and sizes to fit to the screen


"""


class CoordinateSystemHelper:

    def __init__(self, player_camera):
        self.player_camera = player_camera

    """
        platform_pos -> tuple

        accepts a position in the platform coordinate system (world coords), and squeezes pos into screen
(1920, 1080)
    """

    def platform_to_screen_coordinates(self, platform_pos):
        # Get ratio
        width_ratio = 1 / (self.player_camera.width / PlayerCameraConstants.SCREEN_WIDTH)
        camera_relative_x = platform_pos[0] - self.player_camera.x
        camera_relative_y = platform_pos[1] - self.player_camera.y
        # scale by ratio (screen top left 0,0)
```

```
        screen_relative_x = camera_relative_x * width_ratio
        screen_relative_y = camera_relative_y * width_ratio

        return screen_relative_x, screen_relative_y

    def platform_to_screen_radius(self, platform_radius):
        width_ratio = 1 / (self.player_camera.width / PlayerCameraConstants.SCREEN_WIDTH)
        return platform_radius * width_ratio

    def platform_to_screen(self, platform_pos, platform_radius):
        return (self.platform_to_screen_coordinates(platform_pos)),
self.platform_to_screen_radius(platform_radius)
```

edible.py

```
from src.constants import *
import math

"""
    Returns Euclidean distance between two points
    positions in: (x,y)
"""


def get_distance(pos1, pos2):
    return math.hypot(pos1[0] - pos2[0], pos1[1] - pos2[1])


"""
    Edible class, can be eaten by a player.
    Accepts x,y in PLATFORM coords, so it can be determined whether or not to show the edible

"""


class Edible:
    def __init__(self, x, y, color, radius=EdibleConstants.EDIBLE_RADIUS):
```

```
        self.radius = radius
        self.platform_x = x
        self.platform_y = y
        self.color = color

    def should_be_eaten(self, player_pos, player_radius):
        return get_distance(player_pos, (self.platform_x, self.platform_y)) < player_radius + self.radius

    """
        Draw on screen, accepts camera relative coords
    """

    def draw(self, surface, screen_relative_pos, radius):
        pygame.draw.circle(surface, self.color, (screen_relative_pos[0],
                                                 screen_relative_pos[1]), radius)

    def get_position(self):
        return self.platform_x, self.platform_y

    def __str__(self):
        return f"{self.x},{self.y}"

    def __eq__(self, other):
        return isinstance(other, Edible) and other.platform_y == self.platform_y and other.platform_x ==
self.platform_x \
            and other.radius == self.radius
```

interpolator.py

```
"""
    Represents an interpolator, used for smooth scaling

    takes in x, y as starting points, and has dynamic target points.

"""
class Interpolator:
```

```
    def __init__(self, increaser, x):
        self.increaser = increaser
        self.current_increaser = increaser
        self.x = x

        self.is_lerping = False

        self.target_x = x

    def init_lerp(self, x, target_x):
        self.x = x
        self.target_x = target_x
        self.is_lerping = True
        self.current_increaser = self.increaser


    def lerp(self):
        if self.is_lerping:
            res_x = self.x + (self.target_x - self.x) * self.current_increaser
            self.current_increaser = min(self.increaser + self.current_increaser, 1)
            if self.current_increaser == 1:
                self.is_lerping = False
            return res_x
        else:
            return self.target_x
```

player.py

```
import math

import pygame.font
import random
from src.constants import *
from src.interpolator import Interpolator

"""
    Returns Euclidean distance between two points
```

```
    positions in: (x,y)
"""


def get_distance(pos1, pos2):
    return math.hypot(pos1[0] - pos2[0], pos1[1] - pos2[1])


"""
    Player class
    Used for drawing and moving the player
"""

POSSIBLE_FONT_SIZES = range(10, 40)


def get_max_font_size(text, width):
    for size in reversed(POSSIBLE_FONT_SIZES):
        font = pygame.font.SysFont(None, size)
        if font.size(text)[0] <= width:
            return size
    return POSSIBLE_FONT_SIZES[0]


class Player:
    def __init__(self, name):  # velocity measured in pixels per second
        self.name = name
        self.velocity = math.fabs(PlayerConstants.PLAYER_VELOCITY)
        # actual player pos goes by the platform he is on
        self.radius = PlayerConstants.PLAYER_STARTING_RADIUS
        self.x = random.randint(self.radius, PlatformConstants.PLATFORM_WIDTH - self.radius)
        self.y = random.randint(self.radius, PlatformConstants.PLATFORM_HEIGHT - self.radius)

    """
        Function calculates current position change with respect to the mouse (follower)
        and current velocity
```

The player doesn't actually move, because of that the position should be with respect to actual screen coords

But when taking into account the mouse, the change in the position should be with respect to the playerCamera

```
    """

    def move(self):
        dx, dy = pygame.mouse.get_pos()
        # Magnitude of velocity with direction of mouse

        angle = math.atan2(dy - PlayerCameraConstants.SCREEN_HEIGHT / 2,
                           dx - PlayerCameraConstants.SCREEN_WIDTH / 2)

        distance = get_distance((PlayerCameraConstants.SCREEN_WIDTH / 2,
                                 PlayerCameraConstants.SCREEN_HEIGHT / 2), (dx, dy))
        if distance < PlayerConstants.DISTANCE_THRESHOLD:
            self.velocity = 0
        else:
            self.velocity = PlayerConstants.PLAYER_VELOCITY

        self.x += self.velocity * math.cos(angle)
        self.y += self.velocity * math.sin(angle)

    def draw(self, color, surface, coordinate_helper):
        screen_radius = coordinate_helper.platform_to_screen_radius(self.radius)
        screen_x, screen_y = coordinate_helper.platform_to_screen_coordinates(self.get_position())
        pygame.draw.circle(surface, color, (screen_x, screen_y), screen_radius)
        pygame.draw.circle(surface, PlayerConstants.PLAYER_OUTLINE_COLOR, (screen_x,
screen_y),
                           screen_radius,
                           PlayerConstants.PLAYER_STARTING_OUTLINE_THICKNESS)

        cell_size = self.radius*2
        font_size = int(get_max_font_size(self.name, self.radius))
        font = pygame.font.SysFont("Arial", font_size)
        name_surface = font.render(self.name, True, (255,255,255))
        name_rect = name_surface.get_rect()
```

```
        name_rect.center = (screen_x, screen_y)
        surface.blit(name_surface, name_rect)


    """
        Runs periodically, HAS to be called by the game handler
    """

    def execute(self, drawColor, surface, coordinate_helper):
        self.move()
        #print(f"Player pos: {self.x},{self.y}" )
        self.draw(drawColor, surface, coordinate_helper)

    """
        Increases size by area of edible, returns radius change
    """

    def eat(self):
        old_radius = self.radius
        self.radius = ((math.pi * self.radius ** 2 + math.pi * EdibleConstants.EDIBLE_RADIUS ** 2) /
math.pi) ** 0.5
        old_area = old_radius ** 2 * math.pi
        area = self.radius ** 2 * math.pi
        return self.radius - old_radius

    def get_position(self):
        return self.x, self.y
```

player_camera.py

```
from src.constants import *
from src.interpolator import Interpolator
from src.coordinate_system import CoordinateSystemHelper
"""
    This class represents the player camera.
    It manages every 'Drawable', including the functionality of making the camera bigger and smaller.
    To do this, it's position is modified when the player gets bigger,
```

```
"""


class PlayerCamera:
    def __init__(self, window):
        self.window = window
        pygame.display.set_caption(GameSettings.GAME_NAME)
        self.x = 0
        self.y = 0
        self.width = PlayerCameraConstants.SCREEN_WIDTH
        self.height = PlayerCameraConstants.SCREEN_HEIGHT

        self.coordinate_helper = CoordinateSystemHelper(self)

        self.width_interpolator = Interpolator(0.05, self.width)
        self.height_interpolator = Interpolator(0.05, self.height)

    """
        Periodic function, player camera moves with player
        because of this, new pos should be player pos
    """
    def update_window(self, player_pos):
        self.window.fill(PlayerCameraConstants.BACKGROUND_COLOR)
        self.update_position(player_pos)
        self.width = self.width_interpolator.lerp()
        self.height = self.height_interpolator.lerp()
        # self.draw_grids(player_pos)
        # print(f"{self.width},{self.height}")


    """
        Scales height and width of the camera,
        the scalars scale the size by the size of the screen

    """
    def edible_eaten(self, width_scalar, height_scalar):
```

```
        self.width_interpolator.init_lerp(self.width, PlayerCameraConstants.SCREEN_WIDTH *
width_scalar)
        self.height_interpolator.init_lerp(self.height, PlayerCameraConstants.SCREEN_HEIGHT *
height_scalar)

    def draw_edible(self, edible):
        camera_relative_position, edible_radius = self.coordinate_helper.platform_to_screen(
                                                                    edible.get_position(),
                                                                    edible.radius)
        if not camera_relative_position[0] < 0:
            edible.draw(self.window, camera_relative_position, edible_radius)

    """
        Updates position according to player coords
        Position is saved in Platform coordinates, to determine actual location
    """
    def update_position(self, player_pos):
        self.x = player_pos[0] - self.width / 2
        self.y = player_pos[1] - self.height / 2

    def get_position(self):
        return self.x, self.y
```

world.py

```
"""
    Represents the world, all edibles and players
"""
import random

from src.constants import EdibleConstants, PlatformConstants
from src.edible import Edible
```

```
class World:

    def __init__(self, width, height):
        self.edibles = []
        self.players = []
        self.width = width
        self.height = height

    def spawn_edibles(self, amount):
        for i in range(amount):
            self.edibles.append(self.spawn_edible(EdibleConstants.EDIBLE_RADIUS,
EdibleConstants.EDIBLE_COLOR))

    def spawn_edible(self, radius, color):
        return Edible(random.randint(radius, self.width - radius),
                      random.randint(radius, self.height - radius), color)

    """
        Deletes edible
        returns new edible that has spawned in the world
    """
    def delete_edible(self, edible):
        # eq overridden in edible class
        self.edibles.remove(edible)
        new_edible = self.spawn_edible(EdibleConstants.EDIBLE_RADIUS,
EdibleConstants.EDIBLE_COLOR)
        self.edibles.append(new_edible)
        print(f"Edible at location: ({edible.platform_x},{edible.platform_y}) has been removed")
        print(f"Created new edible at location: ({new_edible.platform_x},{new_edible.platform_y})")
        return new_edible
```

client.py

```
import math
import random
```

```
import socket
import sys
import threading
import uuid

import pygame

from src.constants import GameSettings, PlayerConstants, EdibleConstants, PlatformConstants
from src.networking.helpers import game_protocol
from src.edible import Edible
from src.networking.information.player_information import PlayerInformation
from src.networking.helpers.game_protocol import Protocol
from src.player import Player
from src.player_camera import PlayerCamera
from src.networking.helpers.utils import recv_by_size, send_with_size
from uuid import uuid4

POSSIBLE_FONT_SIZES = range(10, 40)
def get_max_font_size(text, width):
    for size in reversed(POSSIBLE_FONT_SIZES):
        font = pygame.font.SysFont(None, size)
        if font.size(text)[0] <= width:
            return size
    return POSSIBLE_FONT_SIZES[0]

window = None
pygame.init()
score = 0
FONT = pygame.font.SysFont('arial', 40)
running = True
player : Player = None
player_camera : PlayerCamera = None
class WorldInformation:

    def __init__(self):
        self.width = 0
        self.height = 0
```

```
        self.edibles = []
        self.players : [PlayerInformation] = []

    def initiate_edibles(self, edibles: [Edible]):
        self.edibles = edibles

    def __add_edible(self, edible):
        self.edibles.append(edible)

    def add_edibles(self, edibles: [Edible]):
        for edible in edibles:
            self.__add_edible(edible)

    def remove_edibles(self, edibles_removed):
        for edible in edibles_removed:
            self.edibles.remove(edible)

    def set_players(self, other_players):
        self.players = other_players

class Client:
    def __init__(self, host, port, world_information: WorldInformation, player_information:
PlayerInformation):
        self.socket = socket.socket()
        self.thread = None
        try:
            self.socket.connect((host, port))
            print("Connected")
        except:
            print("Connection error, please check ip or port!")
            sys.exit()

        # To Handle sending data
        self.world_information = world_information
        self.player_information = player_information
        self.edible_eaten_list = list()
```

```
    """
        Starts recieving and sending messages, opens a seperate thread
    """

    def start_client(self):
        message = recv_by_size(self.socket)
        world_size, edibles = game_protocol.Protocol.parse_server_initiate_world(message)
        self.world_information.initiate_edibles(edibles)
        self.world_information.width, self.world_information.height = world_size

        self.thread = threading.Thread(target=self.__handle_connection, args=())
        self.thread.start()



    """
        Main client func, communicates with the server and updates the server on relevant information
    """
    def __handle_connection(self):
        global running
        unique_id = uuid.uuid4().hex
        while running:
            message = Protocol.generate_client_status_update(self.player_information.x,
self.player_information.y,
                                                             self.player_information.radius,
                                                             self.player_information.name,
                                                             unique_id,
                                                             self.edible_eaten_list.copy())
            self.edible_eaten_list.clear()

            send_with_size(self.socket, message)  # update the server on relevant information
            server_reply = recv_by_size(self.socket)

            if Protocol.parse_server_status_update(server_reply) == "EATEN":
                running = False
                print("bye")
                break
```

```
        edibles_created, other_players, edibles_removed, ate_inc =
Protocol.parse_server_status_update(server_reply)

        self.world_information.remove_edibles(edibles_removed)
        self.world_information.add_edibles(edibles_created)
        self.world_information.set_players(other_players)

        global player
        player.radius += ate_inc
        player_camera.edible_eaten(player.radius / PlayerConstants.PLAYER_STARTING_RADIUS,
                                   player.radius / PlayerConstants.PLAYER_STARTING_RADIUS)




    """
        Will add to a queue for the thread to send to the server
    """

    def notify_eaten_edible(self, edible: Edible):
        self.edible_eaten_list.append(edible)

    """
        This information is sent to the server to update location
    """

    def update_player_information(self, x, y, radius):
        self.player_information.x = x
        self.player_information.y = y
        self.player_information.radius = radius




def update_window(player, player_camera, edibles, client: Client, other_player_information):
    player_camera.update_window(player.get_position())
```

```
    update_edibles(player, player_camera, edibles, client)
    draw_other_players(other_player_information, player_camera.coordinate_helper)
    player.execute(PlayerConstants.PLAYER_COLOR, window, player_camera.coordinate_helper)
    update_score()
    client.update_player_information(player.x, player.y, player.radius)
    pygame.display.flip()


def draw_other_players(other_player_information : [PlayerInformation], coords):
    for player_information in other_player_information:
        draw_other_player(player_information.x, player_information.y, player_information.radius,
PlayerConstants.PLAYER_COLOR, player_information.name, coords)


def draw_other_player(x, y, radius, color, name, coordinate_helper):
    screen_radius = coordinate_helper.platform_to_screen_radius(radius)
    screen_x, screen_y = coordinate_helper.platform_to_screen_coordinates((x, y))

    if not screen_x < 0:
        pygame.draw.circle(window, color, (screen_x, screen_y), screen_radius)
        pygame.draw.circle(window, PlayerConstants.PLAYER_OUTLINE_COLOR, (screen_x,
screen_y),
                           screen_radius,
                           PlayerConstants.PLAYER_STARTING_OUTLINE_THICKNESS)
        font_size = int(get_max_font_size(name, radius))
        font = pygame.font.SysFont("Arial", font_size)
        name_surface = font.render(name, True, (255,255,255))
        name_rect = name_surface.get_rect()
        name_rect.center = (screen_x, screen_y)
        window.blit(name_surface, name_rect)


def draw_text(text, font, text_col, x, y):
    img = font.render(text, True, text_col)
    window.blit(img, (x, y))
```

```
def update_score():
    global score
    text = FONT.render(f"Score: {score}", True, (255, 255, 255))
    text_rect = text.get_rect()
    window.blit(text, text_rect)


def update_edibles(player, player_camera, edibles, client):
    global score

    for edible in edibles:
        player_camera.draw_edible(edible)
        if edible.should_be_eaten(player.get_position(), player.radius):
            score += 1
            client.notify_eaten_edible(edible)  # notify the server that the player has eaten an edible
            scale = player.radius / PlayerConstants.PLAYER_STARTING_RADIUS
            player_camera.edible_eaten(scale,
                                       scale)
            player.eat()
            edibles.remove(edible)


def start(name, ip, port, screen):

    global window
    window = screen

    global player
    global player_camera
    player = Player(name)
    player_camera = PlayerCamera(window)
    world_information = WorldInformation()
    player_information = PlayerInformation(player.x, player.y, player.radius, player.name)
    print(f"Client will connect to: {ip}:{port}")
    client = Client(ip, port, world_information, player_information)
    client.start_client()
```

```
    clock = pygame.time.Clock()
    global running
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
        update_window(player, player_camera, world_information.edibles, client,
world_information.players)
        clock.tick(GameSettings.FPS)
    pygame.quit()
```

server.py

```
import threading

from src.constants import EdibleConstants, PlatformConstants
from src.networking.handlers.collision_detector import CollisionDetector
from src.networking.handlers.player_update_handler import PlayerUpdateHandler
from src.networking.handlers.edible_update_handler import EdibleUpdateHandler

from src.world import World
import socket
from src.networking.helpers.utils import send_with_size, recv_by_size
from src.networking.helpers.game_protocol import Protocol
from threading import Lock

world: World = None
lock = Lock()
```

```
def collision_exists(player1, player2):
    dist = ((player1.x - player2.x) ** 2 + (player1.y - player2.y) ** 2) ** 0.5
    return dist < max(player1.radius, player2.radius)


class Server:
    """
        TCP Connection

        This server handles all clients added to him automatically (threading)

    """

    def __init__(self, max_waiting, ip='0.0.0.0', port=0):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.bind((ip, port))
        self.socket.listen(max_waiting)
        print(f"Socket addr and port: {self.socket.getsockname()}")
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.threads = []


        self.player_update_handler = PlayerUpdateHandler()
        self.edible_update_handler = EdibleUpdateHandler()

        self.player_thread = dict()

        self.collision_detector = CollisionDetector()
        self.collision_detector_thread = threading.Thread(target=self.__handle_collisions, args=())
        self.collision_detector_thread.start()

        self.amount_of_clients = 0

    def __handle_client(self, client_socket, address, thread_id: int):
        # Send first message
        message = Protocol.server_initiate_world((world.width, world.height), world.edibles)
```

```
        send_with_size(client_socket, message)
        self.edible_update_handler.make_space_for_new_thread()
        self.collision_detector.players_eaten_helper.make_space_for_thread()

        should_continue = True
        # start getting status updates from the client
        while should_continue:
            message = recv_by_size(client_socket)  # recieve update
            player_information, edibles_eaten = Protocol.parse_client_status_update(message)

            self.player_thread[player_information.id] = thread_id  # for collision detection

            new_edibles = []
            for edible in edibles_eaten:
                new_edibles.append(world.delete_edible(edible))

            lock.acquire()

            self.edible_update_handler.notify_threads_changing_edible_status(new_edibles,
edibles_eaten, thread_id)
            self.player_update_handler.update_player(player_information)
            other_player_information = self.player_update_handler.get_players(player_information)
            edibles_removed, new_edibles_other =
self.edible_update_handler.fetch_thread_specific_edible_updates(
                thread_id)

            player_eaten_inf = self.collision_detector.players_eaten_helper.get_eaten_status(thread_id)
            rad_increase = player_eaten_inf.get_ate_radius()

            is_eaten = player_eaten_inf.get_killed()
            lock.release()
            if is_eaten:
                should_continue = False
            new_edibles = new_edibles + new_edibles_other
            send_with_size(client_socket, Protocol.generate_server_status_update(new_edibles,
other_player_information,
                                                edibles_removed, rad_increase, is_eaten))
```

```
    # close resources
    lock.acquire() # one last time :/
    self.player_update_handler.remove_player(player_information.id)
    lock.release()

def __handle_collisions(self):
    saved_collisions = set()
    while True:
        lock.acquire()
        players = self.player_update_handler.get_players()
        collisions = self.__detect_collisions(list(players.values()))
        for collision in collisions:
            if (collision[0].id, collision[1].id) not in saved_collisions:
                if collision[0].radius > collision[1].radius:
                    eating_thread_id = self.player_thread[collision[0].id]
                    eaten_thread_id = self.player_thread[collision[1].id]
                    self.collision_detector.players_eaten_helper.ate_player(eating_thread_id,
                                                                            players[collision[1].id].radius)
                    self.collision_detector.players_eaten_helper.player_killed(eaten_thread_id)

                    saved_collisions.add((collision[0].id, collision[1].id))
                else:
                    eating_thread_id = self.player_thread[collision[1].id]
                    eaten_thread_id = self.player_thread[collision[0].id]
                    self.collision_detector.players_eaten_helper.ate_player(eating_thread_id,
                                                                            players[collision[0].id].radius)
                    self.collision_detector.players_eaten_helper.player_killed(eaten_thread_id)
                    saved_collisions.add((collision[0].id, collision[1].id))
        lock.release()

def __detect_collisions(self, players):
    collisions = []
    for player_information in players:
        for collision_search in players:
            if not (isinstance(collision_search, str) or isinstance(player_information, str)):
                if player_information.id != collision_search.id and collision_exists(player_information,
```

```
                                                    collision_search) and
player_information.radius != collision_search.radius:
                    collisions.append((player_information, collision_search))
        return collisions

    """
        Accepts a new client (blocking)
    """

    def accept(self):
        client_socket, address = self.socket.accept()
        t = threading.Thread(target=self.__handle_client, args=(client_socket, address,
self.amount_of_clients))
        t.start()
        self.threads.append(t)
        self.amount_of_clients += 1


def start():
    global world
    world = World(PlatformConstants.PLATFORM_WIDTH,
PlatformConstants.PLATFORM_HEIGHT)
    world.spawn_edibles(EdibleConstants.AMOUNT_OF_EDIBLES)
    print("Accepting Clients....")
    server = Server(2)

    while True:
        # wait for new clients.
        server.accept()
```

collision_detector.py

```
"""
    Runs on seperate thread on the server

"""
from src.networking.helpers.players_eaten_helper import PlayersEatenHelper
from src.networking.information.player_information import PlayerInformation

class CollisionDetector:

    def __init__(self):
        self.players_eaten_helper : PlayersEatenHelper = PlayersEatenHelper()
```

edible_update_handler.py

```
"""
    Handles updating each thread about changing status of edibles
    Every thread has an index in the array from which it pulls the relevant edible information
"""
from src.networking.helpers.thread_update_helper import ThreadUpdateHelper

class EdibleUpdateHandler:

    def __init__(self):
        self.edible_updates : [ThreadUpdateHelper] = []

    def make_space_for_new_thread(self):
        self.edible_updates.append(ThreadUpdateHelper())
```

```
    """
        Adds to the indices of all OTHER threads that they should update the client about the edibles
    """
    def notify_threads_changing_edible_status(self, edibles_created, edibles_removed, thread_id):
        if edibles_created or edibles_removed:
            print(f"Notifying all other threads of edibles status, current thread: {thread_id}")
        for i in range(len(self.edible_updates)):
            if i != thread_id:
                self.edible_updates[i].update_edible_statuses(edibles_created, edibles_removed)

    """
        Fetches for the thread required information about the edibles created and removed by other
players
    """
    def fetch_thread_specific_edible_updates(self, thread_id):
        return self.edible_updates[thread_id].fetch_edibles_removed(),
self.edible_updates[thread_id].fetch_edibles_created()
```

player_update_handler.py

```
from src.networking.information.player_information import PlayerInformation

"""
    This class holds a dict that has the information about all the players in the map
"""


class PlayerUpdateHandler:

    def __init__(self):
        self.players_dict = dict()

    def update_player(self, player_information: PlayerInformation):
        self.players_dict[player_information.id] = player_information
```

```
    """
        Returns the information about all of the players except player given as parameter

        If nothing in dict returns -> []
    """

    def get_players(self, player_information: PlayerInformation = None):
        if player_information is None:
            return self.players_dict
        return [v for k, v in self.players_dict.items() if k != player_information.id and not isinstance(v,
str)]

    def remove_player(self, player_name):
        self.players_dict[player_name] = "KILLED"
```

game_protocol.py

```
from src.edible import Edible
from ast import literal_eval as make_tuple
from src.networking.information.player_information import PlayerInformation
LOG_PROTOCOL = False
EATEN = "EATEN"

class Protocol:
    """
        This method constructs a message that is to be sent to the client, requires information about the
players
        and the edibles.

        world_size - tuple of two (x,y)
        edibles: list of edible object


        Generates a protocol message that looks like:
        ~world_size_x,world_size_y~edible_x,edible_y,edible_color,edible_radius~
```

```
    for example:
    ~20000,20000~200,300,(2,3,4),5~.....~
    .... signifies - for each edible
"""

@staticmethod
def server_initiate_world(level_size, edibles: [Edible]):
    message = ''
    message += f'~{level_size[0]},{level_size[1]}~'
    for edible in edibles:
        tup = str(edible.color).replace(',', ':')
        message += f'{edible.platform_x},{edible.platform_y},{tup},{edible.radius}~'
    return message

"""
    Parses the message: server_initiate_world

    :returns -> tuple (x,y), tuple (edibles)
"""

@staticmethod
def parse_server_initiate_world(message: str):
    message_list = message.split('~')
    # message[1] = width,height
    world_size = (message_list[1].split(',')[0], message_list[1].split(',')[1])

    edible_message_unparsed = message[message[1::].find('~') + 1:]
    # now we have the edibles, parsing...
    edible_list_unparsed = edible_message_unparsed.split('~')
    edibles = []  # return list of edibles
    for edible in edible_list_unparsed:
        if edible != '':
            params = edible.split(',')
            edible_x = int(params[0])
            edible_y = int(params[1])
            color = make_tuple(params[2].replace(':', ','))
            radius = int(params[3])
```

```
            edibles.append(Edible(edible_x, edible_y, color, radius))

        return world_size, edibles

    """
        Updates the server of any changes in the playing field, including edibles being eaten

        message -  ~name,x,y,radius~...~
        ... - Send edible information if it was eaten, so server can remove it from the playing field
    """

    @staticmethod
    def generate_client_status_update(player_x, player_y, player_radius, name, id, edibles: [Edible] =
None):
        message = '~'
        message += f'{id},{name},{player_x},{player_y},{player_radius}~'

        if edibles is not None:
            for edible in edibles:
                tup = str(edible.color).replace(',', ':')
                message += f'{edible.platform_x},{edible.platform_y},{tup},{edible.radius}~'

        return message

    """
        A message has been recieved from a client in the format of the message sent in -
generate_client_status_update
        edible format (reminder) - x,y,color,radius
    """

    @staticmethod
    def parse_client_status_update(message: str):
        if LOG_PROTOCOL:
            print(f"Client sent message: {message}")
        message_list = message.split('~')

        # location of player in world coordinates
```

```
    player_information = message_list[1].split(',')
    player_uuid = player_information[0]
    player_name = player_information[1]
    player_location = float(player_information[2]), float(player_information[3])
    player_radius = float(player_information[4])

    eaten_edibles = []
    # update the server if any edibles were eaten
    if message_list[2] != '':
        # + 1 because we need to account for starting delimeter
        edibles_list_unparsed = message[message[1::].find('~') + 1 + 1:].split('~')
        for edible in edibles_list_unparsed:
            if edible != '':
                params = edible.split(',')
                edible_x = int(params[0])
                edible_y = int(params[1])
                color = make_tuple(params[2].replace(':', ','))
                radius = int(params[3])
                eaten_edibles.append(Edible(edible_x, edible_y, color, radius))
    return PlayerInformation(player_location[0], player_location[1], player_radius, player_name,
player_uuid), eaten_edibles



    """
    Message sent to client to update him of everything he needs to know (changes in world, player
positions)

    special seperator for this type of message - ~!"# -> signifies transition between data
    message -> [size]~5,5,(2,2,2),6~...~!"#6,5,4...~23,45,(..),5~!"....~
    """
    @staticmethod
    def generate_server_status_update(edibles_created: [Edible], other_player_information :
[PlayerInformation], edibles_removed: [Edible], eaten_rad_increase, is_eaten):
    if is_eaten:
        return EATEN
```

```
        message = f'~'

        for edible in edibles_created:
            tup = str(edible.color).replace(',', ':')
            message += f'{edible.platform_x},{edible.platform_y},{tup},{edible.radius}~'
        # added edibles created
        message += '!"#'
        for player_information in other_player_information:
            message +=
f'{player_information.name},{player_information.x},{player_information.y},{player_information.rad
ius}~'
        message += '!"#'

        for edible_removed in edibles_removed:
            tup = str(edible_removed.color).replace(',', ':')
            message +=
f'{edible_removed.platform_x},{edible_removed.platform_y},{tup},{edible_removed.radius}~'


        return message + '!"#' + str(eaten_rad_increase)


    @staticmethod
    def parse_server_status_update(message: str):
        if message.strip() == "EATEN":
            return "EATEN"

        edibles_created_unparsed = message.split('!"#')[0]
        # now we have: ~5,5,(2,2,2),6~...~
        edibles_created_list = edibles_created_unparsed.split('~')
        edibles_created = []
        for edible_created in edibles_created_list:
            if edible_created != '':
                params = edible_created.split(',')
                edible_x = int(params[0])
                edible_y = int(params[1])
                color = make_tuple(params[2].replace(':', ','))
```

```
            radius = int(params[3])
            edibles_created.append(Edible(edible_x, edible_y, color, radius))
        player_information_unparsed = message.split('!"#')[1]
        player_information_list = player_information_unparsed.split('~')
        player_information_parsed : [PlayerInformation] = []
        for information in player_information_list:
            if information != '':
                information_params = information.split(',')
                player_name = information_params[0]
                player_x, player_y = int(float(information_params[1])), int(float(information_params[2]))
                player_radius = int(float(information_params[3]))

player_information_parsed.append(PlayerInformation(player_x,player_y,player_radius,player_name))


        edibles_removed_list = message.split('!"#')[2].split('~')

        edibles_removed : [Edible] = []

        for edible_removed in edibles_removed_list:
            if edible_removed != '':
                params = edible_removed.split(',')
                edible_x = int(params[0])
                edible_y = int(params[1])
                color = make_tuple(params[2].replace(':', ','))
                radius = int(params[3])
                edibles_removed.append(Edible(edible_x, edible_y, color, radius))

        radius_inc = (message.split('!"#')[3])
        radius_inc = float(radius_inc)
        return edibles_created, player_information_parsed, edibles_removed, radius_inc
```

players_eaten_helper.py

```
from src.networking.information.players_eaten_information import PlayersEatenInformation


"""
    Class is used by collision detector to store info for each thread
"""
class PlayersEatenHelper:

    def __init__(self):
        self.handler_list: [PlayersEatenInformation] = []

    def make_space_for_thread(self):
        self.handler_list.append(PlayersEatenInformation())

    def ate_player(self, thread_id, eaten_radius):
        self.handler_list[thread_id].ate_player(eaten_radius)

    def player_killed(self, thread_id):
        self.handler_list[thread_id].killed()

    def get_eaten_status(self, thread_id):
```

```
        return self.handler_list[thread_id]
```

thread_update_helper.py

```
"""
    Stores reliable information for thread to use.
    used by: edible_update_handler
"""


class ThreadUpdateHelper:

    def __init__(self):
        self.edibles_created = []
        self.edibles_removed = []

    def update_edibles_removed(self, edibles_removed):
        for edible in edibles_removed:
            self.edibles_removed.append(edible)

    def update_edibles_created(self, edibles_created):
        for edible in edibles_created:
            self.edibles_created.append(edible)

    def update_edible_statuses(self, edibles_created, edibles_removed):
        self.update_edibles_removed(edibles_removed)
        self.update_edibles_created(edibles_created)

    """
        To be used by the thread, will clear the list
    """
    def fetch_edibles_removed(self):
        edibles_removed = self.edibles_removed.copy()
        self.edibles_removed.clear()
```

```
        return edibles_removed


    def fetch_edibles_created(self):
        edibles_created = self.edibles_created.copy()
        self.edibles_created.clear()
        return edibles_created
```

utils.py

```
import socket, struct

DELIMETER = "~"
SIZE_HEADER_FORMAT = "00000000~"  # n digits for data size + one delimiter
size_header_size = len(SIZE_HEADER_FORMAT)
TCP_DEBUG = False


def recv_by_size(sock, return_type="string"):
    str_size = b""
    data_len = 0
    while len(str_size) < size_header_size:
        _d = sock.recv(size_header_size - len(str_size))
        if len(_d) == 0:
            str_size = b""
            break
        str_size += _d
    data = b""
    str_size = str_size.decode()
    if str_size != "":
        data_len = int(str_size[:size_header_size - 1])
        while len(data) < data_len:
            _d = sock.recv(data_len - len(data))
            if len(_d) == 0:
                data = b""
                break
            data += _d
```

```
    if TCP_DEBUG and len(str_size) > 0:
        data_to_print = data[:100]
        if type(data_to_print) == bytes:
            try:
                data_to_print = data_to_print.decode()
            except (UnicodeDecodeError, AttributeError):
                pass
        print(f"\nReceive({str_size})>>>{data_to_print}")

    if data_len != len(data):
        data = b""  # Partial data is like no data !
    if return_type == "string":
        return data.decode()
    return data



def send_with_size(sock, data):
    len_data = len(data)
    len_data = str(len(data)).zfill(size_header_size - 1) + "~"
    len_data = len_data.encode()
    if type(data) != bytes:
        data = data.encode()
    data = len_data + data
    sock.send(data)

    if TCP_DEBUG and len(len_data) > 0:
        data = data[:100]
        if type(data) == bytes:
            try:
                data = data.decode()
            except (UnicodeDecodeError, AttributeError):
                pass
        print(f"\nSent({len_data})>>>{data}")
```

players_eaten_information.py

```
import math


class PlayersEatenInformation:

    def __init__(self):
        self.is_eaten = False
        self.ate_radius = 0

    """
        Set radius increase to the radius the player should be after he ate another player.

        s_a(r1,r2) = r1**2*pi + r2**2*pi
        A = s_a(r1,r2)
        new_r = sqrt(A/pi)
    """

    def ate_player(self, player_radius):
        sum_area = (player_radius ** 2 * math.pi) + (self.ate_radius ** 2 * math.pi)
        self.ate_radius = (sum_area / math.pi) ** 0.5

    """
        Resets
    """

    def get_ate_radius(self):
        rad = self.ate_radius
        self.ate_radius = 0
        return rad

    def killed(self):
        self.is_eaten = True
    def get_killed(self):
        return self.is_eaten
```

player_information.py

```
class PlayerInformation:
    def __init__(self, x, y, radius, name, id=""):
        self.x = x
        self.y = y
        self.radius = radius
        self.name = name
        self.id = id


    def set_information(self, x, y, radius):
        self.x = x
        self.y = y
        self.radius = radius

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y and self.radius == other.radius and self.name ==
other.name
```

agar.py

```
import src.networking.client as client
import src.networking.server as server
import pygame
import pygame_menu
pygame.init()
"""
    Runs main.
"""
```

```
WIDTH = 1920
HEIGHT = 1080
THEME = pygame_menu.themes.THEME_BLUE
window = None

def set_ip_and_port():
    client_menu = pygame_menu.menu.Menu("Connect", WIDTH, HEIGHT, theme=THEME)
    input_args = {"font_size":80}
    ip = client_menu.add.text_input("IP: ", default="0.0.0.0", **input_args)
    port = client_menu.add.text_input("Port: ", default=0, **input_args)
    name = client_menu.add.text_input("Enter Name: ", default="Johnny", **input_args)
    client_menu.add.vertical_margin(100)
    client_menu.add.button("Connect!", start_client, *(name, ip, port))
    client_menu.mainloop(window)

def main_menu():
    main_menu = pygame_menu.menu.Menu("agar.py!", 1920, 1080, theme=THEME)
    main_menu.add.button("Play", set_ip_and_port).scale(4, 4, False)
    main_menu.add.button("Host", start_server).scale(4, 4, False)
    main_menu.add.button("Quit", pygame.quit).scale(4, 4, False)
    main_menu.mainloop(window)

def start_server():
    pygame.quit()
    server.start()

def start_client(name, ip, port):
    try:
        client.start(name.get_value(), ip.get_value().strip(), int(port.get_value().strip()), window)
    except:
        print("Invalid Parameters!")
        set_ip_and_port()
if __name__ == '__main__':
    window = pygame.display.set_mode((1920, 1080))
    main_menu()
```