VRIJE UNIVERSITEIT BRUSSEL

Project Computersystemen

Brick Breaker
Group G19

Artyom Petrosyan & Kyan David

December 22, 2023

Academic year 2023-2024

EIT

1 Introduction

In the realm of classic arcade games, the Brick Breaker game is one of the most popular. The game offers a combination of skill and strategy. We developed the 80386 Brick Breaker game to explore the world of the 80386 32-bit assembly environment. Our aim was to develop an interactable and fun game. The first level of the game is shown in the screenshot below.

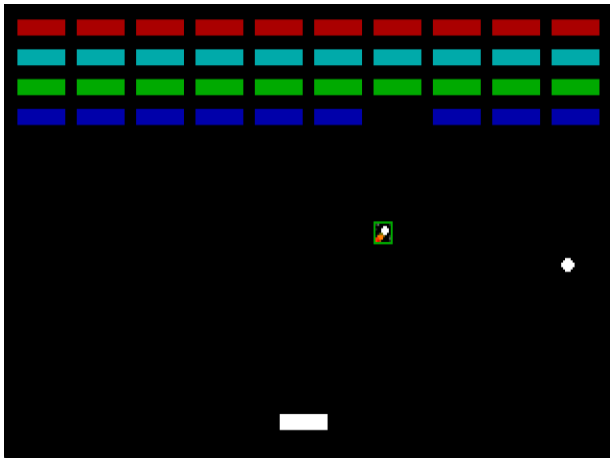

Figure 1-1: Example screenshot level 1

Our project consists of two primary objectives. The first objective was to create a playable and interactive game using the 80386 architecture. Using assembly, we were able to create a smooth and responsive gameplay and enhance user experience. The second objective was to make the game more dynamic by implementing different levels and powerups while keeping the interface simple and accessible.

The game rules lie very close to the classic Brick Breaker game. The player has to control a paddles horizontal movement to not let the ball fall and try to break every brick above. By breaking bricks, there is a chance to get powerups which will give the player an advantage.

2 Manual

Instead of using the arrow keys, the game can be operated using ‘s’ or ‘d’ to move left and right respectively. To go up or down in the GUI, the ‘r’ and ‘f’ can be used respectively. The idea of the game is to not let the ball fall and to try to break every brick. When the ball falls, the game ends and goes back to the menu, where the level can be restarted, or another level can be chosen.

3 Program features

The game consists of interactable object, e.g. the paddle, to add a skill element to the gameplay. Each time a brick gets hit by the ball, the brick loses health points, which are represented by the color index of the standard palette of the 13h video mode. When breaking a brick there is a 33% chance that a powerup or a powerdown will fall. By catching the powerups the object properties, e.g. the ball speed, will update accordingly. To make the gameplay funnier we implemented a continuous rebound angle depending on how the ball is caught.

To make the game more graphically pleasing, we made use of sprites to display images. The interactable menu is made from self-made images, which gives the freedom to make a custom an easily implementation in the code. The powerups are also displayed by self-made sprites to clearly show which powerup or powerdown can be caught. The powerups wear off when the timer of the powerups ends. The game consists of three different levels that can be freely chosen from the menu. Each level consists of a different structure to show the freedom of designing the levels. In the figures below can see the GUI and the powerups. Sound is an important part of games to not make the game boring and monotone. We decided to implement separate music for the menu and each level, which will give the player an enjoyable gaming experience.

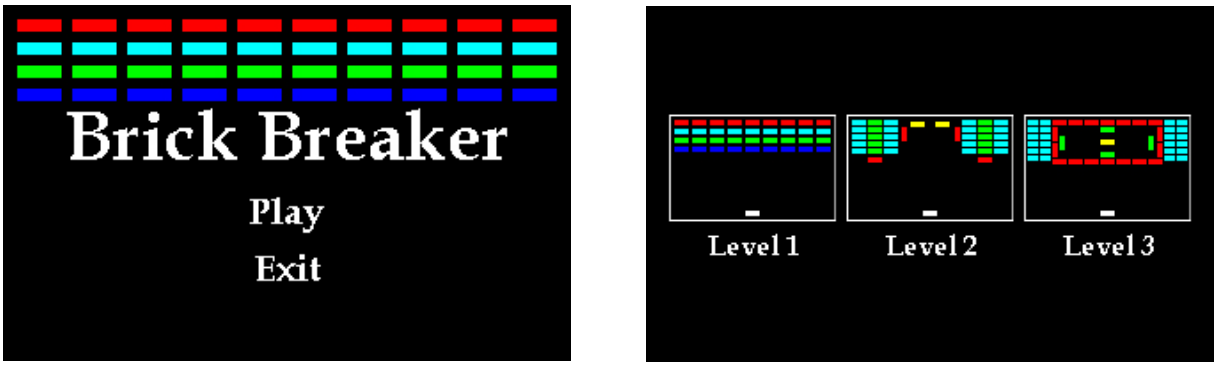

Figure 3-1: GUI

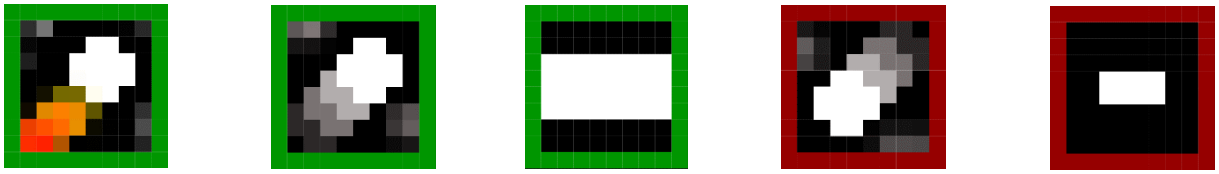

Figure 3-2: Powerups and Powerdowns

While the game has all these features, it still has its limitations. For example, the 80386 architecture has certain constraints on graphics, which limits us to a small resolution. When performing extensive tasks, such as looping through an array to check a condition, the player will experience less smoothness. When reading large files into memory, the program is subjected to delays, which will decrease the overall performance. To optimize the game, we used a technique of activity of certain objects. This means that each object has a value 0, if inactive, and a value 1, if active. Before looping over elements in an array, the program first checks to see if the element that needs to be updated is active or not. If the element is inactive, the program jumps to the next element without performing the update function to the inactive element. This way, only the active and useful elements will undergo the update. Drawing the objects every time over and over again, can also cause some performance issues. To fix that problem, the program just redraws the moving part with the background color, in this case black, to remove the object. Then after the object is removed, the program redraws the object at the updated position to simulate movement. This way, the performance will improve, and the game will run more smoothly.

4 Program design

4.1 Videomodes:

There are numerous video modes that could be used. We chose video mode 13h, primarily based on the simple implementation of the color palate. In the game, there will be different bricks which will be distinguished based on their color. Therefore, it is necessary to use a good color palate. We use a procedure to set the video mode, which could also be used to set other video modes than 13h. The default color palette of the 13h video mode is shown in the figure below.

0 Black		8 Dark grey	
1 Blue		9 Light blue	
2 Green		10 Light green	
3 Cyan		11 Light Cyan	
4 Red		12 Light Red	
5 Magenta		13 Light magenta	
6 Brown		14 Yellow	
7 Light grey		15 White	

Figure 4-1: VGA color palette

4.2 Reading and initializing data:

To initialize the data, we used binary files which are generated using a higher-level programming language, in this case MATLAB. With a ReadFile procedure, it is possible to not only read data files, but also images. When starting the program, the user will see an interactable GUI, where a level can be selected. When selecting a level, all the files that are needed to initialize the game are read and loaded into arrays of structs with a predefined size.

4.3 Structs:

For implementation of the bricks, ball and powerups we used structs to make the code more user friendly and readable. At first, we used an array to implement the data, such as the position. The array approach did not make the implementation easier, which made us switch to structs. Structs have an analogy to the higher-level programming objects.

4.4 GUI:

To make starting the game easy and simple, we designed a GUI to do so. When starting the game, the GUI will appear. This part is hardcoded, because it is a very specific implementation that does not need to be general. By pressing different keys, the player can navigate freely through the GUI. In the GUI, the player can choose to play the different levels or exit the program. When choosing a level, the program will initialize the chosen level and the level music. The level will be overwritten when the player loses and starts another level, so there will not be any trace of the previous level, each level is unique.

4.5 Sound

We reused the playwav file from the badapple example to implement music in the game. To make it a bit more fun we adjusted the file to make it possible to give an argument to the startmusic procedure and thus play multiple music files. When exiting the level, we need to stop the corresponding music and start the menu music. The needed audio file is a headerless WAV-file with 8-bit PCM and 22050Hz sampling rate. It is important to use “unsigned” 8-bit raw files as signed will not result in the same quality.

4.6 Notion of time

To animate the game, we need a certain notion of time. In x86 assembly you can access the system time. The 21h interrupt is used with ah = 2Ch. This interruption returns the hours, the minutes, the seconds and the 100th of seconds in the ch, cl, dh and dl registers respectively. It is important to note that most of the clocks associated with x86 processors have a resolution of only 500th of a second. Immediately you can understand that this can result in problems of inconsistency. This was confirmed as we were informed that it was not a reliable way of updating the game. We then decided to use the "wait_VBlank" procedure that was accessible. It is a procedure that waits for a certain number of frames.

4.7 Movement

In this game there are three "objects" with a motion: the ball, the paddle and the powerups. They all have the same mechanic. They have an intrinsic speed and every time the motion updates their coordinates update accordingly. By adding the speed value to the coordinates.

The movements need to be restricted by the boundaries of the screen. This is done by comparing the value associated to the position of the object and the screen boundaries.

To adjust the speed of the objects more precisely we scale up all the coordinates initially, for calculation and other, and only scale them down for drawing.

4.8 Collision

For collision detection we will loop over the objects that can collide and check for collision with the Axis Aligned Bounded Box (AABB) method. So for collision with the ball, we will loop over all the bricks as they are the only ones that can collide with the ball. And we will make use of the AABB method. In short this is a method where you make an AABB just big enough to encompass the shape. The space occupied by this AABB can be defined by 2 points (in 2D: $(minx, miny)_1$ and $(maxx, maxy)_2$. This characteristic can be used to quickly check as to whether two AABBs intersect. Consider the two AABBs in the diagram below:

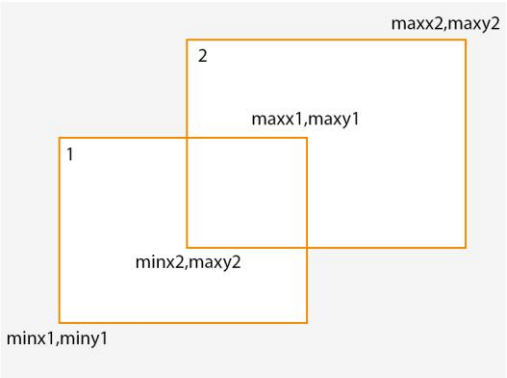

Figure 4-2:AABB

"In Figure 4-3 we have two AABB's defined by a pair of points. Using the following expression one can determine if the two AABB's intersect.

maxx1 > minx2 && minx1 < maxx2 && maxy1 > miny1 && miny1 < maxy2

It is important to notice that this expression is composed of exclusively and's which means that if one of the conditions fails there won't be any collision." (James, 2017)

4.9 Bricks:

The "level_bricks" array can be divided into two different parts: the paddle and the bricks.

The paddle is the first element of the array, and it is peculiar because it is the only object that moves following inputs of the player. The Y-coordinate of the paddle is constant and only the X-coordinate is updated with the general method. The challenge lies in the input. We need to move to the paddle to the right and the left according to the corresponding key, 'd' and 's' respectively. We also check for capital letters for more reliability. If the paddle exceeds the boundaries, the coordinates are adjusted so that it only moves to the maximum or minimum value. The paddle also has an additional check after moving, if the ball moves in the paddle when moving the paddle, the ball's position will be adjusted such that the ball does not jump into the paddle.

There are two different types of bricks: stationary and moving bricks. The stationary bricks will just be drawn at the given coordinates with a given size in a specific color. Their x- and y-velocities are equal to zero, which will not result in any movement.

For the movement of the bricks, during each frame, the program calls the "updateBrick" procedure, that takes a type and a brick index as arguments, with type 1 as argument. In this case, it does not matter which brick index the procedure receives as argument, because when the type equals 1, the procedure jumps over the part where the brick index is needed. In this procedure, the brick positions will update by adding the x- and y-velocities of the brick to the x- and y-coordinates. The moving bricks get a rectangular boundary, which they cannot exit. When they hit a threshold, the corresponding velocity will negate. For this to work, the program needs to loop over every brick to move all the needed bricks. The bricks can only break when the ball collides with it. Therefore, we loop over every brick when the ball moves. If there is collision (AABB) with another brick than the paddle we call the "updateBrick" procedure that updates the collided brick to subtract health points; according to the damage that the ball can deliver, and thus change the color.

4.10 Ball:

The ball has some extra condition to the original movement, namely the collisions. The ball can collide with the border and the bricks. As the principle for checking collision was explained earlier. In the next two paragraphs we will focus on the reaction of the ball after colliding.

4.10.1 Collision with the border

The collision with the border is probably the easiest to implement as none of the borders move. We compare the ball position with the window dimensions. If you exceed the borders in any way the ball speed is negated in such a way that it bounces off. So, for example, the horizontal velocity (x_Vel) is negated when colliding with the left or right border. The same goes for the upper border. And for the lower border we do not negate because when the ball hits the ground the game ends.

4.10.2 Collision with a brick

The ball can collide with the bricks and the paddle. When colliding with a brick it needs to adjust its velocity so that it will bounce back. If it hits the side (left or right), it negates the x-velocity and if it hits the bottom or top it negates the y-velocity. To compute which border gets hits, we look with what the "line in the movement's direction" (called "velocity vector" from now on) intersects. Because the ball is already colliding, we know that if the y-velocity is negative (ball going upwards), the ball cannot hit the top. At this point we look whether the velocity vector intersects with the bottom. If there is collision with the bottom, then we negate the y-velocity, otherwise we know it hits a side because it doesn't hit the top and negate the x-velocity. The same process is valid if the y-velocity is positive, ball going downwards. This method is thorough and easily reusable if more features need to be added concerning the collision direction.

To check the intersection between segments, we use a line segment intersection algorithm found on the internet that we will call the Counterclockwise method (CCW method). To sum up briefly, it is a method where a base line segment is compared with the 2 points of another segment separately. We can determine if a point is oriented clockwise or counterclockwise relative to a segment. If the two points have the same orientation, then there is no intersection as illustrated here under in Figure 4.10.2, if they have not, then we need to check the orientation of the original base segment relative to the other segment. If again, they do not have the same orientation, there is an intersection. The actual implementation of the CCW method does not handle collinearity, not proper intersection, as it is used so that collinearity does not happen. We choose the length of the velocity vector so that none of the beginning- and endpoint can be on the border of the brick and thus we always have proper intersection or none. For further information we advise checking the references.

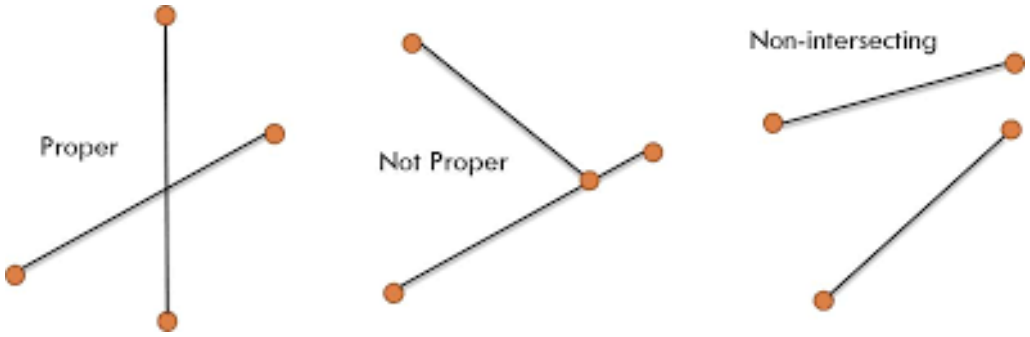

Figure 4-3:Intersection

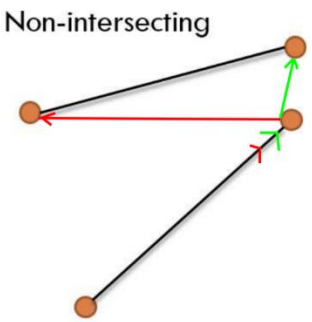

Figure 4-4: CCW illustrated on Non-Intersecting line segments

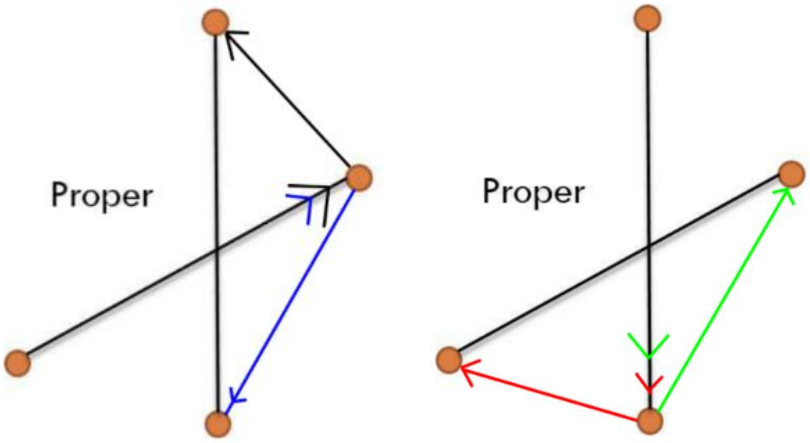

Figure 4-5:CCW illustrated on proper intersecting line segments

4.10.3 Gradient (FPU)

The general collision with the bricks is implemented. To make the gameplay more interesting and controllable we wanted a "gradient" determining the exit angle depending on the contact point with the paddle. To make the gradient more uniform we decided to implement this using the Floating-Point Unit (FPU).

We want an angle in function of contact position relative to the center of the paddle. It seems evident that we will need to take the inverse of a trigonometric function. The angle needs to be zero at relative distance (d_r), that we define ball_X_middle – Paddle_X_mid, equaling 0. It is important to note that the position values refer to the top left corner of the object, hence the fact that we use X_middle. To achieve the angle at 0 for d_r, we use the arcsine function. The arcsine takes a value between -1 and 1 therefore we must normalize the d_r. We also want a maximal rebound angle of only 45° and the image of the arcsine function is $[-\pi/2, \pi/2]$ so we divide the angle by 2 to make it match with our need. Then to apply this exit angle to our speed we multiply cos (angle) to the y-velocity and sin(angle) to the x-velocity.

4.11 Powerups:

A powerup of powerdown will be randomly given when a brick breaks. To make the powerups random, we made use of two pseudo random number generators. The first one gives a number between 0 and 2, and when the number equals 0 for example, which means a 33% chance, the game gives the player a powerup or powerdown. The second pseudo random number generator gives a number between 0 and Powerup_Amount – 1, which selects which powerup the player gets. Note that the Powerup_Amount variable is not equal to 5, the different types of powerups, but equals to the amount of powerups in the powerups array. We initialized each powerup multiple times to make sure that for example, multiple fastball powerups could be active at the same time.

For the powerups, we only use a vertical velocity component, because the powerup does not need to go left or right. By breaking a brick, the powerup gets the brick coordinates and is set to active. For each powerup, we designed a logo that will be displayed as a sprite, which are displayed in Figure 3-2. When hit, it will be covered up by a black rectangle. The powerup logos are represented in the figures below.

Each powerup has multiplicators which will, for example, multiply the ball speed with a given value to make the ball faster or slower. When a powerup does not get caught and just falls out of the frame it gets deactivated and can be randomly selected again. But when the powerup is caught, the data of the ball and the paddle will be modified accordingly to which powerup is caught and a timer of one minute will start. When the timer ends, the powerups will be deactivated and the data will reset to their original values.

Different powerups can get activated at the same time, each powerup has its own timer. After the timer ends, the paddle and ball will go back to its original state. When powerups are caught they can add on the already active powerup, for example when the paddle is already bigger due to the "Bigpad" powerup and the player catches a "Smallpad" powerup, the paddle will just go to its original state. The powerups can cancel each other out.

5 Encountered problems

One problem that we encountered was the initialization of the data segment. To read the files, there had to be a dataread that was uninitiated. We put all the datareads at the end of the datasegment, which in its turn caused problems. We got illegal reads and writes when running the code for a minute. To solve this problem, we simply had to put the uninitialized dataread before the files in the data segment.

Another problem that was hard to find was the naming of the files. When trying to read the files with its original name, we got the error that the file could not be opened. The solution was shortening the file name to less than 8 characters to be able to read the file.

A problem that we didn't have time to solve was the gradient. As you can see the ball's rebound angle isn't the same if the ball hits the paddle from the right or the left. We tried to modify the "gradientdistance" but the problem remained. A possible fix would be to use the arctangent instead or half of the arcsine as the angle. This solution occurred to us while writing this report. We tried to implement this but due to lack of time we couldn't be able to successfully solve that problem.

We also had a problem including the "playwav" file. The problem was that the playwav file needs to be included in the file itself.

Another feature we tried to implement, was to play a sound when a certain action is performed, like a "pong" sound when the ball collides with the bricks. The problem we encountered was that in the "playwav" example file, the applied method is auto initialize DMA, meaning that the file will be repeated until it is stopped in the code. There is another method of Single Cycle DMA that is more applicable to the feature as it stops after only one play. We didn't implement this due to lack of time.

6 Conclusion

In summary, the development of the 80386 Brick Breaker game successfully delivered a playable and interactive experience with dynamic levels and engaging powerups. The game design featured an intuitive GUI, smooth gameplay, and visually appealing sprites. Despite the inherent limitations of the 80386 architecture, optimization techniques were implemented to ensure a responsive gaming experience.

The program design demonstrated a structured approach, utilizing video mode 13h, binary file data initialization, and the effective use of structs for object implementation. The game's unique features, such as distinct powerups, varied levels, and carefully crafted collision mechanics, contributed to an enjoyable and challenging gaming experience.

While certain challenges were encountered and addressed during development, such as data segment initialization and file naming issues, others, like refining the ball rebound gradient, remained unresolved due to time constraints.

In conclusion, the 80386 Brick Breaker game project successfully highlighted the creative and technical possibilities within the chosen assembly environment. The combination of classic gameplay elements and innovative features showcased the potential for future enhancements, emphasizing the project's success in achieving its primary objectives.

7 References

[1] *3D collision detection*. (n.d.). MDN Web Docs. https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection

[2] *Assembler taylor series arctg x*. (2014, January). Stack Overflow. https://stackoverflow.com/questions/20960987/assembler-taylor-series-arctg-x

[3] *Assembler: Procedures*. (n.d.). Retrieved December 24, 2023, from http://www.sci.brooklyn.cuny.edu/~jones/cisc3310/Assembler%20files/Assembler%20procedures.pdf

[4] Burt, G. (2004). *Structures in Assembly*. Redirect.cs.umbc.edu. https://redirect.cs.umbc.edu/courses/undergraduate/313/spring05/burt_katz/lectures/Lect10/structuresInAsm.html

[5] *IMUL — Signed Multiply*. (n.d.). Www.felixcloutier.com. https://www.felixcloutier.com/x86/imul

[6] James. (2017, January 15). *Introductory Guide to AABB Tree Collision Detection – Azure From The Trenches*. Www.azurefromthetrenches.com. https://www.azurefromthetrenches.com/introductory-guide-to-aabb-tree-collision- detection/

[7] *Procedures*. (n.d.). Www.c-Jump.com. Retrieved December 24, 2023, from http://www.c-jump.com/CIS77/ASM/Procedures/lecture.html#P77_0010_call_ret

[8] *python - How can I check if two segments intersect?* (2010, October). Stack Overflow. https://stackoverflow.com/questions/3838329/how-can-i-check-if-two-segments-intersect

[9] Sedory, D. (2013). *The 8086 Registers; Guide to MS-DEBUG*. Pcministry.com. https://thestarman.pcministry.com/asm/debug/8086REGs.htm

[10] *x86 Assembly/Floating Point - Wikibooks, open books for an open world*. (2023, October 23). En.wikibooks.org. https://en.wikibooks.org/wiki/X86_Assembly/Floating_Point