INF-1100: Introduction to programing and computer behavior

Assignment 3 report

Amund H. Strøm

November 12, 2021

Introduction

In this assignment we were tasked to write a program that animated 5 balls bouncing on a screen. We got a lot of pre-written code to help us complete this, but our task was to develop a linked list to keep track of all the different balls, create an object data structure to represent the balls, implement an animation loop, and write an algorithm for the balls to bounce on the screen. In this report I will be talking about how I managed to complete this task, some things I found interesting and some problems I encountered on this path.

Technical background

There were a lot of new concepts we had to learn in this assignment, some harder to understand than other. The main concept was the use of linked list, but before we talk about that, we need to understand some basic knowledge that linked lists builds on. That is the use of structs and dynamic memory allocation.

Structs (short for structures) is a data type that is defined by the user and can contain different data types (such as integers, characters, floats, etc.) under a single name. This is a great tool to group together data types that are related to each other.

Dynamic memory allocation is the procedure to change the size of a data structure (like array or linked list) during runtime. This means that we can change how much space in the memory something is using during the time the program is running. Since C is a structured programing

language, this is something that must be done and cannot be ignored. If not done correctly or not at all, we will have something call a memory leak. A memory leak means that the program is not freeing the memory correctly and will just keep taking more and more space in the memory, eventually it will slow the computer down or even halt it completely. In other programing languages this is done automatically, but the benefit C has is that you can perfectly manage the memory and in some cases a program runs much smoother in C than any other languages.

Now that we understand structs and dynamic memory allocation, we can talk about linked lists. Linked lists are like an array, a linear data structure, but have the benefit to add or remove specific elements inside of the list without recreating the whole list again. This is because linked list is not stored at a contiguous location, but each element is linked using pointers. To represent a linked list, we use structures to create nodes for the list, and inside of a node there is at least data and a pointer to the next node. To access the linked list, you need a pointer to the first node in the list, which is called the head. And if the head is empty, the value of the node is NULL. This also means that if there are two or more nodes, the last node must always point to NULL, so the program understands that this is the last node. If we would illustrate it, it would look something like this.

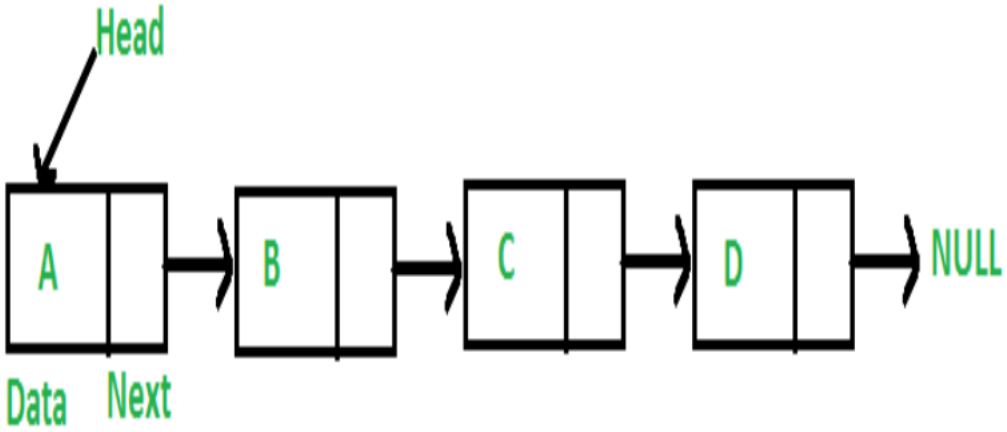


Figure 1: https://www.geeksforgeeks.org/linked-list-set-1-introduction/

For every time you want to create a new list or add a new node to a given list, you need to allocate memory for this to happen. And the same is done for every time you want to remove a list or node, then you need to free the memory of the list or node.

Now this begs the question, when do we use lists over arrays? Arrays are good for when you want to store linear data of similar types, but there are some limitations. The size of the array cannot be changed during runtime, so we must know in advance the upper limits of the array.

Also, if you want to add an element in the middle of an array, room must be created for this element. This means that every element after the spot you want your element to be, must be shifted. The same logic is applied when you want to remove an element in the middle. Now what are the advantages lists have over arrays? Unlike arrays, list is good for when you want to store linear data of different types. They also have dynamic size, which means you can change the size of the list during runtime, this is useful for when you know that the size of the list is going to change a lot. It is way easier to add or remove elements, you just simply direct the pointer for a node to a new location. The drawbacks of lists are that there is no random access, since you always need to go in chronological order. And that the pointer in a node takes up extra space in the memory, compared to arrays which don't have any pointer between elements.

Implementation

In this assignment we were tasked to write a linked list, create an object data structure, implement an animation loop, write an algorithm for the balls to bounce off screen edges, and remove the balls when they are still. I will be talking about how I solved these in the order that I did them in.

Linked lists

There were several functions that was needed to make a list that would work for our purpose. First, we need to create the list and give it space in the memory. The list we created was also supposed to be empty at first, and point directly to NULL, meaning that this is the only node in the list.
The second function is used to destroy or free the list, so it wouldn't take up space in the memory.
The third function is used to add a new node with an item to the start of the given list. To do so, we first need to create space in the memory for this node. Then we want the pointer for this node to point at the current head of the list, and then change the current head of the list to point at this node. Now we have effectively changed the head of the list, and it is important to do so in this specific order, otherwise things will go wrong. Now we set the item in this node equal to the item we got as an argument, and increase number of items by 1, so we can keep track off how many items we have in the list.

The fourth function is used to add an item last in the list. This is done somewhat like adding an item first in the list. But instead of changing the head pointer, we are changing the NULL pointer. First, we need to make space in the memory for this node. Then we want to find the NULL pointer, to do so we need to iterate through the list, till we find the pointer that points to NULL. We then change this node to point to the new last node and tell the new last node to point to NULL. Afterward we set the item in this node equal to the item we got as an argument and increase the number of items by 1.

The fifth function is used to remove a given item form the given list. This means that we want to remove the item we got as an argument in the list that it belongs to. To do so we iterate through the list till we meet the matching item. When we eventually meet this item, we change the pointer for the past node to point to the node after the node that contains the matching item. To illustrate this, it will look something like this.

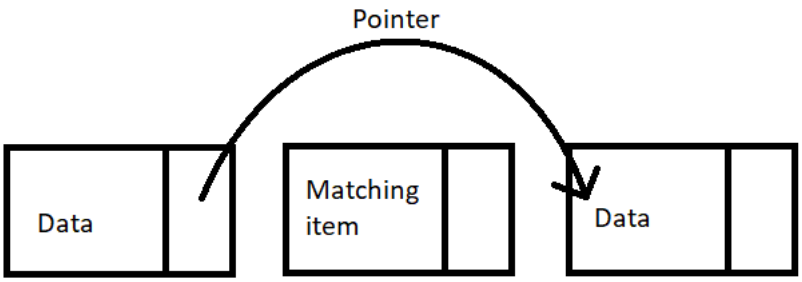


Figure 2: By Amund Strøm

We then free or destroy the node with the matching time, so it won't take up space in the memory, and decrease the number of items in the list by 1. But to make this function work properly, there are three different cases that we need to solve. That is when the item is at the start of the list, if the item is at the end of the list, and if the item is not at the end of the list. We simply write an if statement before we iterate through the list to check if the item is at the start, and when we iterate, we check if the item is the last node, if not, it will be somewhere in the middle.

The sixth is simply used to tell the size of the given list. This is easily done since we have counted every time we have added and removed an item form the list. So, we just return the number of items in the list.

Now for the seventh function, here we will create a list iterator, we do this so we can more effectively look at the items the given list contains. To do so we need to create space in the memory, set the iterator equal to the list, so they will have the same content, and then set the next node for the iterator equal to the head of the original list so the iterator will know where to look.

The eight function is used to free or destroy the list iterator, so it won't take up space in the memory.
The ninth functions is used to move the iterator to the next item in the list and return the current item, until it reaches NULL. We do this by creating a void item that can hold all variables and set that equal to the current item in the iterator, then we move the iterator to the next node.
The tenth and final function in the list implementation is used to reset the iterator, so it will point to the start if the list again. We do this by telling the iterator to point to head, and by doing so, we can iterate through the list again from start to end.

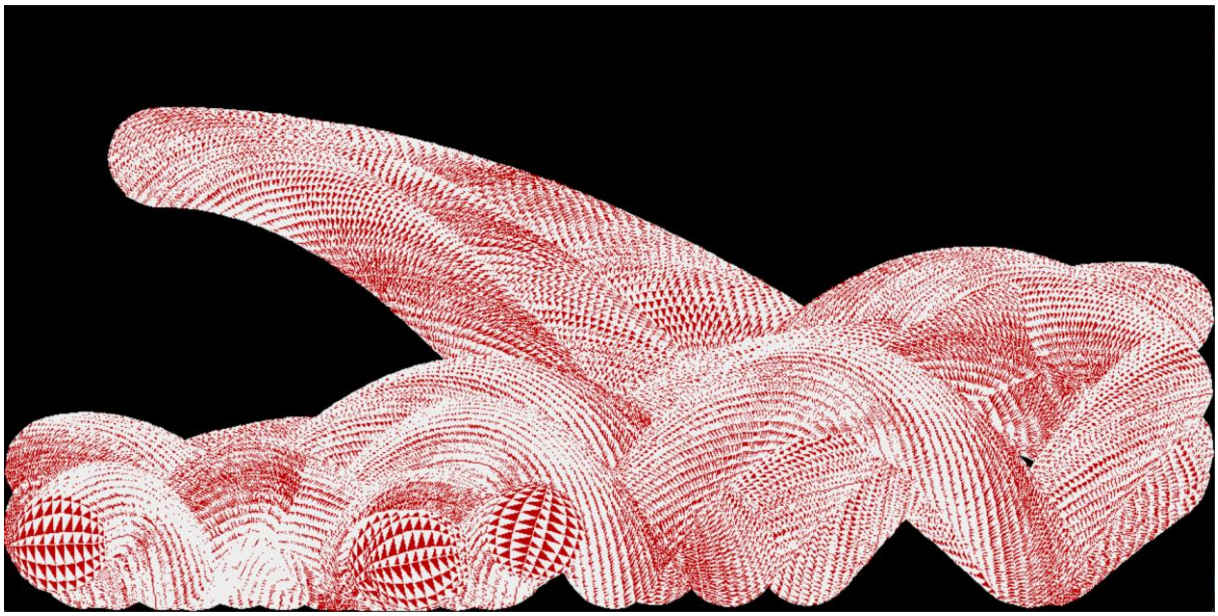
Object data structure

By creating an object data structure, we mean to create a container that holds all the different variables necessary to make something meaningful out of it. We use this information to create the ball for this assignment. To do so we must give reason to every variable that a function takes as argument. Some of these arguments we give number values, such as the speed or the position of the ball. Other arguments such as the model for the ball, is a pointer to all the information needed to draw triangles, so we can make the ball visible. We of course must create space in the memory for the object. I also added another parameter in the data structure that I found useful, that is the radius of the ball. This is something that will be a very important tool later, so I figured it would be easier to just add it to the data structure.
Now that we have created a data structure and given it space in the memory, we also need to make a function that frees or destroys the object from the memory.
For the final part to create the ball, we need to draw it on the screen. this function uses a for-loop that iterates through every triangle model that is given. This is for example the sphere model that is used in this assignment, or the teapot model that is used in the last assignment. Since these models are based on arrays, every array indices represent one triangle, therefor we must iterate through the array and draw each triangle.

Bouncing ball algorithm

inside this function is where the magic happens. But before we do anything, we must make an animation-loop. I did this by copying and pasting some pre-written code we got in the other assignments. The most important code is a while-loop that will repeat itself while the window is still open. This function uses the SDL library and will update the window surface for every time it repeats. To make the illusion of the ball moving, we must first clear the screen, or in other words, we must draw the entire screen black. Think of these as different layers, first we

draw the ball on the screen, then we make the entire screen black, then we draw the ball again but with a different position and repeat. This way it will look like the ball is moving, but it is just layers upon layers of a black screen and the balls. Here is an illustration for when we don't clear the screen.



Now that we have an animation-loop, we can start working on drawing the balls on the screen. My approach to this was to first create only one ball and figure out all the physics for the ball and how to remove it. To do so we must create a list, an object (the ball), add it to the list, then draw it. Now we have a ball that stands still on the screen. To make it move we must change the position of the ball for every iterate of the animation. Now it will move until it reaches the edge of the screen, where it will just disappear. We now need to add two different if statements that checks if the ball is outside of the screen width or the screen height. To do so we check if the position + the radius + the speed of the ball is outside of the width or height. If it is outside, we multiply the speed by negative 1, so it will change direction. We also want the ball to lose some speed when it hits the bottom of the screen, therefor we add a pre-written function in the if statement, that will reduce the speed of the ball every time the function is called.

We now have a ball that moves and bounces of the screen borders, the next step is to remove it when it stands still. To do so we add an if statement that checks if the ball is still. Since the ball uses floats for its speed, we can't simply check if the speed is equal to 0, instead we must check if the ball is between two small numbers. Since the ball can move both to the right and left, it will either have a positive or a negative speed. This is the reason why we must check if it is between two small numbers. When the ball eventually stands still, we simply reduce a

pre-set value by one for every iterate of the animation-loop. When it reaches zero, we remove the item form the list and destroy the object. When the ball is removed, we create a new one and add it in the same list.
Now the last step to complete the program is drawing more than one ball. To do so we need to create a list that contain more than one object. To solve this we make a for loop before the animation-loop that creates objects and adds them in the list. We also need to create an iterator for the list that the balls are in. With this iterator we can easily go through the balls that are in the list and check each ball individually. For this to happen we need to put the animation content inside a for loop that will repeat itself equal to the size of the list. The animation content consists of; changing the position of the ball, checking if the ball is outside of the screen borders, if the ball is standing still, and drawing of the ball. We also set the object equal to the iterate function that returns the current item it holds and moves the iterator to the next item. This way we go through every item in the list and updates their values. An important step is to reset the iterator outside of the for loop so the iterate function starts at the first item again.

Discussion

In the if statements that checks if the balls are outside of the window borders, it is very important to add the speed of the balls when we are checking the position. This is because if we don't add the speed, the ball will sink in the ground when it gets low enough speed. The reason for this is that it will not have enough speed to "escape" the window border, and therefor the position will just go outside of the border.
The if statement that removes the ball, I am not using the perfect solution. By my method it will not remove the ball by exactly 5 seconds, which the assignment stated, but it will almost be 5 seconds, sometimes less and sometimes more, depending on how fast your computer handles the program. This is because I am just decreasing a number by one every time it checks if the ball is still. So, if your computer runs the program smoothly without any lag, it will go through the animation-loop faster, and if your computer runs the program with lag, it will go through the animation-loop slower. The perfect solution to this is with the use of SDL_GetTicks, but I didn't bother to do this since my method works.

When I am removing a ball form the list, I am just creating a new one and adding it directly in the same list. By doing so we will always be using the same list, and we don't have to create a new list. This means we will never get use of the functions that frees the list and list-iterator form the memory, but I have tested it in my code, and it works. To test this, we don't add any new balls to the list when they are removed and write an if statement that checks if the size of the list is equal to zero, if it is, free list and list-iterator.

Conclusion

To conclude this report, we were tasked to write a program that animated 5 bouncing balls. We got a lot of pre-written code, and were tasked implement a linked list, a data structure to represent the balls, an animation-loop, and an algorithm for the balls to bounce of the window borders. The linked list consist of a list and a list-iterator, we used this to keep track of the balls and their values, the object data structure was used to make a container and give meaning to the different arguments it got and eventually make a ball out of it, the bouncing ball algorithm consisted of an animation loop, changing the position of the balls and detecting if the balls are outside of the window surface, removing the balls if they had been still for 5 seconds, and creating a list that contained 5 balls. With all this, we get 5 bouncing balls.

References & Acknowledgement

Figure 1: https://www.geeksforgeeks.org/linked-list-set-1-introduction/

On the bouncing ball algorithm, I worked together with Morten Jansen and Tarek Lein.