Elastic Collision Simulation in 2D Environment

Konstantin Vasilyev
Northeastern University
Boston, USA
vasilyev.k@northeastern.edu

***Abstract*—Collision simulation is one of the very universal problems used anywhere from car risk assessment to videogames. This paper is aimed at beginners in collision simulation and focuces mainly on perfectly elastic collisions of various shapes in 2D environment, including (i) shape movement; (ii) information storage; (iii) collision detection; (iv) post-collision behaviour. In this specific project, C++ language was used for compilation and single thread of an FPGA processor was used for all calculations.**

***Index Terms*—collision, 2D, physical simulation, C++, FPGA**

I. Introduction

This paper describes a project of 2D elastic collision simulation, performed on DE1-SoC runninng on an FPGA CPU. The Paper goes into detail about setting up and maintaining this simulation and is aimed primarily at beginner engineers who wish to try out collision simulation on their own. The main goal of this project was to acheive decent performance during the simulation of collisions of multiple 2D objects on the screen which collide both with the screen edges and each other. In this paper, I go into detail on how do most important part of collision simulation work which may help some individuals to replicate it for their own education purposes.

II. Algorithm Implementation

A. Timer Loop

As the main hardware interaction used in this project was manually painting every pixel, it was logical to use a global timer loop to control the flow of movement and check for collisions. The timer will make the flow of the program controllable and not dependent on number of tasks it has to perform in this specific loop. For example, for this project I have selected a frequency of 10 fps, which means the main loop runs 10 times per second, checking all collisions as moving objects around if necessary.
Since an FPGA Processor with the frequency of 200 MHz was used in the project, the timer counter was set to 20000000.

B. DDA Algorithm

Simulating collisions required having a number of shapes for the user to choose from. In order to make the displaying of the shape easier to write, a DDA line drawing algorythm [2] was used. This algorithm receives x amd y coordinates for two points and then displays a line connecting them. This then allows to display shapes such as a square by using only four functions, one for each of its sides.
The DDA Algorithm functions in a very simple way:

1) Calculate *dy* and *dx* for given coordinates
2) Compare absolute values of *dy* and *dx*, use the greater as *n* number of runs for the loop
3) Calculate values each coordinate must be incremented by with each loop like the following:
$$x_{inc} = dx/n$$
4) Run the loop *n* number of times while changing initial coordinates by calculated increments and displaying the pixel at new coordinates.

This algorithm is simple to implement and allows to paint complex shapes without much issues and potentially rotate them if it is required (although the latter was not implemented in this specific project).

C. Object information storage

There are multiple ways to store information (such as its current position, size, color, etc.) about each object to be displayed on the screen. A *linked list* can be used for this task, but for this project I have decided to use an *array of classes* with a class consisting of several integers storing information about each object, such as: current coordinates, size, color, velocity in x and y planes, movement counters. Using arrays as the main way to store information allowed me to quickly access it in a loop as each member of the array has a designated number to it.

D. Movement

In order to make movement of objects possible, I have implemented a system consisting of two velocities for x and y and a special counter for each. Since the main timer loop runs as fast as 10 times per second, these counters are incremented by one each time the main loop runs and allow to track the moment when the shape has to move.
For example, if we want the shape to have a velocity 10 pixels/second for x and 0 pix/sec for y, we set Vx to 1 and increment x coordinate by vx every time counter for vx reaches 1. If we want the shape to have a speed of 5 pix/sec for x, we set Vx to 1 but this time change shape's coordinate only when the counter reaches 2.
Using this method, I managed to display all shapes at given

Written for EECE 2160 Spring 2021 Prof. Marpaung

coordinates 10 times per second. When the counter for each reaches the pre-calculated value, Vx/Vy were added to position variables so in the next loop this shape will appear shifted.

E. Collision detection

In order to detect collision between objects on the screen, I have simplified each shape to a box with given coordinates for its top left and right bottom corners. These coordinates were used in order to process all collisions. Collision detection for objects works a little different for *Object / Screen* and *Object / Object*.

Detecting collision for ***Object / Screen*** comes down to declaring limits for x and y coordinates that exist within the specific hardware used. For example, the DE1-SoC board used in this research allows 320x240 VGA Output, so if if the object has x coordinate $posx < 0$ or $posx > 320$, and/or y coordinate $posy < 0$ or $posy > 240$, there was a collision with the screen edge and appropriate actions described in the following section must be taken.

Detecting collision for ***Object / Object*** works a little bit different since we can no longer use some static numerical value for detection, this time the only option is only to run a comparison between each pair of shapes on the screen for each main timer loop. The collision detection itself [1] is based on a very basic principle: if shape B is not above, below, on the right or on the left of shape A, shape B *must* be within shape A, therefore there was a collision of these two shapes. This principle can be further simplified into two conditions:

- Shape B is above the top edge of shape A
- Shape B is on the left of the left edge of shape A

If either of two conditions is true, there is no collision since given two shapes do not overlap.

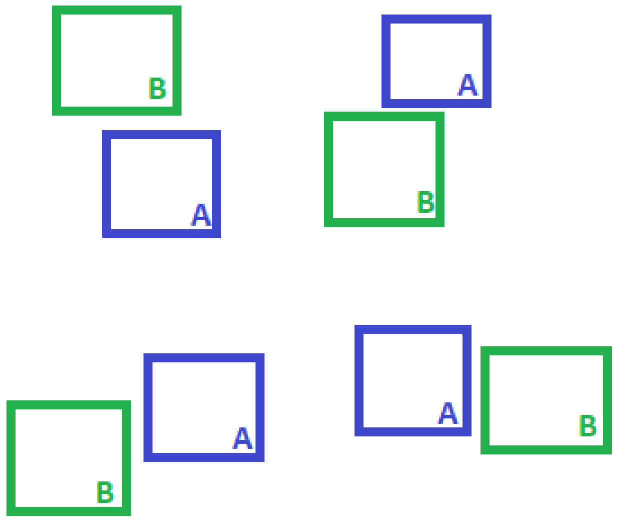

Fig. 1. No collision examples

As it can be seen on *Figure 1*, all of these examples fall under one of the conditions.

Therefore, for collision to happen, *both* conditions should be false. Within the main loop when each individual shape was re-drawn on the display, I have made so this shape runs this collision detection algorithm against every other shape on the screen and if this algorithm returns *true*, an appropriate impact is simulated.

F. Collision impact

Since this project did not attempt to recreate an accurate physical representation of collision, I have went with a very simplistic model that assumes that when an object collides with something, it keeps its momentum the same and reverses its direction of movement in order to appear as if it has bounced off. This, again, requires different approaches when the collision happens with the edge of the screen or another object.

For collision with the screen edge:

- If the object collides with top / bottom of the screen - its horizontal velocity stays the same, its vertical velocity is reversed.
- If the object collides with left / right side of the screen - its vertical velocity stays the same, its horizontal velocity is reversed.

Collision of an object with another object requires some additional evaluation since now there are no stationary objects and both objects are in movement, it is drastically important to compare initial directions of movement of given two objects before changing them. In my project I have decided to keep it relatively simple by calculating the relative positioning of two objects and then using this to decide which component of their velocities should be reversed or kept the same. To do this, I have used variables *xdiff* and *ydiff* to compare which part of the overlapping shape is more prominent (horizontal or vertical). If the horizontal overlapping was larger, I would invert the vertical velocity and horizontal velocity in an opposite situation.

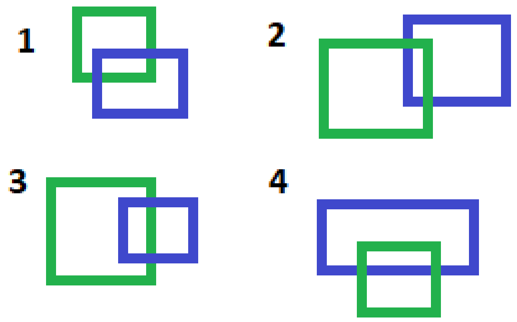

Fig. 2. Object collision examples

On *Figure 2*:

1) Difference between left edge of blue box and right edge of green box are larger than the difference between their corresponding top egde and bottom edge, therefore $xdiff > ydiff$, therefore the vertical velocity should be flipped for both objects.
2) This time vertical difference *ydiff* is larger, therefore the horizontal velocity should be flipped for both objects.
3) The horizontal difference *xdiff* is slightly larger, therefore the vertical velocity should be flipped for both objects.
4) The horizontal difference *xdiff* is again slightly larger, therefore the vertical velocity should be flipped for both objects.

When processing this type of collision it is also important to remember that since objects are currently overlapping and if we do not want them to get "stuck" in each other, they have to be manually separated. This can be done by substracting their current vertical and horizontal velocities from their current coordinates in order to reverse the last loop when they have overlapped, causing the collision detection mechanism to initially go off.
Of course, this collision mechanism does not have anything to do with any representation of realistic physics, but it allows to simulate collisions of 2D objects without many calculations and much stress put on CPU.

III. Improvements to Initial Algorithm

There were several changes I have made to the original algorithm in order to improve performance and user experience.

A. 10 Fps rendering

One of the main questions when creating some basic graphics manually is how often should you refresh the screen. Since my project did not have any specific requirements for it, I have initially chosen 1 fps rendering with all shapes being re-drawn on the screen every second. I found the performance and more importantly accuracy of such approach unsatisfactory since it not only makes all movements of the shapes appeach clunky to the human eye, but also creates potential problems with collision detection at high speeds. For example, two 10x10 pixel boxes approaching each other with a magnitude velocity of 10 pixels/second would jump though each other without any collision actually happening.
As the result, I have decided to make the main timer loop run 10 times per second, therefore creating a 10 fps output on the screen, this allowed me to acheive both more pleasing animation for human eye and more accurate collision detection at high speeds. This number can of course be increased if needed for a specific task.

B. Never clearing the whole screen

Since clearing every single pixel of 320x240 pixel display requires exactly 76,800 memory operations, it puts a lot of stress on the CPU and creates a delay visible by human eye when used during every single loop to re-draw all the objects (doing this in 10fps results in 760,800 operations per second). Because of this, I adjusted my code to specifically paint all objects with a color of 0 in the beginning of each loop to wipe out only pixels that were on, which reduced the number of operations the CPU has to perform and therefore drastically increased performance.

C. Box collision approximation

As described in detail in *Section II D*, in the final version of the project, all collisions are detected by approximating all shapes on the screen as boxes and then comparing their cooritanes with screen limits and each other, while the original version of the project has included pixel-by-pixel collision detection which was based on reading the pixel at given coordinates before another pixel would be displayed at them, and if they were "taken" already, a collision sequence would be triggered. This practically doubled the number of memory operations and therefore introduced a lot of lag, which was the main reason I have decided to rework my collision detection with a new method [1].

IV. Results

The resultant algorithm has allowed me to display 50 objects, with approximated box of 10x10 pixels, moving around the screen with the speed of 10 pixels/second in 10 fps with no delay detected by the *clock()* function. It is possible that some lag would appear with even more objects on the screen, but I have found additional testing not necessary due to relatively small resolution of the VGA output supported by specific computer board used in this project. There is no necessity to display any table since all tests have returned a value of 0 seconds for each timer loop, meaning the time the CPU takes to process everything is almost negligible.

V. Conclusions

As discussed in this paper, there are several pitfalls and issues one can encounter while constructing a collision simulator and I attempted to address them in this paper. My project aimed to acheive a decent level of performance during simulation and it did by overcoming some of the challenges and taking some optimisation decisions. It is highly recommended for any engineers attempting similar projects to begin with clearly stating their own goals since most of the decisions are based on the required level of accuracy and performance. If performance is more important than realism, boxes appoximation for object collision work well in reducing the number of operations required to detect one, and the bounce itself can be simplified to inversion of velocity with the same momentum, if it is not necessary for the simulation to follow physical laws.

References

[1] A. Gupta, "Find if two rectangles overlap," GeeksforGeeks, 18-Mar-2021. [Online]. Available: https://www.geeksforgeeks.org/find-two-rectangles-overlap/. [Accessed: 18-Apr-2021].
[2] N. Mishra, "DDA Line Drawing Algorithm in C and C++," The Crazy Programmer, 20-Jan-2017. [Online]. Available: https://www.thecrazyprogrammer.com/2017/01/dda-line-drawing-algorithm-c-c.html. [Accessed: 18-Apr-2021].