Midterm Research Project Report

Jamin Eisenberg
eisenberg.ja@northeastern.edu

Youtube link: https://youtu.be/jVT8-vfrYd4

Submit date: 4/3/2021
Due Date: 4/4/2021

1.0 Project Description

The main goal of this project was to implement and optimize a realistic 2D collision detection algorithm. The algorithm was supposed to be able to evaluate a group of shapes and determine which of the shapes were intersecting. Two shapes were said to intersect if any of their lines intersected. Auxiliary goals of the project included drawing shapes to a screen and simple collision handling – updating the trajectories of shapes after they collided with other shapes.

2.0 Hardware Used

- DE1-SoC: does all the computation and storage necessary for the project
- Keypad: 16 buttons that allow the user to create shapes and control the program
- Speaker: alerts the user when objects collide
- VGA port: allows the DE1-SoC to communicate with an external monitor for drawing

3.0 Algorithm Used

3.1 Axis-Aligned Bounding Box Collision Detection

From the IEEE paper listed below, the Axis-Aligned Bounding Box (AABB) approach was used, which is labeled “algorithm 1.”

3.2 The IEEE/ASEE Paper where the Algorithm/Method is located

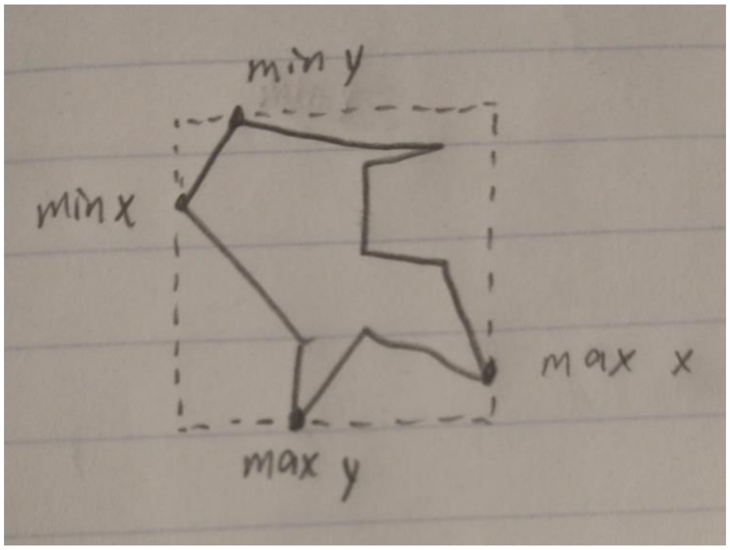
The selected IEEE paper was titled “The Design of Collision Detection Algorithm in 2D Grapple Games.” It can be found at the following link: https://ieeexplore-ieee-org.ezproxy.neu.edu/document/5363537.

Here is the full citation:

Y. Tan, C. Liu and Y. Yu, "The Design of Collision Detection Algorithm in 2D Grapple Games," *2009 International Conference on Information Engineering and Computer Science*, Wuhan, China, 2009, pp. 1-4, doi: 10.1109/ICIECS.2009.5363537.

3.3 Explanation of the Algorithm

The given algorithm is extremely simple, thought it does require an understanding of the term “AABB.” An AABB is a rectangle that fully encloses a shape and is oriented such that its lines are parallel to the x and y axes. To calculate the AABB for a given shape, we can simply go through all the shape’s points to determine the minimum x and y coordinates, and the maximum x and y coordinates. The two points that these values create are the corners of the AABB. An example is shown below.



Note that this is a reduction of the shape. If two shapes’ AABBs are colliding, that does not mean the shapes themselves are colliding. This will be elaborated on in the implementation section.

This reduction simplifies the problem significantly. Now, according to the referenced paper, all we need to do to find out if two AABBs are overlapping is to evaluate two Boolean expressions: if

$$Len1 \leq (R\_Width + R'\_Width)/2 \text{ and } Len2 \leq (R\_Height + R'\_Height)/2$$

are both true, then a collision has occurred. Len1 and Len2 here are the x and y differences between the two shapes’ positions:

$$Len1 = |R(x) - R'(x)|, \quad Len2 = |R(y) - R'(y)|$$

We now know how to determine if two shapes' AABBs are colliding. To use the algorithm for our purposes, we simply need to check if any shape is colliding with any other shape. This can be done by iterating over every shape, and for every shape, iterating over every *other* shape, and using the above expression to determine if they're colliding. In other words, we use a nested for loop to check collisions between every shape and every other shape.

This algorithm was clearly the best outlined in the paper. From the paper, "It can be easily found that the time consuming of Alg.1 is least." Additionally, it's extremely simple to implement (which is part of why it's fast). It only uses extremely cheap operations, like addition, subtraction, and Boolean operators. The sole weakness of the algorithm is not in checking whether two AABBs are colliding. Instead, it's the fact that we need to check every shape with every other shape. If we have *n* shapes, that's n^2 checks. This scales very quickly. Checking 10 shapes makes for 100 checks, but 100 shapes is 10,000 checks! This is exactly where I propose improvements can be made.

3.4 Implementation of the Algorithm

Due to the simplicity of the algorithm, I was able to implement it myself easily. There were several parts to doing this. I prefer top-down design, so the first part was defining how to check whether any two shapes are colliding. I chose to store an array of shapes (Polygons) in my program, so the check looked like the following:

```
for(int i = 0; i < numShapes; i++) {
    for (int j = 0; j < numShapes; j++) {
        if (&shapes[i] == &shapes[j]) continue;
        if (shapes[i].isColliding(&shapes[j])) collided->push_back(new
CollisionPair(&shapes[i], &shapes[j]));
    }
}
```

(Note that this had to be broken up in my final program, for checking collision upon insertion.)

This is straightforward. For each shape in the array, for each shape in the array (again), if the shape from the outer loop is colliding with the shape from the inner loop, add the shapes to a list of colliding shapes. The one tricky part is that we don't want to check a shape against itself, because that would always give us a collision.

This next part is where we implement the part of the algorithm given in the IEEE paper. We reference an "isColliding" function above that we have to define. Here it is:

```
bool Polygon::isColliding(Polygon *other) {
    int len1 = abs(this->pos.x - other->pos.x);
    int len2 = abs(this->pos.y - other->pos.y);

    Vector2D thisSize = this->aabbMax.copy();
    thisSize.sub(&this->aabbMin);

    Vector2D otherSize = other->aabbMax.copy();
    otherSize.sub(&other->aabbMin);

    if (len1 <= (thisSize.x + otherSize.x) / 2 &&
        len2 <= (thisSize.y + otherSize.y) / 2) {

        // some code that will return true if the shapes' lines are colliding
    }

    return false;
}
```

There are some obvious parallels with the algorithm in the paper. We define Len1 to be the x distance between the shapes, and Len2 to be the y distance. Rather than using R_Width-like notation, I used a vector class I created to store the width and height of the AABBs. Then I use the exact Boolean expression laid out in the paper. If it's true, we run some code to check whether the shapes' lines are intersecting. If it's not true, we know the shapes are not colliding.

The line intersection code is outside the scope of this algorithm, but I used code found here: https://gamedev.stackexchange.com/questions/26004/how-to-detect-2d-line-on-line-collision

The last part of the implementation of this algorithm is calculating the AABBs! We use them in the above code, but they need to be calculated. This is done in Polygon's update method. It's pretty standard min/max finding code.

```
int xMax, yMax;
int xMin = xMax = lines->at(0).p1.x;
int yMin = yMax = lines->at(0).p1.y;

for (int i = 1; i < lines->size(); i++) {
    Vector2D point = lines->at(i).p1;
    if (point.x < xMin) xMin = point.x;
    if (point.y < yMin) yMin = point.y;
    if (point.x > xMax) xMax = point.x;
    if (point.y > yMax) yMax = point.y;
}
aabbMin.x = xMin;
aabbMin.y = yMin;
aabbMax.x = xMax;
aabbMax.y = yMax;
```

We simply iterate through the list of this Polygon's lines and keep track of the largest and smallest x and y values that have been seen so far. We then use those points to construct the AABB for the shape. This is done every frame, so the AABB is always a snug fit around the shape.

4.0 Improvement Evaluation

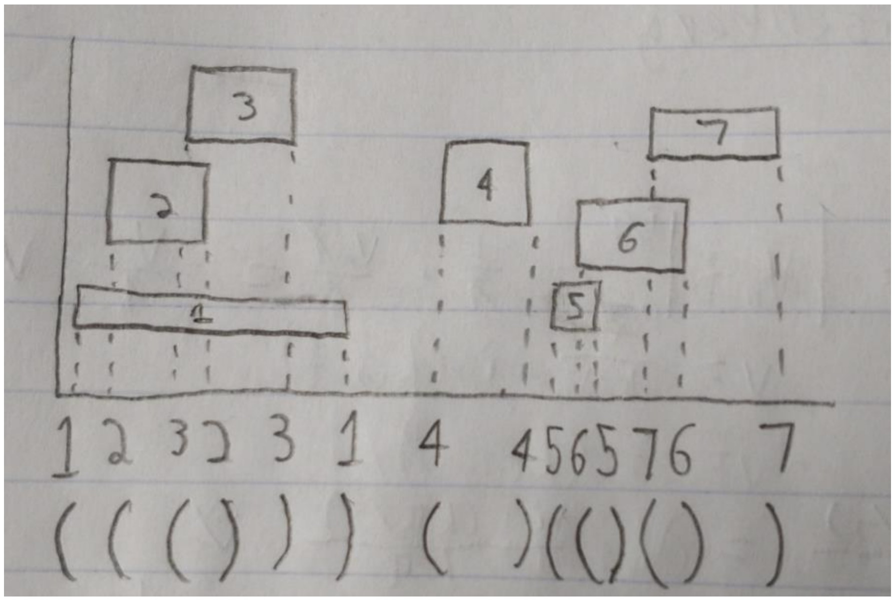
4.1 Goal/Target of Improvement

The general goal of the improvement was to reduce the time complexity of the function from O(n^2) to O(n log n). Where n = 100, calculating collisions with an O(n^2) algorithm made for 10,000 calculations. With an O(n log n) algorithm, this is reduced to just 200 calculations (assuming log base 10). This is just for illustrative purposes; these aren't concrete values. As is apparent, O(n log n) is a massive improvement. More concretely, my hope was to be able to simulate collisions for 100 random shapes without a significant frame rate reduction. This part of the evaluation includes drawing, moving, and handling collisions.

4.2 Current State of Improvement

As mentioned above, the biggest slowdown in the original algorithm is the fact that every shape needs to be checked against every other shape. We can eliminate many of these checks using a common problem reduction in computer science: sorting.

First, we must change and organize our approach a little bit. I propose that we think about detecting collisions between AABBs as an "interval" problem. The edges of shapes act as opening and closing intervals, and our goal is to check if any two intervals are open at the same time. Here's an illustration of what I mean by this:



We are now presented with a different problem. The goal now is to determine which intervals overlap. In other words, which intervals become active while another interval is active. Before this is possible as in the illustration above, we need to do two things: get the intervals from the shapes and sort the intervals by their positions. Here is the Interval struct header I created:

```
struct Interval {
    double *pos;
    bool isBegin;
    int shapeIndex;

    Interval();
    Interval(double *pos, bool isBegin, int shapeIndex);
    bool operator<(const Interval &i2) const;
};
```

This keeps track of all the things that are kept track of in the figure above: the position of the interval, whether it's the start or the end of an interval, and which shape it refers to. Additionally, a less-than operator is defined for sorting. Intervals are sorted by their position using std::sort, which is a highly optimized sorting algorithm with time complexity O(*n* log *n*). Intervals are easy to create if you have a polygon. Here's an example:

```
Interval(&shapes[i].aabbMin.x, true, i)
```

This creates an interval with the minimum AABB x-value, which makes it the start of an interval, and it has index I, which is where the shape came from.

That's all the necessary setup, the actual algorithm follows. The core of it happens in getPossibleCollided(), which takes a vector of collision intervals and the main array of shapes. It returns a set of CollisionPairs filled with all of the Polygons that are colliding with each other. This set starts out as empty, as does the (linked) list of active intervals. As described, the vector of intervals is sorted. Then,

- For each interval marker in the vector of intervals:
    - If the interval marker is the start of an interval
        - If there's at least one active interval
            - Insert all of the active intervals into the set of possible collisions (by making a CollisionPair with them and the current interval marker)
        - Prepend this interval marker to the list of active intervals
    - If it's the end of an interval
        - Find the interval marker's starting pair
        - Remove it from the list of active intervals

To rephrase briefly, we keep track of all the shapes we're currently "inside of" as we traverse the vector of intervals. If a new interval opens when there are active intervals, we know there's a collision, so we add it to the set of CollisionPairs. When we encounter a closing interval marker, we delete its opening pair from the list of active intervals. This whole process allows us to find collisions on one axis…

But we need to find collisions on both axes! If we get a set of the CollisionPairs on each axis, finding the intersection of those two sets will give us the shapes colliding on *both* axes, which are exactly the ones we want. Success.

Aside: this part was tricky because I had to define a way to compare CollisionPairs. In the end I went with comparing the first two, and if they weren't the same, comparing the second two. This is like how alphabetical strings are compared. This is the cpCmp struct you'll see in the code.

I'll gloss over time complexity here a little bit because the figures below show clear success in this area. Inserting an item into a set, which could happen *n* times, is in O(log *n*), so that makes O(*n* log *n*). Inserting an item onto my linked list implementation is always a constant time operation, so we don't have to worry about it. The real parts that seem problematic are finding and erasing an item from the linked list, which is O(*n*) time, and could theoretically happen *n* times. Although, we know that the thing we're trying to find will usually be close to the front of the list because that's where things are inserted, so it's not a big concern.

While sorting and interval problems are not new to computer science, I devised and implemented this solution entirely myself.

4.3 Method of Evaluation

I decided to use CPU time to evaluate the efficiency of the original algorithm and my improved algorithm. Time complexity of algorithms is usually considered the most important attribute of an algorithm in computer science, but particularly in graphics and game development. I wanted to use a respected form of evaluation. I decided against wall time simply because it seemed less consistent and accurate for measurement. I didn't want the measurement to rely on other aspects of the DE1-SoC like any other processes it may be running. There are only a couple of ways to measure CPU time in C++ in Linux. I chose <ctime> for its simplicity.

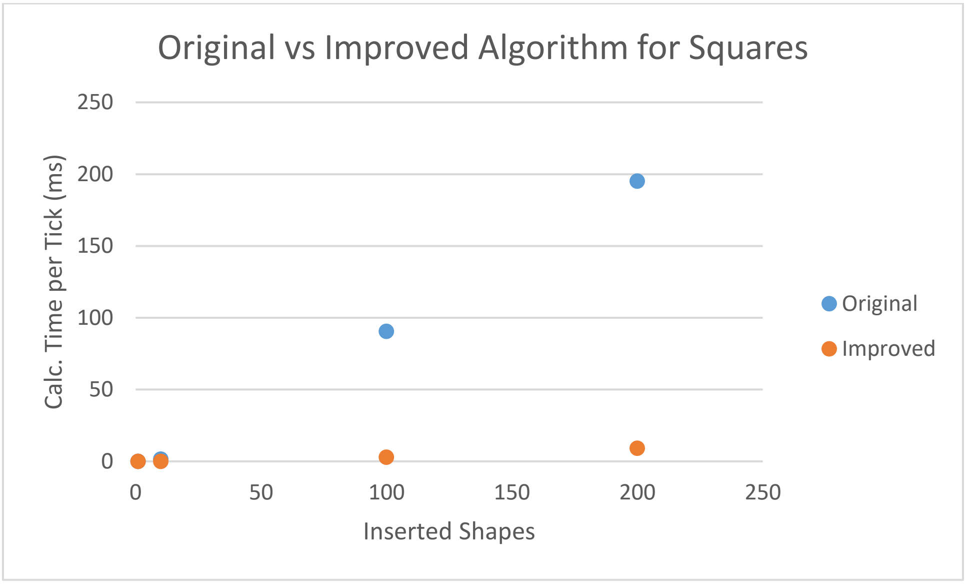
I only measured the time it took the algorithms to generate all the collision pairs. I did not include drawing, handling, or updating time. This is partially because it was likely to be the same for both algorithms, and partially because I was only evaluating the collision detection algorithms themselves, not the other algorithms.

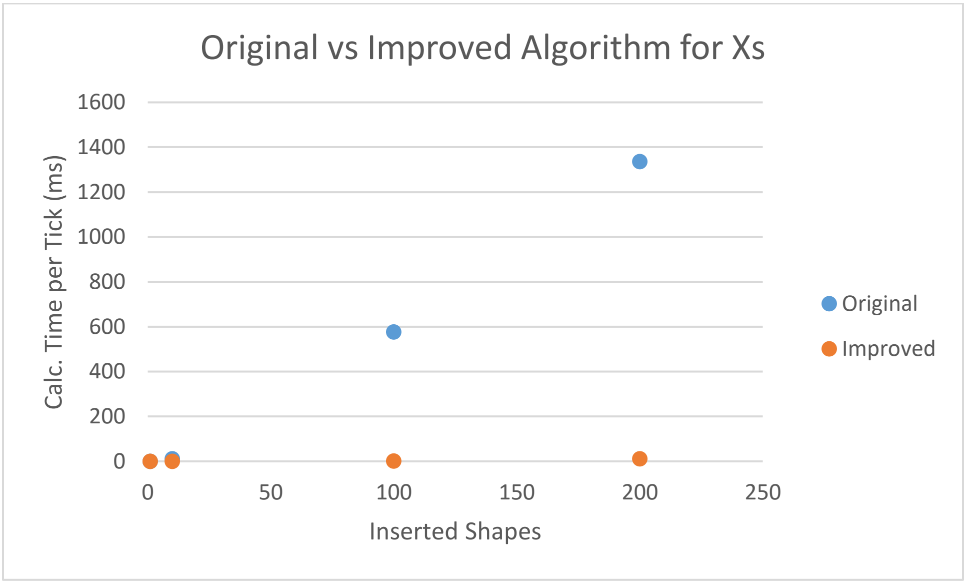
For the measurements, I decided to vary the number of inserted shapes (at 1, 10, 100, 200), as well as which shape was being drawn. I compared the square shape to the 'X' shape for both algorithms. The square shape had 4 lines and the 'X' shape had 12. Note that I varied the number of *inserted* shapes. In my implementation, the edges of the screen were also Polygons that had to be processed, so inserting 1 shape makes 5 shapes total.

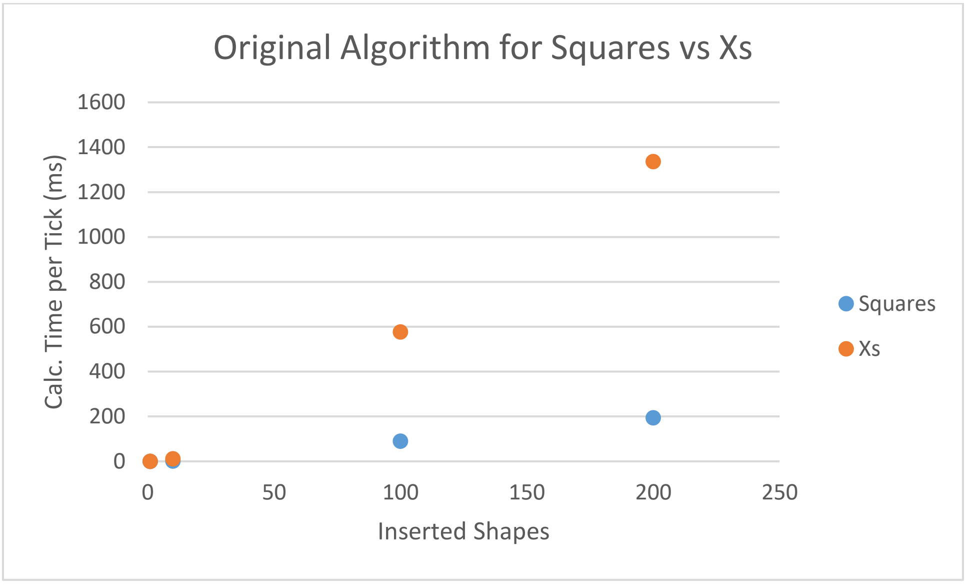
4.4 Result of Improvement

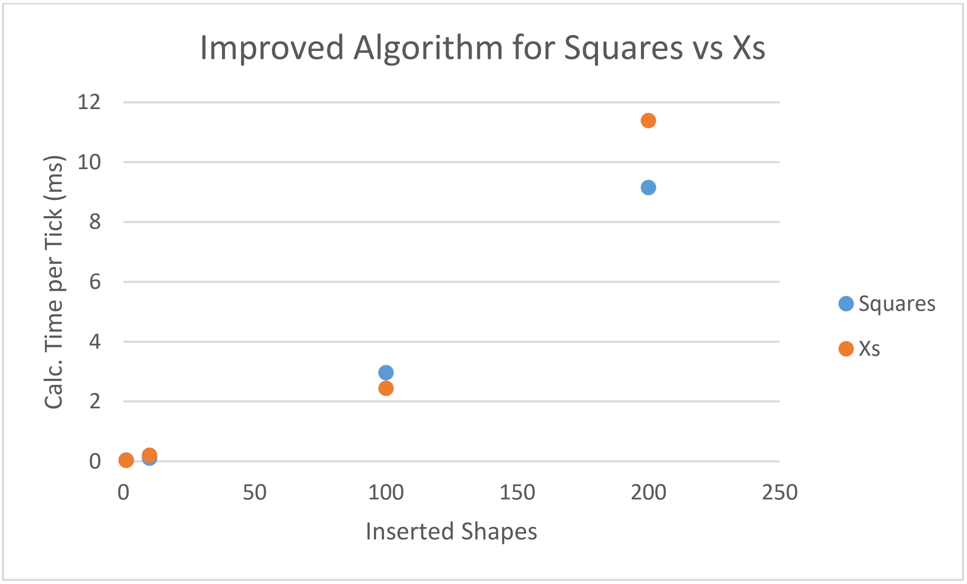
As shown in the table and figures below, my improved algorithm is faster than the original in every case!

Shape	Inserted Shapes	Average Calculation Time per Tick (ms)		Speed Improvement Factor
		Original Algorithm	Improved Algorithm	
Square	1	0.133668	0.0506073	2.6
	10	1.57945	0.110429	14.3
	100	90.6606	2.97351	30.5
	200	195.187	9.15208	21.3
X	1	0.241546	0.0365497	6.6
	10	12.4653	0.202086	61.7
	100	577.627	2.44561	236.2
	200	1336.33	11.396	117.3


Original vs Improved Algorithm for Squares
Calc. Time per Tick (ms)
Inserted Shapes
0
50
100
150
200
250
● Original
● Improved


Original vs Improved Algorithm for Xs
Calc. Time per Tick (ms)
Inserted Shapes
0
200
400
600
800
1000
1200
1400
1600
0
50
100
150
200
250
● Original
● Improved





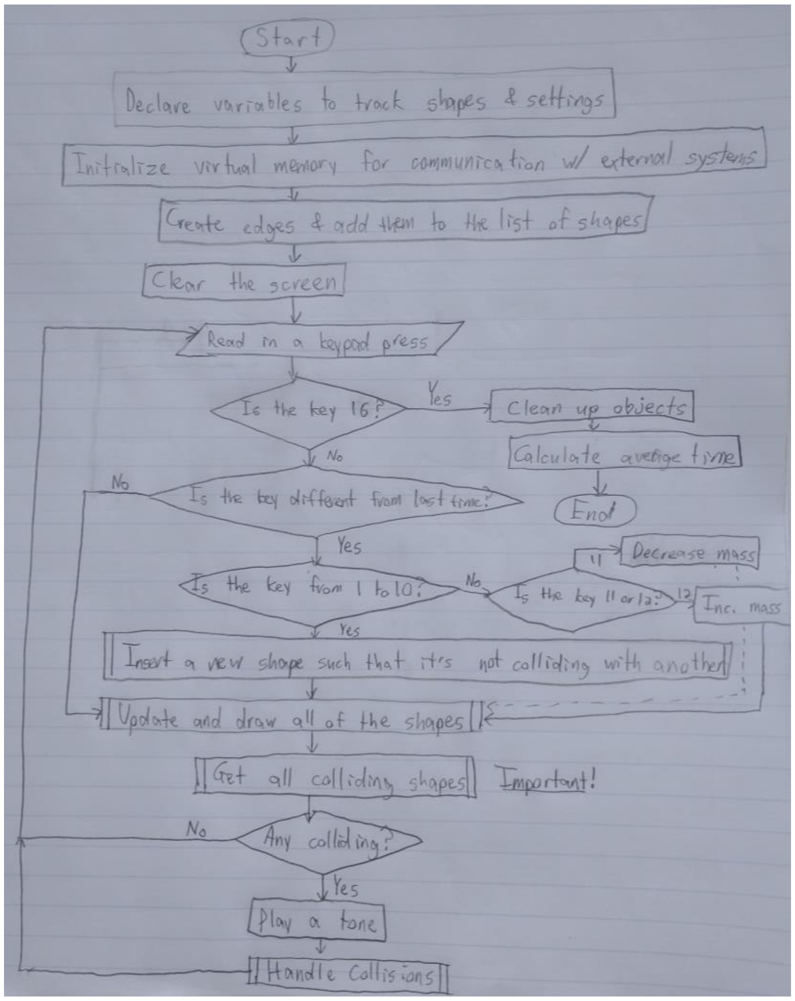
As you can see, there is a very large difference between my improved algorithm and the original algorithm. It is particularly noticeable for higher values of inserted shapes. I am particularly proud of the fact that my algorithm is over 200 times faster than the original for 100 'X' shapes. The plot also clearly shows that the original algorithm's time complexity is *growing* faster than my improved algorithm, which is really the key.

One positive thing that I was not expecting was that my algorithm reduced the difference between computation times for different shapes. The original algorithm has a very large difference between the computation times of the 'X' shape and the square shape – a difference that the improved algorithm does not have.

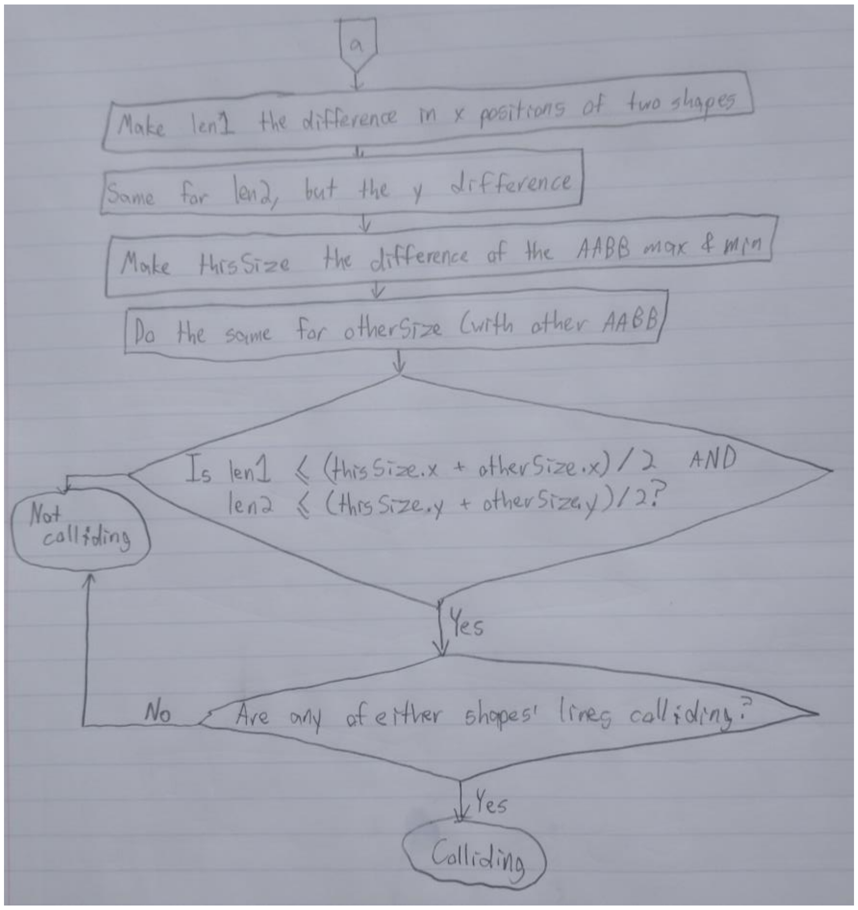
5.0 C++ Code Explanation

5.1 Flowchart of the code found in Section 6.0

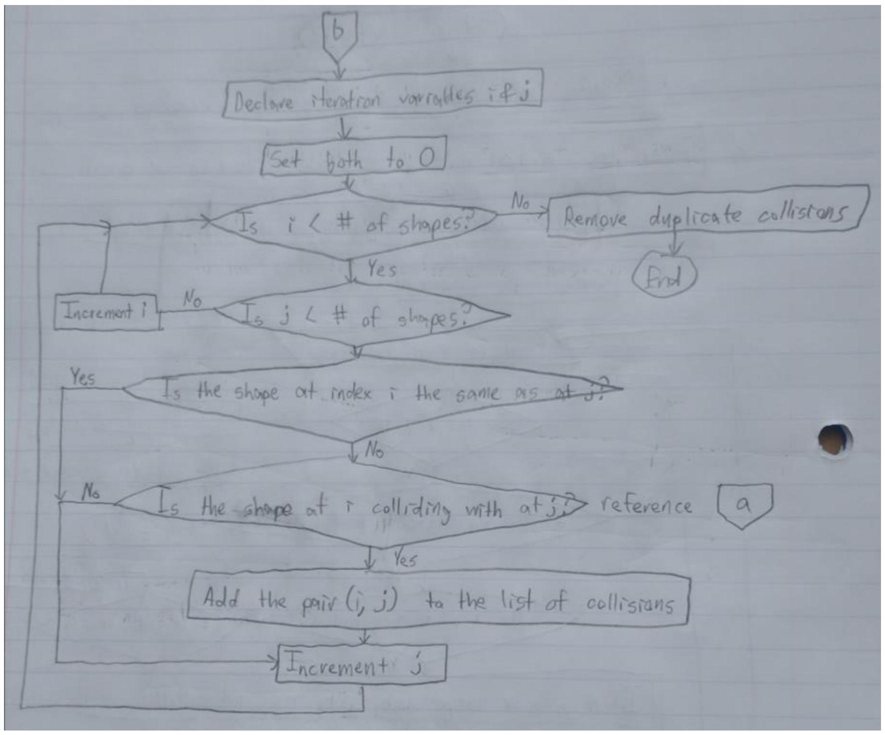
Main method flowchart


Start
Declare variables to track shapes & settings
Initialize virtual memory for communication w/ external systems
Create edges & add them to the list of shapes
Clear the screen
Read in a keypad press
Is the key 16?
Yes
Clean up objects
Calculate average time
End
No
Is the key different from last time?
No
Yes
Is the key from 1 to 10?
No
Is the key 11 or 12?
11
Decrease mass
12
Inc. mass
Yes
Insert a new shape such that it's not colliding with another
Update and draw all of the shapes
Get all colliding shapes
Important!
Any colliding?
No
Yes
Play a tone
Handle Collisions

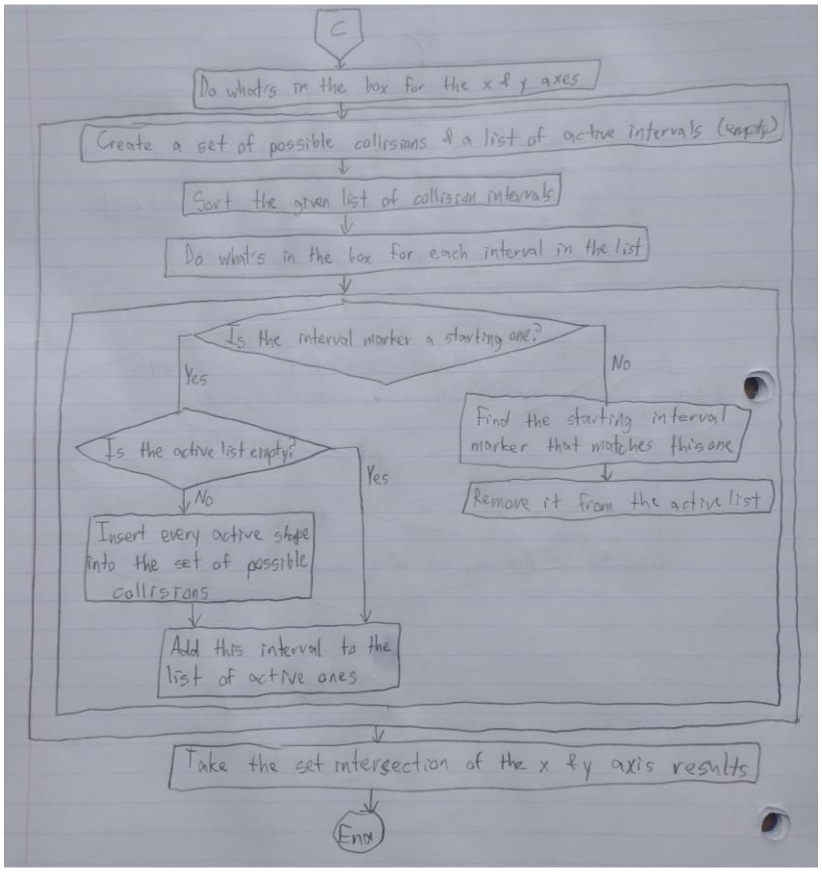
Both algorithms' AABB collision detection flowchart


a
Make len1 the difference in x positions of two shapes
Same for len2, but the y difference
Make thisSize the difference of the AABB max & min
Do the same for otherSize (with other AABB
Is len1 ≤ (thisSize.x + otherSize.x) / 2 AND len2 ≤ (thisSize.y + otherSize.y)/2?
Not colliding
Yes
No
Are any of either shapes' lines colliding?
Yes
Colliding

Original algorithm flowchart


b
Declare iteration variables i & j
Set both to 0
Is i < # of shapes?
No
Remove duplicate collisions
End
Yes
Increment i
No
Is j < # of shapes?
Yes
Is the shape at index i the same as ~~at~~ j?
No
No
Is the ~~shape~~ shape at i colliding with at j?
reference
a
Yes
Add the pair (i, j) to the list of collisions
Increment j

Improved Algorithm Flowchart


c
Do what's in the box for the x & y axes
Create a set of possible collisions & a list of active intervals (empty)
Sort the given list of collision intervals
Do what's in the box for each interval in the list
Is the interval marker a starting one?
Yes
No
Is the active list empty?
Yes
No
Insert every active shape into the set of possible collisions
Add this interval to the list of active ones
Find the starting interval marker that matches this one
Remove it from the active list
Take the set intersection of the x & y axis results
End

5.2 Explanation of the Code found in Section 6.0

I'll start with a bottom-up class outline for each class detailing what they contain and what they can do:

- Vector2D
  - Contains x and y coordinates representing a physics vector
  - Can be translated by another Vector2D using the add and sub functions
  - Can be scaled by a constant factor using the mult function
  - Can be rotated by an angle using the rotate function
    - Uses the imaginary plane for speed and convenience
  - Can be copied for convenience
  - Can be drawn
- Line
  - Contains two endpoints that are Vector2Ds
  - Can do any transformation a Vector2D can do
    - The transformation is just called on its composite points
  - Can detect collisions between itself and another Line
    - Using an algorithm like the one found here: https://gamedev.stackexchange.com/questions/26004/how-to-detect-2d-line-on-line-collision
  - Can be drawn
    - Using the Bresenham line-drawing algorithm found here: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html
- Polygon
  - Contains a position, velocity, angle and angular velocity
  - Contains a mass for resolving collisions and sizing
  - Contains a vector of lines that make up the shape
  - Contains an id representing its place in the main shapes array
  - Can detect collisions between itself and another Polygon
    - Uses the algorithm described above
  - Can be updated
    - Updates the AABB as discussed above
    - Updates position, angle, and lines by adding the velocity vector and the angular velocity to them
  - Can be drawn
    - Draws each of its lines, then its position, to the screen
  - Can be compared (by id) to another Polygon

- CollisionPair
    - Contains two Polygons that are guaranteed to be colliding
    - Can be compared to other CollisionPairs, for equality and inequality
        - Compared lexicographically, as described above
    - Can handle the collision between its Polygons (create a valid and semi-realistic response)
        - Explained in detail below

That's the Polygon hierarchy. Now for the FPGA hierarchy. We'll do this one top-down:

- DE1SoCfpga
    - Can read and write to registers that have been mapped to virtual memory
- Screen
    - Contains lots of externally useless information about what the VGA port has learned about its display
    - Can write black or white to a pixel at a given coordinate
        - Uses the operations in the "Video" example found in the manual
    - Can update the screen with the most recently drawn pixels
        - Uses buffer swapping outlined in the chip's manual
        - Waits until it's time to update (60 Hz) using the timing feature outlined in the manual
    - Can be cleared
        - Writes every pixel black
- Speaker
    - Can play a given tone for a given duration
- Keypad
    - Can read the key that is currently being pressed on an external keypad
        - Explained in detail below

My linked list is sort of outside the hierarchy. That file contains Interval as well, which has already been explained.

- LoIntervals is one of: MtLoInterval, ConsLoInterval
    - Can return whether it's empty
    - Can return its size
    - Can return the Interval at the given index
    - Can erase the Interval at the given index
    - Fundies-style recursive, dynamic dispatch design

Keypad::read():

First, the rows are set to be outputs and the columns inputs by writing to the direction register. Then, each row is made HIGH one at a time, and a 2D array is created by reading the data register at the proper pins (which are stored in a COL_VALS array). If the button is pressed, that column should read HIGH when the row is HIGH. This generated a strange result, even after tweaking it for some time. Every row except the one that was pressed was HIGH in every column. The row where one was pressed was LOW except for one of them, which was the pressed one.

So, we then just traversed this array row-by-row until a zero was encountered, then traversed that column until we found a HIGH value, and that was the key that was pressed. The function returns a number between 0 and 16, where 0 is no keys pressed, and 1 – 16 are the corresponding keys on the keypad.

Collision handling is the one not entirely complete part of this project, a fact I am not very happy about because I poured quite a lot of work into it. I did my best to implement ideas found here: http://www.kuffner.org/james/software/dynamics/mirtich/mirtichThesis.pdf, particularly starting on page 56. However, my solution only works *sometimes*, and I can't even tell what differentiates the cases where it works from the cases it doesn't work. I'll briefly explain what I have anyway.

First, the relative velocities and positions of the polygons are calculated by subtracting them from one another. Then, the half-widths and heights for both are calculated using their AABBs. The x and y overlap are calculated by adding these half-sizes and subtracting their relative positions. This represents how far in each direction the Polygons have penetrated each other. If the x overlap is larger, we know that the collision must be resolved in the y direction and vice versa (it helps me to think of a square and a large wall).

The normal of the collision is negative if the first shape's position is larger than the second's, and positive otherwise. The velocity along this normal is just the dot product of the normal and the relative velocity. If the velocity is positive, they're moving away, so there is no resolution necessary. The impulse itself will be a negative version of the initial velocity, as described in the linked dissertation. The normal is then scaled by the impulse and added to each shape's velocity, according to their mass proportions of the total mass.

5.3 Polymorphism or Linked List

I used a linked list (that uses polymorphism) in my program. I chose to use a linked list for my improved algorithm for collision detection because insertion is always constant time, which is better than vectors' amortized constant time. The list of active intervals is a linked list. That is the only place I use it and that is the only reason why I chose it.

My definition of a linked list can be found on pages 28 and 29, and it is used in the improved algorithm on pages 40 and 41.

6.0 C++ Code

6.1 Original Program and Improved Program (comments differentiate the portions that are different – only in main.cpp)

CollisionPair.h

```
#ifndef PROJECT_COLLISIONPAIR_H
#define PROJECT_COLLISIONPAIR_H


#include "Polygon.h"

// represents a pair of Polygons that we know are colliding
class CollisionPair {
public:
    Polygon *p1;
    Polygon *p2;

    // guarantees that the smaller Polygon will be in the first slot, for ordering
purposes
    CollisionPair(Polygon *, Polygon *);

    // compares two CollisionPairs by comparing the ids of their Polygons
    static bool equals(CollisionPair *, CollisionPair *);

    // compares two CollisionPairs by comparing their Polygons lexicographically
    bool operator<(const CollisionPair &p2) const;

    // updates the Polygons' trajectories based on their past trajectories
    void handleCollision();
};


#endif
```

DE1SoCfpga.h

```
#ifndef DE1SOCFPGA_H
#define DE1SOCFPGA_H

const unsigned int LW_BRIDGE_BASE = 0xFF200000;
const unsigned int LW_BRIDGE_SPAN = 0x00DEC700;
const unsigned int SW_OFFSET = 64;

const unsigned int DATA_REG_ADDR = 0x60;

// represents a memory mapping that can be used to get values "built into" the board
class DE1SoCfpga {
public:
    char *pBase;
    int fd;

    // maps memory to be accessed
    DE1SoCfpga(int bridgeBase = LW_BRIDGE_BASE, int bridgeSpan = LW_BRIDGE_SPAN);

    ~DE1SoCfpga();
    // writes the given value to the given mapped address
    void RegisterWrite(unsigned intoffset, int value);

    // reads the value at the given mapped address
    int RegisterRead(unsigned intoffset);
};

#endif
```

Keypad.h

```
#ifndef KEY_H
#define KEY_H

#include "DE1SoCfpga.h"

const unsigned int DIRECTION_REG_ADDR = DATA_REG_ADDR + 4;

const unsigned int COL_VALS[] = {14, 13, 11, 9};
const unsigned int ROW_VALS[] = {21, 19, 17, 15};

// represents a 4x4 external keypad
class Keypad : public DE1SoCfpga {
public:
    Keypad();

    ~Keypad();

    // gets the key being currently pressed
    int read();
};

#endif
```

Line.h

```
#ifndef PROJECT_LINE_H
#define PROJECT_LINE_H

#include "Vector2D.h"
#include "Screen.h"

// represents a line with two endpoints
class Line {
public:
    Vector2D p1;
    Vector2D p2;

    // creates a line with the given vectors as endpoints
    Line(Vector2D *, Vector2D *);

    ~Line();
    // translates the vector by the given vector
    void add(Vector2D*);

    // translates the vector by the negative of the given vector
    void sub(Vector2D*);

    // scales the vector by a constant factor
    void mult(double);

    // rotates the vector by a given angle
    void rotate(double);

    // return whether this line is colliding with the given line
    bool isColliding(Line*);

    // draws this line to the given screen
    void draw(Screen*, bool);
};

#endif
```

LoIntervals.h

```
#ifndef PROJECT_LOINTERVALS_H
#define PROJECT_LOINTERVALS_H

// represents an interval marker, the opening or closing of a shape interval
struct Interval {
    double *pos;
    bool isBegin;
    int shapeIndex;

    Interval();

    Interval(double *pos, bool isBegin, int shapeIndex);

    // compares intervals by comparing their positions
    bool operator<(const Interval &i2) const;
};

// represents a linked list of intervals
class LoIntervals {
public:
    // return whether the list is empty
    virtual bool empty() = 0;

    // returns the size of the list
    virtual int size() = 0;

    // returns the element at the given index
    virtual Interval *at(int index) = 0;

    // returns the list with the element at the given index removed
    virtual LoIntervals *erase(int index) = 0;
};

class MtLoInterval : public LoIntervals {
public:
    MtLoInterval();

    bool empty();

    int size();

    Interval *at(int index);

    LoIntervals *erase(int index);
};

class ConsoLoInterval : public LoIntervals {
private:
    Interval *first;
    LoIntervals *rest;
public:
    ConsoLoInterval(Interval *, LoIntervals *);

    bool empty();

    int size();

    Interval *at(int index);

    LoIntervals *erase(int index);
};

#endif
```

Polygon.h

```
#ifndef PROJECT_POLYGON_H
#define PROJECT_POLYGON_H

#include "Vector2D.h"
#include "Line.h"
#include "Screen.h"
#include <vector>

using namespace std;

// represents a polygon, in the traditional, geometric sense
class Polygon {
private:
    Vector2D pos;
    Vector2D vel;
    double angle;
    double angleVel;
    double inv_mass;
    vector<Line> *lines;

    friend class CollisionPair;

    // initializes a polygon with all of the necessary information, called by
constructors
    void initPolygon(Vector2D *, Vector2D *, double, double, double, vector<Line> *,
int);

public:
    Vector2D aabbMin;
    Vector2D aabbMax;

    int id;

    Polygon();

    Polygon(Vector2D *, Vector2D *, double, double, double, vector<Line> *, int);

    Polygon(Vector2D *, Vector2D *, double, double, double, int, int);

    ~Polygon();

    // returns whether this Polygon is colliding with the given one
    bool isColliding(Polygon *);

    // updates the AABB, position, angle, and lines of this Polygon, once per tick
    void update(Screen *);

    // draws this Polygon to the given screen
    void draw(Screen *, bool);

    // compares two Polygons by their ids
    bool operator<(const Polygon &p2) const;
};

#endif
```

Screen.h

```


#ifndef PROJECT_SCREEN_H
#define PROJECT_SCREEN_H

#include "DE1SoCfpga.h"

//const unsigned int BASE = 0xC0000000;
const unsigned int SPAN = 0x4B000;

const unsigned int PIXEL_BUF_CTRL_BASE = 0x3020;
const unsigned int RGB_RESAMPLER_BASE = 0x3010;
```

```
// represents the screen the VGA port draws to
class Screen {
private:
    char *pBase;
    int fd;

    int screen_x;
    int res_offset;
    int col_offset;
    int x_factor, y_factor;

    int speaker;

    short white;

    int resample_rgb(int, int);
    int get_data_bits(int);

public:
    Screen(DE1SoCfpga*);

    ~Screen();

    // writes black or white (bool) at the given pixel (x, y)
    void writePixel(int, int, bool);

    // Uses buffer swapping outlined in the chip’s manual
    // Waits until it’s time to update (60 Hz) using the timing feature outlined in
the manual
    void update(DE1SoCfpga*);

    // writes every pixel black
    void clear();
};

#endif
```

Speaker.h

```
#ifndef PROJECT_SPEAKER_H
#define PROJECT_SPEAKER_H

#include "DE1SoCfpga.h"

const unsigned int MPCORE_PRIV_TIMER_LOAD_OFFSET = 0xDEC600;
const unsigned int MPCORE_PRIV_TIMER_COUNTER_OFFSET = 0xDEC604;
const unsigned int MPCORE_PRIV_TIMER_CONTROL_OFFSET = 0xDEC608;
const unsigned int MPCORE_PRIV_TIMER_INTERRUPT_OFFSET = 0xDEC60C;

class Speaker : public DE1SoCfpga {
private:
    unsigned int initialvalueLoadMPCore;
    unsigned int initialvalueControlMPCore;
    unsigned int initialvalueInterruptMPCore;
public:
    Speaker();

    ~Speaker();

    // plays the given tone for the given duration
    void playTone(double, int);
};


#endif
```

Vector2D.h

```
#ifndef PROJECT_VECTOR2D_H
#define PROJECT_VECTOR2D_H

#include "Screen.h"

// represents a 2D vector for use in physics
class Vector2D {
public:
    double x;
    double y;

    Vector2D();

    Vector2D(double, double);

    ~Vector2D();

    // translates the vector by the given vector
    void add(Vector2D*);

    // translates the vector by the negative of the given vector
    void sub(Vector2D*);

    // scales the vector by a constant factor
    void mult(double);

    // rotates the vector by a given angle
    void rotate(double);

    // draws this line to the given screen
    void draw(Screen*, bool);

    // copies this vector to a new vector
    Vector2D copy();
};

#endif
```

CollisionPair.cpp

```
#include "CollisionPair.h"
#include <iostream>
#include <stdlib.h>

// guarantees that the smaller Polygon will be in the first slot, for ordering
purposes
CollisionPair::CollisionPair(Polygon *p1, Polygon *p2) {
    if (p1->id < p2->id) {
        this->p1 = p1;
        this->p2 = p2;
    }
    else{
        this->p1 = p2;
        this->p2 = p1;
    }
}

// compares two CollisionPairs by comparing the ids of their Polygons
bool CollisionPair::equals(CollisionPair *a, CollisionPair *b) {
    return a->p1->id == b->p1->id && a->p2->id == b->p2->id;
}

// compares two CollisionPairs by comparing their Polygons lexicographically
bool CollisionPair::operator<(const CollisionPair &cp2) const {
//    return this->p1 < cp2.p1 || this->p2 < cp2.p2;
//    return (this->p1 < cp2.p1 && this->p2 < cp2.p2) || (this->p1 < cp2.p2 && this-
>p2 < cp2.p1);
//    return this->intRep() < cp2.intRep();
    if(this->p1->id != cp2.p1->id){
        return this->p1->id < cp2.p1->id;
    }
    else{
        return this->p2->id < cp2.p2->id;
    }
}

// updates the Polygons' trajectories based on their past trajectories
void CollisionPair::handleCollision() {
    // currently not quite working

    Vector2D rv = p2->vel.copy();
    rv.sub(&p1->vel);

    Vector2D *normal;

    Vector2D n = p2->pos.copy();
    n.sub(&p1->pos);

    double ax_extent = abs(p1->aabbMax.x - p1->aabbMin.x) / 2;
    double bx_extent = abs(p2->aabbMax.x - p2->aabbMin.x) / 2;
    double x_overlap = ax_extent + bx_extent - abs(n.x);

    double ay_extent = abs(p1->aabbMax.y - p1->aabbMin.y) / 2;
    double by_extent = abs(p2->aabbMax.y - p2->aabbMin.y) / 2;

    double y_overlap = ay_extent + by_extent - abs(n.y);

    if (x_overlap > y_overlap) {
        if (n.x < 0)
            normal = new Vector2D(0, -1);
        else
            normal = new Vector2D(0, 1);
    } else {
        if (n.y < 0)
            normal = new Vector2D(-1, 0);
        else
            normal = new Vector2D(1, 0);
    }

    double velAlongNormal = rv.x * normal->x + rv.y * normal->y;

    if (velAlongNormal > 0){
        return;
    }

    double j = -2 * velAlongNormal;
    j /= p1->inv_mass + p2->inv_mass;

    normal->mult(j);

    Vector2D impulse1 = normal->copy();
    impulse1.mult(p1->inv_mass);

    Vector2D impulse2 = normal->copy();
    impulse2.mult(p2->inv_mass);

    p1->vel.sub(&impulse1);
    p2->vel.add(&impulse2);
}
```

DE1SoCfpga.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>

#include "DE1SoCfpga.h"
```

```
using namespace std;

// maps memory to be accessed
DE1SoCfpga::DE1SoCfpga(int bridgeBase, int bridgeSpan) {
    fd = open("/dev/mem", (O_RDWR | O_SYNC));
    if (fd == -1) // check for errors in openning /dev/mem
    {
        cout << "ERROR: could not open /dev/mem..." << endl;
        exit(1);
    }
    // Get a mapping from physical addresses to virtual addresses
    pBase = (char *) mmap(NULL, bridgeSpan, (PROT_READ | PROT_WRITE), MAP_SHARED, fd,
bridgeBase);
    if (pBase == MAP_FAILED) // check for errors
    {
        cout << "ERROR: mmap() failed..." << endl;
        close(fd); // close memory before exiting
        exit(1); // Returns 1 to the operating system;
    }
}

DE1SoCfpga::~DE1SoCfpga() {
    if (munmap(pBase, LW_BRIDGE_SPAN) != 0) {
        cout << "ERROR: munmap() failed..." << endl;
        exit(1);
    }
    close(fd); // close memory
}

// writes the given value to the given mapped address
void DE1SoCfpga::RegisterWrite(unsigned intoffset, int value) {
    *(volatile unsigned int *) (pBase + intoffset) = value;
}

// reads the value at the given mapped address
int DE1SoCfpga::RegisterRead(unsigned intoffset) {
    return *(volatile unsigned int *) (pBase + intoffset);
}
```

Keypad.cpp

```
#include "Keypad.h"

#include <iostream>

using namespace std;

Keypad::Keypad() = default;

Keypad::~Keypad() = default;

// gets the key being currently pressed
```

```
int Keypad::read() {
    RegisterWrite(DIRECTION_REG_ADDR, ~0x6A00);

    int data[4][4];

    for (int row = 0; row < 4; row++) {
        RegisterWrite(DATA_REG_ADDR, 1 << ROW_VALS[row]);
        for (int col = 0; col < 4; col++) {
            data[row][col] = ((unsigned int) RegisterRead(DATA_REG_ADDR) >>
COL_VALS[col]) & 1;
        }
    }

    for (int row = 0; row < 4; row++) {
        for (int col = 0; col < 4; col++) {
            if (data[row][col] == 0) {
                for (int i = 0; i < 4; i++) {
                    if (data[i][col] == 1) {
                        return 4 * i + col + 1;
                    }
                }
            }
        }
    }

    return 0;
}
```

Line.cpp

```
#include <algorithm>
#include <iostream>
#include "Line.h"

using namespace std;

// creates a line with the given vectors as endpoints
Line::Line(Vector2D *p1, Vector2D *p2) {
    this->p1 = *p1;
    this->p2 = *p2;
}

Line::~Line() = default;

// translates the vector by the given vector
void Line::add(Vector2D *other) {
    p1.add(other);
    p2.add(other);
}

// translates the vector by the negative of the given vector
void Line::sub(Vector2D *other) {
    p1.sub(other);
    p2.sub(other);
}

// scales the vector by a constant factor
void Line::mult(double k){
    p1.mult(k);
    p2.mult(k);
}

// rotates the vector by a given angle
void Line::rotate(double angle) {
    p1.rotate(angle);
    p2.rotate(angle);
}

// return whether this line is colliding with the given line
bool Line::isColliding(Line *other) {
    // Using an algorithm like the one found here:
https://gamedev.stackexchange.com/questions/26004/how-to-detect-2d-line-on-line-
collision

    double denominator = ((p2.x - p1.x) * (other->p2.y - other->p1.y)) - ((p2.y -
p1.y) * (other->p2.x - other->p1.x));
    double numerator1 =
            ((p1.y - other->p1.y) * (other->p2.x - other->p1.x)) - ((p1.x - other-
>p1.x) * (other->p2.y - other->p1.y));
    double numerator2 = ((p1.y - other->p1.y) * (p2.x - p1.x)) - ((p1.x - other-
>p1.x) * (p2.y - p1.y));

    if (denominator == 0) return numerator1 == 0 && numerator2 == 0;

    double r = numerator1 / denominator;
    double s = numerator2 / denominator;

    return (r >= 0 && r <= 1) && (s >= 0 && s <= 1);
}

// draws this line to the given screen
void Line::draw(Screen *s, bool isWhite) {
    // Using the Bresenham line-drawing algorithm found here:
https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

    int x0 = p1.x, x1 = p2.x;
    int y0 = p1.y, y1 = p2.y;
    int dx = abs(x1 - x0);
    int sx = x0 < x1 ? 1 : -1;
    int dy = -abs(y1 - y0);
    int sy = y0 < y1 ? 1 : -1;
    int err = dx + dy;
    int e2;

    while (true) {
        s->writePixel(x0, y0, isWhite);
        if (x0 == x1 && y0 == y1) break;
        e2 = 2 * err;
        if (e2 >= dy) {
            err += dy;
            x0 += sx;
        }
        if (e2 <= dx) {
            err += dx;
            y0 += sy;
        }
    }
}
```

LoIntervals.cpp

```
#include "LoIntervals.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

Interval::Interval() = default;

Interval::Interval(double *pos, bool isBegin, int shapeIndex) {
    this->pos = pos;
    this->isBegin = isBegin;
    this->shapeIndex = shapeIndex;
}

// compares intervals by comparing their positions
bool Interval::operator<(const Interval &i2) const {
    return *this->pos < *i2.pos;
}

MtLoInterval::MtLoInterval() = default;

// return whether the list is empty
bool MtLoInterval::empty() {
    return true;
}

// returns the size of the list
int MtLoInterval::size() {
    return 0;
}

// returns the element at the given index
Interval *MtLoInterval::at(int index) {
    cout << "Error: index out of bounds." << endl;
    exit(1);
}
```

```
// returns the list with the element at the given index removed
LoIntervals *MtLoInterval::erase(int index) {
    cout << "Error: index out of bounds." << endl;
    exit(1);
}


ConsoLoInterval::ConsoLoInterval(Interval *first, LoIntervals *rest) {
    this->first = first;
    this->rest = rest;
}

// return whether the list is empty
bool ConsoLoInterval::empty() {
    return false;
}

// returns the size of the list
int ConsoLoInterval::size() {
    return 1 + this->rest->size();
}

// returns the element at the given index
Interval *ConsoLoInterval::at(int index) {
    if (index == 0)
        return first;
    return this->rest->at(index - 1);
}

// returns the list with the element at the given index removed
LoIntervals *ConsoLoInterval::erase(int index) {
    if (index == 0)
        return rest;
    return new ConsoLoInterval(this->first, this->rest->erase(index - 1));
}
```

Polygon.cpp

```
#include <stdlib.h>
#include <iostream>
#include "Polygon.h"

// initializes a polygon with all of the necessary information, called by
constructors
void Polygon::initPolygon(Vector2D *pos, Vector2D *vel, double angle, double
angleVel, double inv_mass,
                          vector<Line> *lines, int id) {
    this->pos = *pos;
    this->vel = *vel;
    this->angle = angle;
    this->angleVel = angleVel;
    this->inv_mass = inv_mass;
    this->aabbMin = *(new Vector2D());
    this->aabbMax = *(new Vector2D());

    this->lines = lines;

    this->id = id;
}

Polygon::Polygon() = default;

Polygon::Polygon(Vector2D *pos, Vector2D *vel, double angle, double angleVel, double
inv_mass, vector<Line> *lines,
                 int id) {
    initPolygon(pos, vel, angle, angleVel, inv_mass, lines, id);
}

Polygon::Polygon(Vector2D *pos, Vector2D *vel, double angle, double angleVel, double
inv_mass, int type, int id) {
    vector<Vector2D> *points = new vector<Vector2D>;
    vector<Line> *lines = new vector<Line>;

/* 1 - []
 * 2 - *
 * 3 - X
 * 4 - L
 * 5 - T
 * 6 - V
 * 7 - triangle
 * 8 - pentagon
 * 9 - hexagon
 * 10 - stick
 */
    switch (type) {
        case 1:
            points->push_back(*(new Vector2D(25, 25)));
            points->push_back(*(new Vector2D(-25, 25)));
            points->push_back(*(new Vector2D(-25, -25)));
            points->push_back(*(new Vector2D(25, -25)));
            break;
        case 2:
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 10)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 10)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 10)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 10)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 10)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).rotate(0.6283185 * i);
            }
            break;
        case 3:
            points->push_back(*(new Vector2D(-5, 25)));
            points->push_back(*(new Vector2D(5, 25)));
            points->push_back(*(new Vector2D(5, 5)));
            points->push_back(*(new Vector2D(25, 5)));
            points->push_back(*(new Vector2D(25, -5)));
            points->push_back(*(new Vector2D(5, -5)));
            points->push_back(*(new Vector2D(5, -25)));
            points->push_back(*(new Vector2D(-5, -25)));
            points->push_back(*(new Vector2D(-5, -5)));
            points->push_back(*(new Vector2D(-25, -5)));
            points->push_back(*(new Vector2D(-25, 5)));
            points->push_back(*(new Vector2D(-5, 5)));
            break;
        case 4:
            points->push_back(*(new Vector2D(-25, -25)));
            points->push_back(*(new Vector2D(-25, 25)));
            points->push_back(*(new Vector2D(-15, 25)));
            points->push_back(*(new Vector2D(-15, -15)));
            points->push_back(*(new Vector2D(0, -15)));
            points->push_back(*(new Vector2D(0, -25)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).add(new Vector2D(10, 11));
            }
            break;
        case 5:
            points->push_back(*(new Vector2D(-25, 25)));
            points->push_back(*(new Vector2D(25, 25)));
            points->push_back(*(new Vector2D(25, 15)));
            points->push_back(*(new Vector2D(5, 15)));
            points->push_back(*(new Vector2D(5, -25)));
            points->push_back(*(new Vector2D(-5, -25)));
            points->push_back(*(new Vector2D(-5, 15)));
            points->push_back(*(new Vector2D(-25, 15)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).add(new Vector2D(0, -13));
            }
            break;
        case 6:
            points->push_back(*(new Vector2D(-25, -25)));
            points->push_back(*(new Vector2D(-25, 25)));
            points->push_back(*(new Vector2D(-15, 25)));
            points->push_back(*(new Vector2D(-15, -15)));
            points->push_back(*(new Vector2D(25, -15)));
            points->push_back(*(new Vector2D(25, -25)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).add(new Vector2D(9, 9));
            }
            break;
        case 7:
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->at(1).rotate(2.0944);
            points->at(2).rotate(-2.0944);
            break;
        case 8:
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).rotate(1.25664 * i);
            }
            break;
        case 9:
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));
            points->push_back(*(new Vector2D(0, 25)));

            for (int i = 0; i < points->size(); i++) {
                points->at(i).rotate(1.0472 * i);
            }
            break;
        case 10:
            points->push_back(*(new Vector2D(-5, 25)));
            points->push_back(*(new Vector2D(5, 25)));
            points->push_back(*(new Vector2D(5, -25)));
            points->push_back(*(new Vector2D(-5, -25)));
            break;
        default:
            cout << "Polygons must be created with a type between 1 and 10." << endl;
            exit(1);
    }

    // put lines in the vector of lines that connect all of the poitns that were just
created
    for (int i = 0; i < points->size() - 1; i++) {
        lines->push_back(*(new Line(&(points->at(i)), &(points->at(i + 1)))));
    }
    lines->push_back(*(new Line(&(points->at(points->size() - 1)), &(points-
>at(0)))));

    // initialize all the lines to be in the right position and at the right angle
    for (int i = 0; i < lines->size(); i++) {
        lines->at(i).mult(1 / inv_mass);
        lines->at(i).rotate(angle);
        lines->at(i).add(pos);
    }

    initPolygon(pos, vel, angle, angleVel, inv_mass, lines, id);
}

Polygon::~Polygon() {
    delete lines;
}

// returns whether this Polygon is colliding with the given one
bool Polygon::isColliding(Polygon *other) {
    int len1 = abs(this->pos.x - other->pos.x);
    int len2 = abs(this->pos.y - other->pos.y);

    Vector2D thisSize = this->aabbMax.copy();
    thisSize.sub(&this->aabbMin);

    Vector2D otherSize = other->aabbMax.copy();
    otherSize.sub(&other->aabbMin);

    // AABB algorithm outlined in IEEE paper
    if (len1 <= (thisSize.x + otherSize.x) / 2 &&
        len2 <= (thisSize.y + otherSize.y) / 2) {

        // check if any of the lines are colliding with the other one's lines
        for (int i = 0; i < lines->size(); i++) {
            for (int j = 0; j < other->lines->size(); j++) {
                if (lines->at(i).isColliding(&(other->lines->at(j)))) return true;
            }
        }
    }

    return false;
}

// updates the AABB, position, angle, and lines of this Polygon, once per tick
void Polygon::update(Screen *s) {
    draw(s, false);

    pos.add(&vel);
    angle += angleVel;

    for (int i = 0; i < lines->size(); i++) {
        lines->at(i).sub(&pos);

        // line must be translated to the origin, because rotation occurs about the
origin

        lines->at(i).rotate(angleVel);

        lines->at(i).add(&pos);
        lines->at(i).add(&vel);
    }

    // calculates the new AABB
    int xMax, yMax;
    int xMin = xMax = lines->at(0).p1.x;
    int yMin = yMax = lines->at(0).p1.y;

    for (int i = 1; i < lines->size(); i++) {
        Vector2D point = lines->at(i).p1;
        if (point.x < xMin) xMin = point.x;
        if (point.y < yMin) yMin = point.y;
        if (point.x > xMax) xMax = point.x;
        if (point.y > yMax) yMax = point.y;
    }
    aabbMin.x = xMin;
    aabbMin.y = yMin;
    aabbMax.x = xMax;
    aabbMax.y = yMax;
}

// draws this Polygon to the given screen
void Polygon::draw(Screen *s, bool isWhite) {
    pos.draw(s, isWhite);
    for (int i = 0; i < lines->size(); i++) {
        lines->at(i).draw(s, isWhite);
    }
}

// compares two Polygons by their ids
bool Polygon::operator<(const Polygon &p2) const {
    return this->id < p2.id;
}
```

Screen.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <iostream>

#include "Screen.h"
#include "DE1SoCfpga.h"

using namespace std;

Screen::Screen(DE1SoCfpga *fpga) {
    fd = open("/dev/mem", (O_RDWR | O_SYNC));
    if (fd == -1) // check for errors in opening /dev/mem
    {
        cout << "ERROR: could not open /dev/mem..." << endl;
        exit(1);
    }
    // Get a mapping from physical addresses to virtual addresses
    pBase = (char *) mmap(NULL, SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, fpga-
>RegisterRead(PIXEL_BUF_CTRL_BASE + 0x4));
    if (pBase == MAP_FAILED) // check for errors
    {
        cout << "ERROR: mmap() failed..." << endl;
        close(fd); // close memory before exiting
        exit(1); // Returns 1 to the operating system;
    }

    int res = fpga->RegisterRead(PIXEL_BUF_CTRL_BASE + 0x8);

    screen_x = res & 0xFFFF;

    int db = get_data_bits(fpga->RegisterRead(RGB_RESAMPLER_BASE) & 0x3F);
/* check if resolution is smaller than the standard 320 x 240 */
    res_offset = (screen_x == 160) ? 1 : 0;
/* check if number of data bits is less than the standard 16-bits */
    col_offset = (db == 8) ? 1 : 0;

    x_factor = 0x1 << (res_offset + col_offset);
    y_factor = 0x1 << (res_offset);

    white = resample_rgb(db, 0xFFFFFF);

    speaker = 0;
}

Screen::~Screen() {
    if (munmap(pBase, SPAN) != 0) {
        cout << "ERROR: munmap() failed..." << endl;
        exit(1);
    }
    close(fd); // close memory
}

int Screen::resample_rgb(int num_bits, int color) {
    if (num_bits == 8) {
        color = (((color >> 16) & 0x000000E0) | ((color >> 11) & 0x0000001C) |
                 ((color >> 6) & 0x00000003));
        color = (color << 8) | color;
    } else if (num_bits == 16) {
        color = (((color >> 8) & 0x0000F800) | ((color >> 5) & 0x000007E0) |
                 ((color >> 3) & 0x0000001F));
    }
    return color;
}

int Screen::get_data_bits(int mode) {
    switch (mode) {
        case 0x0:
            return 1;
        case 0x7:
            return 8;
        case 0x11:
            return 8;
        case 0x12:
            return 9;
        case 0x14:
            return 16;
        case 0x17:
            return 24;
        case 0x19:
            return 30;
        case 0x31:
            return 8;
        case 0x32:
            return 12;
        case 0x33:
            return 16;
        case 0x37:
            return 32;
        case 0x39:
            return 40;
        default:
            return -1;
    }
}

// writes black or white (bool) at the given pixel (x, y)
void Screen::writePixel(int x, int y, bool isWhite) {
    // Uses the operations in the “Video” example found in the manual

    // if within bounds
    if(x >= 0 && x < 320 && y >= 0 && y < 240) {

        x /= x_factor;
        y /= y_factor;

        int pixel_ptr = (y << (10 - res_offset - col_offset)) + (x << 1);
        *(volatile unsigned int *) (pBase + pixel_ptr) = (isWhite ? white : 0);
    }
}

// Uses buffer swapping outlined in the chip’s manual
// Waits until it’s time to update (60 Hz) using the timing feature outlined in the
manual
void Screen::update(DE1SoCfpga *fpga) {
    while((fpga->RegisterRead(PIXEL_BUF_CTRL_BASE + 0xC) & 1) == 1){}
    fpga->RegisterWrite(PIXEL_BUF_CTRL_BASE, 1);
    fpga->RegisterWrite(0x64, 1);
    speaker = ~speaker;
    fpga->RegisterWrite(0x60, speaker);
}
```

```
// writes every pixel black
void Screen::clear() {
    for(int y = 0; y < 240; y++){
        for(int x = 0; x < 320; x++){
            writePixel(x, y, false);
        }
    }
}
```

Speaker.cpp

```
#include "Speaker.h"
#include <iostream>

using namespace std;

Speaker::Speaker(){
    initialvalueLoadMPCore = RegisterRead(MPCORE_PRIV_TIMER_LOAD_OFFSET);
    initialvalueControlMPCore = RegisterRead(MPCORE_PRIV_TIMER_CONTROL_OFFSET);
    initialvalueInterruptMPCore = RegisterRead(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET);
}

Speaker::~Speaker() {
    RegisterWrite(MPCORE_PRIV_TIMER_LOAD_OFFSET, initialvalueLoadMPCore);
    RegisterWrite(MPCORE_PRIV_TIMER_CONTROL_OFFSET, initialvalueControlMPCore);
    RegisterWrite(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET, initialvalueInterruptMPCore);
}

// plays the given tone for the given duration
void Speaker::playTone(double duration, int hz) {
    int lowcount = (int) (200000000 / hz) / 2;

    RegisterWrite(MPCORE_PRIV_TIMER_LOAD_OFFSET, lowcount);

    int entervalue = 0;
    int half_wave_count = duration * hz;

    while (half_wave_count > 0 && (RegisterRead(SW_OFFSET) & 1) != 0){
        if (RegisterRead(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET) != 0) {
            cout << half_wave_count << endl;
            RegisterWrite(MPCORE_PRIV_TIMER_INTERRUPT_OFFSET, 1); // reset timer flag
            entervalue = entervalue^0x1;
            RegisterWrite(DATA_REG_ADDR, entervalue);

            half_wave_count--;
        }
    }
}
```

Vector2D.cpp

```
#include <cmath>
#include <iostream>
#include <complex>
#include "Vector2D.h"

using namespace std;

Vector2D::Vector2D() {
    x = 0;
    y = 0;
}

Vector2D::Vector2D(double x, double y) {
    this->x = x;
    this->y = y;
}

Vector2D::~Vector2D() = default;

// translates the vector by the given vector
void Vector2D::add(Vector2D *other) {
    x += other->x;
    y += other->y;
}

// translates the vector by the negative of the given vector
void Vector2D::sub(Vector2D *other) {
    x -= other->x;
    y -= other->y;
}

// scales the vector by a constant factor
void Vector2D::mult(double k) {
    x *= k;
    y *= k;
}

typedef complex<double> point;

// rotates the vector by a given angle
void Vector2D::rotate(double ang) {

    // uses the imaginary plane for speed and convenience
    // also trigonometry was shrinking the vectors for some odd reason
    point p(x, y);

    point pNew = (p) * polar(1.0, ang);

    x = pNew.real();
    y = pNew.imag();
}

// draws this line to the given screen
```

```
void Vector2D::draw(Screen *s, bool isWhite) {
    s->writePixel(x, y, isWhite);
}

// copies this vector to a new vector
Vector2D Vector2D::copy() {
    return Vector2D(x, y);
}
```

main.cpp

```
#include <iostream>
//#include <stdlib.h>
#include <ctime>
#include <set>
#include <algorithm>
#include "DE1SoCfpga.h"
#include "Screen.h"
#include "Keypad.h"
#include "Polygon.h"
#include "Speaker.h"
#include "CollisionPair.h"
#include "LoIntervals.h"

/* TODO
   make presentation
   make video
   submit
 */

// Cited IEEE paper for collision detection: https://ieeexplore-ieee-
org.ezproxy.neu.edu/document/5363537
// Collision handling:
http://www.kuffner.org/james/software/dynamics/mirtich/mirtichThesis.pdf

// returns the polygon that the given polygon is colliding with
Polygon *collidingWithAnything(Polygon *shape, Polygon *shapes, int numShapes) {
    for (int i = 0; i < numShapes; i++) {
        if (&shapes[i] == shape) continue;
        if (shape->isColliding(&shapes[i])) return &shapes[i];
    }

    return NULL;
}

// makes a new polygon with random attributes besides mass and type
Polygon *makeRandomPolygon(double mass, int type, Screen *s, int id) {
//    Polygon *p = new Polygon(
//            new Vector2D(((double) 320 * rand() / RAND_MAX), ((double) 240 * rand()
/ RAND_MAX)),
//            new Vector2D(((double) 2 * rand() / RAND_MAX) - 1, ((double) 2 * rand()
/ RAND_MAX) - 1),
//            ((double) 6.28 * rand() / RAND_MAX), ((double) 0.05 * rand() /
RAND_MAX), 1 / mass, type, id);
    Polygon *p = new Polygon(
            new Vector2D(((double) 320 * rand() / RAND_MAX), ((double) 240 * rand() / 
RAND_MAX)),
            new Vector2D(((double) 2 * rand() / RAND_MAX) - 1, ((double) 2 * rand() / 
RAND_MAX) - 1),
            0, 0, 1 / mass, type, id);
    p->update(s); // give it a proper AABB
    return p;
}

// compares two CollisionPairs (overriding operator< wasn't working for some reason)
struct cpCmp {
    bool operator()(const CollisionPair *a, const CollisionPair *b) {
        if (a->p1->id != b->p1->id) {
            return a->p1->id < b->p1->id;
        } else {
            return a->p2->id < b->p2->id;
        }
    }
};

// the core of the optimized algorithm, returns a set of CollisionPairs that are 
colliding in the given vector of intervals
set<CollisionPair *, cpCmp> *getPossibleCollided(vector<Interval> 
*collisionIntervals, Polygon *shapes) {
    set<CollisionPair *, cpCmp> *possibleCollided = new set<CollisionPair *, 
cpCmp>();

    LoIntervals *active = new MtLoInterval();

    sort(collisionIntervals->begin(), collisionIntervals->end());

    for (int i = 0; i < collisionIntervals->size(); i++) {
        if (collisionIntervals->at(i).isBegin) {
            if (!active->empty()) {
                for (int j = 0; j < active->size(); j++) {
                    possibleCollided->insert(new 
CollisionPair(&shapes[collisionIntervals->at(i).shapeIndex],
                                                               &shapes[active->at(j)-
>shapeIndex]));
                }
            }

            active = new ConsoLoInterval(&collisionIntervals->at(i), active);
        } else {
            int indexOfPair = -1;

            for (int j = 0; j < active->size(); j++) {
                if (collisionIntervals->at(i).shapeIndex == active->at(j)-
>shapeIndex) {
                    indexOfPair = j;
                    break;
                }
            }

            active = active->erase(indexOfPair);
        }
    }

    return possibleCollided;
}

// initializes the edges' lines
void initEdges(vector<Line> *edge1, vector<Line> *edge2, vector<Line> *edge3,
vector<Line> *edge4) {
    edge1->push_back(Line(new Vector2D(-1, 0), new Vector2D(-1, 240 - 1)));
    edge1->push_back(Line(new Vector2D(-20, 0), new Vector2D(-20, 240 - 1)));
    edge1->push_back(Line(new Vector2D(-1, 0), new Vector2D(-20, 0)));
    edge1->push_back(Line(new Vector2D(-1, 240 - 1), new Vector2D(-20, 240 - 1)));

    edge2->push_back(Line(new Vector2D(0, 241), new Vector2D(320 - 1, 241)));
    edge2->push_back(Line(new Vector2D(0, 260), new Vector2D(320 - 1, 260)));
    edge2->push_back(Line(new Vector2D(0, 241), new Vector2D(0, 260)));
    edge2->push_back(Line(new Vector2D(320 - 1, 241), new Vector2D(320 - 1, 260)));

    edge3->push_back(Line(new Vector2D(321, 240), new Vector2D(321, 1)));
    edge3->push_back(Line(new Vector2D(340, 240), new Vector2D(340, 1)));
    edge3->push_back(Line(new Vector2D(321, 240), new Vector2D(340, 240)));
    edge3->push_back(Line(new Vector2D(321, 1), new Vector2D(340, 1)));

    edge4->push_back(Line(new Vector2D(320, -1), new Vector2D(1, -1)));
    edge4->push_back(Line(new Vector2D(320, -20), new Vector2D(1, -20)));
    edge4->push_back(Line(new Vector2D(320, -1), new Vector2D(320, -20)));
    edge4->push_back(Line(new Vector2D(1, -1), new Vector2D(1, -20)));
}

int main() {
    cout << "Starting..." << endl;

    // how many shapes can be processed
    const int MAX_SHAPES = 250;

    DE1SoCfpga *fpga = new DE1SoCfpga();
    Screen *screen = new Screen(fpga);
    Keypad *keypad = new Keypad();

    // keep track of x and y intervals
    vector<Interval> *xCollisionIntervals = new vector<Interval>();
    vector<Interval> *yCollisionIntervals = new vector<Interval>();

    // for finding the average time of calculation per tick
    double totalTime = 0;
    int ticksRecorded = 0;

    srand(time(NULL));
    Speaker *speaker = new Speaker();

    vector<Line> *edge1 = new vector<Line>();
    vector<Line> *edge2 = new vector<Line>();
    vector<Line> *edge3 = new vector<Line>();
    vector<Line> *edge4 = new vector<Line>();

    initEdges(edge1, edge2, edge3, edge4);

    // edges are Polygons as well, and we always start with them
    Polygon shapes[MAX_SHAPES] = {
            Polygon(new Vector2D(-10, 120), new Vector2D(), 0, 0, 0, edge1, 0),
            Polygon(new Vector2D(160, 250), new Vector2D(), 0, 0, 0, edge2, 1),
            Polygon(new Vector2D(330, 120), new Vector2D(), 0, 0, 0, edge3, 2),
            Polygon(new Vector2D(160, -10), new Vector2D(), 0, 0, 0, edge4, 3)
    };
    int numShapes = 4;

    // add the edges' AABBs to the intervals
    for (int i = 0; i < numShapes; i++) {
        xCollisionIntervals->push_back(Interval(&shapes[i].aabbMin.x, true, i));
        xCollisionIntervals->push_back(Interval(&shapes[i].aabbMax.x, false, i));
        yCollisionIntervals->push_back(Interval(&shapes[i].aabbMin.y, true, i));
        yCollisionIntervals->push_back(Interval(&shapes[i].aabbMax.y, false, i));
        shapes[i].update(screen);
    }

    double mass = 1;
    int keyPressed;
    int oldKeyPressed = -1;

    screen->clear();

    keyPressed = keypad->read();
    while (keyPressed != 16) { // Pressing 'D' ends the simulation
        if (oldKeyPressed != keyPressed && keyPressed != 0) { // if a new key is
being pressed
            if (keyPressed >= 1 && keyPressed <= 10) { // if it's an insertion key
                int numInsertionTries = 0;

                do { // try to insert a new random polygon
                    shapes[numShapes] = *makeRandomPolygon(mass, keyPressed, screen,
numShapes);
                    numInsertionTries++;
                } while (numInsertionTries < 100 &&
collidingWithAnything(&shapes[numShapes], shapes, numShapes + 1));
                // if the inserted polygon is colliding, try again up to 100 times

                // if we couldn't insert it
                if (numInsertionTries >= 100) {
                    cout << "Screen full. Insertion failed after " <<
numInsertionTries << "tries." << endl;
```

```
                } else {
                    cout << "New type " << keyPressed << " shape inserted with mass "
<< mass << endl;
                    // add the new shape's AABB to the intervals
                    xCollisionIntervals-
>push_back(Interval(&shapes[numShapes].aabbMin.x, true, numShapes));
                    xCollisionIntervals-
>push_back(Interval(&shapes[numShapes].aabbMax.x, false, numShapes));
                    yCollisionIntervals-
>push_back(Interval(&shapes[numShapes].aabbMin.y, true, numShapes));
                    yCollisionIntervals-
>push_back(Interval(&shapes[numShapes].aabbMax.y, false, numShapes));
                    numShapes++;
                }
            } else if (keyPressed == 11) { // '9' to decrease mass
                mass *= 0.9;
                cout << "Mass decreased to " << mass << endl;
            } else if (keyPressed == 12) { // 'C' to increase mass
                mass *= 1.11111;
                cout << "Mass increased to " << mass << endl;
            }
        }

        // update and draw every shape
        for (int i = 0; i < numShapes; i++) {
            shapes[i].update(screen);
            shapes[i].draw(screen, true);
        }

        clock_t start = clock();
        // start timing

        // IMPROVED ALGORITHM:
        set<CollisionPair *, cpCmp> *possibleXCollided =
getPossibleCollided(xCollisionIntervals, shapes);
        set<CollisionPair *, cpCmp> *possibleYCollided =
getPossibleCollided(yCollisionIntervals, shapes);

        vector<CollisionPair *> possibleCollided;

        set_intersection(possibleXCollided->begin(), possibleXCollided->end(),
possibleYCollided->begin(),
                         possibleYCollided->end(), back_inserter(possibleCollided),
cpCmp());

        vector<CollisionPair *> *collided = new vector<CollisionPair *>();

        for (int i = 0; i < possibleCollided.size(); i++) {
            CollisionPair *cp = possibleCollided.at(i);
            if (cp->p1->isColliding(cp->p2)) {
                collided->push_back(cp);
            }
        }
        // ORIGINAL ALGORITHM:
//        for (int i = 0; i < numShapes; i++) {
//            for (int j = 0; j < numShapes; j++) {
//                if (&shapes[i] == &shapes[j]) continue;
//                if (shapes[i].isColliding(&shapes[j])) collided->push_back(new
CollisionPair(&shapes[i], &shapes[j]));
//            }
//        }

        // remove duplicate CollisionPairs
        sort(collided->begin(), collided->end());
        collided->erase(unique(collided->begin(), collided->end(),
CollisionPair::equals), collided->end());


        vector<CollisionPair *> *newCollided = new vector<CollisionPair *>();
        for (int i = 0; i < collided->size(); i++) {
            newCollided->push_back(collided->at(i));
            newCollided->push_back(new CollisionPair(collided->at(i)->p2, collided-
>at(i)->p1));
        }

        // stop timing
        clock_t end = clock();

        // we've now timed another tick
        totalTime += double(end - start) / CLOCKS_PER_SEC;
        ticksRecorded++;

        if (!collided->empty()) speaker->playTone(0.016, 880); // play a short beep
if a collision occurred


        // handle all collisions
        for (int i = 0; i < collided->size(); i++) {
            collided->at(i)->handleCollision();
        }

        // write to the screen
        screen->update(fpga);

        // get the new keypress
        oldKeyPressed = keyPressed;
        keyPressed = keypad->read();
    }

    cout << "Average collision calculation time: " << totalTime / ticksRecorded <<
endl;

    screen->clear();

    delete speaker;
    delete keypad;
    delete screen;
    delete fpga;

    cout << "Ending" << endl;

    exit(0);
}
```

References

[1] Prof. Julius Marpaung, "*Lab Report Guide*", Northeastern University, January 6 2020.

[2] Y. Tan, C. Liu and Y. Yu, "The Design of Collision Detection Algorithm in 2D Grapple Games," *2009 International Conference on Information Engineering and Computer Science*, Wuhan, China, 2009, pp. 1-4, doi: 10.1109/ICIECS.2009.5363537.

[3] https://gamedev.stackexchange.com/questions/26004/how-to-detect-2d-line-on-line-collision

[4] "Bresenham's Line Generation Algorithm." GeeksforGeeks, GeeksforGeeks, 22 Mar. 2021, www.geeksforgeeks.org/bresenhams-line-generation-algorithm/.

[5] Mirtich, Brian Vincent. "Impulse-Based Dynamic Simulation of Rigid Body System." *University of California at Berkeley*, University of California at Berkeley, 1996.