Final Report of CSE 4/590 Project: Collision Detection of 3D Printer

KAUSHIK RAMASUBRAMANIAN MITALI VIJAY BHIWANDE TEJASVI BALARAM SANKHE

Abstract — This project is related to collision detection of G-codes for a 3D printer. 3D printing has become a mainstream manufacturing process in various industry fields. This refers to a process by which 3D digital design data is used to build up a 3D physical object in layers by depositing material. G-code, being the programming language to control the nozzle movement of a 3D printer. But, sometimes the malicious G-code may lead to collision between printer nozzle and printed object on the platform to damage the printer. This project aimed to detect this malicious G-code. There are various methods for collision detection, we have used line segment intersection method to perform this collision detection. The project detects malicious lines for each 3D plane by comparing two lines at a time and determining if the line is malicious or non-malicious.

I. Project Background and Introduction

Introduction:

Additive manufacturing (AM), also known as 3D printing, has been becoming a mainstream manufacturing process in various industry fields. Specifically, it refers to a process by which 3D digital design data (in the cyber domain) is used to build up a 3D physical object in layers by depositing material (in the physical domain). The G-code is the programming language to control the nozzle movement of a 3D printer. However, some malicious G-code may induce collision between the printer nozzle and printed object on the platform to damage the printer. In this project, we aim to detect all the malicious G-code in the given printing files.

Background of G-code:

G-code is a common name for the programming language that controls NC and CNC machine tools. Developed by the Electronic Industries Alliance in the early 1960s, a final revision was approved in February 1980 as RS274D. Machine control manufactures extended the standard G-Code to include storage variables, more so-called canned cycles, go to, if-then-else flow control, AND,OR,NOR logic, and even so-called conversational programming (select operation, select size, select depth, select direction, speeds and feeds, push go; the machine generates the G-Code), all to the goal of increasing production, decreasing cycle time, improving cost-effectiveness.

Machines were constructed that could have their working ways altered by a punched paper tape, which was fed into a reader, which in turn controlled the machine operation. As technology progressed, the paper tape was replaced with magnetic media, but the basic operations required to produce the same candlestick really did not change; the same operations had to occur, in the same order; only the timeframe for the production of the product had shifted from hours to seconds. The production of G-Code reflected that process; the produced code had little need of more than the ability to loop operations and utilize subroutines. Typical operations were:

- Re-iterate a cutting / drilling / boring / milling procedure until final depth was reached.
- Utilize a preexisting movements (subroutine) when a particular point in the main program was reached (move the machine to a specific spot, trigger the drilling subroutine, move to another spot, trigger the drilling routine again).

It is a language in which people tell computerized machine tools how to make something. The "how" is defined by instructions on where to move, how fast to move, and what path to follow. The most common situation is that, within a machine tool, a cutting tool is moved according to these instructions through a toolpath and cuts away material to leave only the finished work piece. The same concept also extends to noncutting tools such as forming or burnishing tools, photo plotting, additive methods such as 3D printing, and measuring instruments.

Few examples of G-codes is given in the below table:

G-code	Description	G-code	Description
G00	Rapid Linear Interpolation	G12	Clockwise Circle With Entrance And Exit Arcs
G01	Linear Interpolation	G13	Counter Clockwise Circle With Entrance And Exit Arcs
G02	Clockwise Circular Interpolation	G17	X-Y Plane Selection
G03	Counter Clockwise Circular Interpolation	G28	Return To Reference Point
G04	Dwell	G34	Special Fixed Cycle (Bolt Hole Circle)
G05	High Speed Machining Mode	G40	Tool Radius Compensation Cancel
G10	Offset Input By Program	G41	Tool Radius Compensation Left

Table 1: Few Examples of G-codes

G-Code is the most popular programming language used for programming CNC machinery. Some G words alter the state of the machine so that it changes from cutting straight lines to cutting arcs. Other G words cause the interpretation of numbers as millimeters rather than inches. Some G words set or remove tool length or diameter offsets.

II. DESCRIPTION OF DESIGN METHOD

We have used **line segment intersection method** as the design method in the project.

Testing for intersection between segments is a bit trickier than finding intersections between infinite lines on the form y = ax + b, as they are a bit easier to solve symbolically.

When testing for intersection between two line segments, there are five cases to consider. The code will go through these cases one by one.

- The line segments are collinear and overlapping, meaning that they share more than one point.
- The line segments are collinear but not overlapping, sort of "chunks" of the same line.
- The line segments are parallel and non-intersecting.
- The line segments have a single point of intersection.
- The line segments do not intersect.

Design:

a) Reading rows from the file:

- The line intersection technique requires us to determine if the two line segments in a given plane intersect or not.
- In order to do this, we read the gcode.txt file row-wise using the test bench Verilog code, which gives us one-line segment.
- In Collision Detect module we store the line segments, in a 2D array. At every positive edge of clock, we check if the in_val == 1'b1, and then store the values.

b) Line Intersection:

- For determining if the line segments intersect, we used the method described in the following link (http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/).

The method we used for determining if two segments intersect uses the concept of orientation of an ordered triplets of points in a plane, it can be counterclockwise, clockwise or collinear.

Two segments (p1, q1) and (p2, q2) intersect if and only if one of the following two conditions is verified

1. General Case:
– (p1, q1, p2) and (p1, q1, q2) have different orientations and
– (p2, q2, p1) and (p2, q2, q1) have different orientations.
In this case the two segments have one intersection point.

2. Special Case
– (p1, q1, p2), (p1, q1, q2), (p2, q2, p1), and (p2, q2, q1) are all collinear and
– the x-projections of (p1, q1) and (p2, q2) intersect
– the y-projections of (p1, q1) and (p2, q2) intersect
In this case, there are one or more intersection points lying on each segments.

We define functions in Verilog module to calculate the orientations and check the special case. For two lines that are being considered we compute orientations for all co-ordinates required in the general and special cases.

Once we have the computed orientations we check for the general and special case to determine if the two line segments intersect.

Duplicate lines are considered as special case since there are multiple intersection points as the two lines completely overlap each other and thus duplicate lines are termed malicious by our code implementation.

- We used the C++ code as the basis for our Verilog code, and defined the required functions and used them accordingly.
- For our code we pass the co-ordinate pairs (x,y) of the required points by referencing the appropriate columns of the 2D array.
- The code at a time checks the incoming line with all the non-malicious lines in the G-code to determine whether it collides with them.
- At this moment we ignore the z coordinate and hence we work in 2-Dimensional plane.
- If the conditions for intersection are being satisfied then, we set the result bit to ‘1’ and the invalid bit to 1 indicating the line is malicious and should not be considered for collisions in future.

- At the end, if the result bit is '1', we assign out_val to 1'b1 and assign the index i.e. row number to lineID.

- For every line segment, we check if the z coordinates are equal.
- If yes, then the code to check if they intersect executes. Hence we make sure lines lying in different planes are not compared to each other.

Proper and Non – Proper Intersection:

There exists a concept of proper and non-proper intersection. A proper intersection is when the two segments share exactly one point and this point lies in the interior of both segments, whereas a non-proper intersection would occur in one of the segment's start or end point. At the moment, the code does not distinguish between these two cases, but a check could be easily added by testing the intersection point for equality against the four input vectors.

Collinearity and Non-Proper Intersection

When two segments are overlapping, e.g. (0, 0->2, 0) and (1, 0->2, 0), we have no meaningful concept of an intersection point, as there in theory are an infinite amount of them. However, consider the two line segments along the x-axis (0, 0->1, 0) and (1, 0 ->2, 0). These two segments have a non-proper intersection in the point (1, 0). If we include non-proper intersections, we actually would have a valid intersection point in this case.

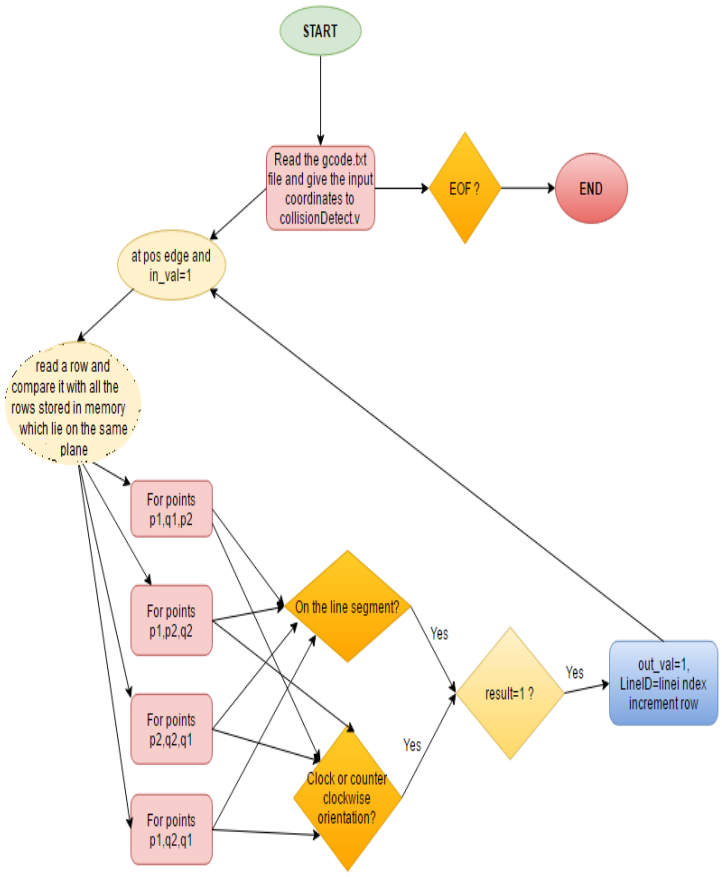

Figure 1: DESIGN FLOW CHART

III. VERILOG SOFTWARE FLOW CHART AND DESCRIPTION

1. Firstly, a row is read from the gcode.txt file containing the coordinates of the points of the line segment and given as an input to the collisionDetect.v by the test_collision_module.v i.e. the test bench.

The row co-ordinates are stored in a 2D array with 7 columns. The first 6 columns contain the start and end x, y, z co-ordinates of the line and the 7th column is the invalid line indicator which is set to '0' by default and changed to '1' if the line intersects with the existing lines.

2. Every time the test bench reads a new line, we check if it intersects with the previous existing non malicious lines, by looping through them and comparing the points for those.

3. As described in the above methodology, we check for the general and special cases of line intersection using different combinations of points for the given two lines.

The incoming line segment is checked with all existing non malicious lines.

According to the onSegment and orientation checks if the lines interest, result bit and the invalid bit is set to 1.

If we consider p1, p2, q1 for orientation check. In the below snippet,

```
input p1,p2,q1,q2,r1,r2;
integer p1,p2,q1,q2,r1,r2;
begin
orientation=(q2-p2)*(r1-q1)-(q1-p1)*(r2-q2);
if(orientation==0)
orientation=0;
else if(orientation>0)
orientation=1;
else if(orientation<0)
orientation=2;
end
```

The highlighted portion is the formula that is used to determine the orientation of the points i.e. collinear (0), clock (1) or counter-clockwise (2).

For special case we check if the points lie on the line segment. The highlighted portion below shows the formula that was used to check the same.

```
function integer onSegment;
input p1,p2,q1,q2,r1,r2;
integer p1,p2,q1,q2,r1,r2;
begin
if(q1<=max(p1,r1) && q1>=min(p1,r1) && q2<=max(p2,r2) && q2>=min(p2,r2))
    onSegment=1;
else
    onSegment=0;
end
endfunction
```

4. After all the required checks for each iteration, if result bit is '1', then we assign the output variables out_val as 1 and line index to lineID. The test bench module writes the lineID to result.txt if out_val=1.

5. The above procedure is repeated for every row in the file till we reach the last row (EOF).

IV. Simulation Result

1. General Case

   Two line segments forming a '+' or 'x' on intersection i.e. the two lines will have one intersection point.

   Considered input:

   3 1 1 3 4 1
   1 1 1 1 5 1
   1 3 1 6 3 1
   1 6 1 5 6 1

   In this case the line 3 (1 3 1 6 3 1) intersects with line 1 (3 1 1 3 4 1) and same can be seen from the simulation result as lineID contains value 011.

Simulation Result:

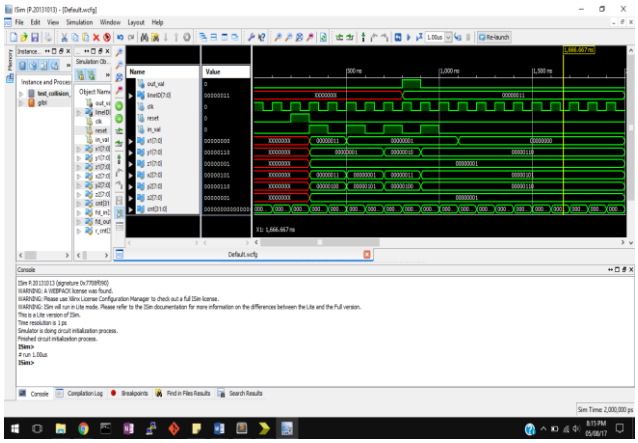

Figure 2: SIMULATION RESULTS FOR GENERAL CASE

2. Boundary Case

   Boundary cases including duplicate lines and end-point intersection such that two line segments form 'L' or 'T' shapes.

   Considered input:

   1 1 1 5 1 1
   3 1 1 3 3 1
   3 4 1 5 4 1
   1 2 1 3 5 1
   1 2 1 3 5 1

   Simulation Result:

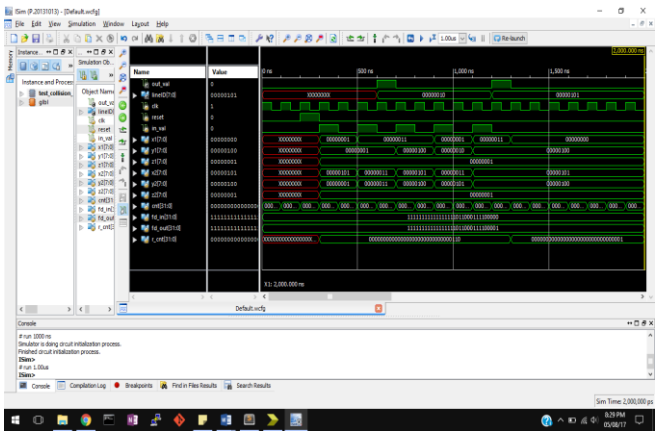

Figure 3: SIMULATION RESULTS FOR BOUNDARY CASE

V. DISCUSSION

There were many things to learn from this project related to the functionality and the implementation of a 3D printer. They are listed below:

- We got to know that a 3D printed object is achieved through an additive process. The additive process works in a manner that an object is created by laying down successive layers of material until the object is created. Each of these layers can be seen as a thinly sliced horizontal cross-section of the eventual object.

- Since we used the line segment intersection method, to detect the layer by layer collision detection. Although, we weren't able to implement the 3D model for the same and restricted the implementation for 2D, we got to know that a 3D line segment intersection is detected if the two lines are coplanar and aren't parallel to each other.

- It also came to our knowledge that, during each positive clock cycle there is only one line segment that is fetched from memory.

- In order to check for line segment intersection, we need to check the orientation of the points and also if the points are collinear.

- We also learnt that the algorithm that was used for 2D cannot be used for 3D, because of the extra Z coordinate.

- Since parts are created additively through 3D printing, they are also limited in size. There 3D printers which are small in size and can be accommodated with a desktop. But large companies need larger 3D printers. These are expensive and also take long time for creation of the object.

- We got to know that 3D printing is a viable option only in the case of prototyping as it's inexpensive and economical way of creating one-run parts for which there's no tooling creation required. On the other hand in terms of manufacturing process, 3D printing isn't a viable option as it objects are made in minutes and not hours.

- While 3D printers have made advances in accuracy in recent years, many of the plastic materials still come with an accuracy disclaimer. For instance, many materials print to either +/- 0.1 mm in accuracy, meaning there is room for error.

- Since 3D printing uses a bottom-up approach, the material of choice is plastic, as it can be deposited in melted layers to form the final part.

- While some of these algorithms for collision detection are simple enough to calculate, it can be a waste of cycles to test every entity with every other entity. Usually games will split collision into two phases, broad and narrow.

- Broad phase should give you a list of entities that could be colliding. This can be implemented with a spatial data structure that will give you a rough idea of where the entity exists and what exist around it. Some examples of spatial data structures are Quad Trees, R-Trees or a Spatial Hash Map.

- Adding collision detection near the end of a project is very difficult. Building a quick collision detection hack near the end of a development cycle will probably ruin the 3D printer as a whole.

- When the collision detection has to be performed on a large number of objects then there are large number of checks to be performed. This slows down the performance.

- One way to speed things up is to reduce the number of checks that have to be made. Two coordinates opposite to each other cannot possibly collide, so there is no need to check the collision between them.

- Quadtree is a data structure which can be used to divide a 2D region into more manageable parts. It's an extended binary tree, but instead of two child nodes it has four.

- In this manner, if a 2D region is divided into four quadrants and the coordinates are located in one of these four quadrants, then the coordinates that are in opposite quadrants won't collide. Therefore, there is no collision detection performed for these coordinates. This saves time and hence performance.

- Bounding boxes (or bounding volumes) are most often a 2D rectangle or 3D cuboid, but other shapes are possible. The bounding diamond, the minimum bounding parallelogram, the convex hull, the bounding circle or bounding ball, and the bounding ellipse have all been tried, but bounding boxes remain the most popular due to their simplicity.

- Bounding boxes are typically used in the early (pruning) stage of collision detection, so that only objects with overlapping bounding boxes need be compared in detail.

- A triangle mesh object is commonly used in 3D body modeling. It is literal, a closed set. Normally the collision function is a triangle to triangle intercept or a bounding shape associated with the mesh.

- A triangle centroid is a center of mass location such that it would balance on a pencil tip. The simulation need only add a centroid dimension to

the physics parameters. Given centroid points in both object and target it is possible to define the line segment connecting these two points.

- We also learnt that in order to detect the collision of the coordinates only additive and multiplicative operations can be performed. Division cannot be used as it leads the model source code cannot be synthesized in that manner.

- Also the use of registers should be limited. Large number of register can reduce the code performance and cannot be synthesized to a hardware implementation.

- Lastly, we had implemented the code for 3D but didn't give us proper results. Thus, we got to know that the algorithm should be designed so that it takes care of all the corner cases for both 2D and 3D.

REFERENCES

http://www.teskolaser.com/gcode_list.html

http://technobauble.ca/computer-numeric-control-cnc/tool-chain/g-code-aka-rs-274/brief-history-of-g-code-numerical-control-and-g-code-complilers

https://www.codeproject.com/Tips/862988/Find-the-Intersection-Point-of-Two-Line-Segments

http://ask.metafilter.com/191187/Understanding-line-segment-intersections-in-2d

http://yourbusiness.azcentral.com/disadvantages-3d-printers-2212.html

https://developer.mozilla.org/en-US/docs/Games/Techniques/2D_collision_detection

https://en.wikipedia.org/wiki/Collision_detection

http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf