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

Amaan Akhtarali Modak, Libing Wu, Shashank Suresh

Abstract— This paper deals with the detection of collisions in a 3D printer. We aim to detect all the malicious G-code in the given printing files, that leads to collision between the printer nozzle and printed object on the platform, to damage the printer. The methods of Verilog design used here is line segment interaction method. We aim to detect all the malicious G-code in the given printing files.

I. Project Background and Introduction

Additive manufacturing is referred to as 3D printing, as it works in a way similar to a laser printer. This technique is used to build a solid object from a series of layers – each of which is printed directly on top of the previous layer. 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 concept of 3D printing is, however, not new. In the mid-1980s, Chuck Hull invented and patented stereolithography (also called solid imaging) when he founded 3D Systems, Inc. Since then, a lot of advances in the technology have been made, including the size of the printers themselves, the type of materials they can use and much more. It all starts with a particular concept. The first stage of 3D printing is designing an original idea with a digital modeling — that is, with the help of computer aided design (CAD) or animation modeling software.

Whichever program the user chooses, they able to create a virtual blueprint of the object they want to print. The program then divides the object into digital cross-sections so that the printer is able to build it layer by layer. The cross-sections act as guides for the printer, so that the object is the exact size and shape the user wants. Both CAD and animation modeling software are WYSIWYG graphics editors — "what you see is what you get."

If the user is not very design-inclined, they can purchase, download or request ready-made designs from sites like Shapeways, Sculpteo or Thingiverse.

Once the user has a completed design, they send it to the 3D printer with the standard file extension STL (for "stereolithography" or "Standard Tessellation Language"). The STL files contain 3D polygons that are sliced up, so that the printer is able to easily digest its information.

The first thing to note is that the 3D printing is characterized as an "additive" manufacturing process, which means that a 3D solid object is constructed by adding material layer by layer. This is in sharp contrast to the regular "subtractive" manufacturing, through which any object is constructed by cutting the (or "machining") raw material into a desired shape.

After the finished design file is sent to the 3D printer, the user can choose a specific material. This, material can be rubber, plastics, paper, polyurethane-like materials, metals and more; depending on the printer.

Printer processes can vary, but the material is usually either sprayed, squeezed or transferred from the printer onto a platform. Then, a 3D printer deposits layer on top of layer of material to create the finished product. This operation can take several hours or days depending on the size and complexity of the object that is printed. On an average, the 3D-printed layer is approximately 100 microns (or micrometers), which is equivalent to 0.1 millimeters.

All of these layers which are different from one another are fused automatically, to create a single 3D object in a DPI resolution. This happens throughout the process.

The G-code is the programming language that is used to control the nozzle movement of a 3D printer. G-code is a language in which people can tell computerized machine tools how to make an object. The "how" is defined by instructions on where to move, how fast to move, and what path to follow. One of the common situations is that, within a machine tool, an adding tool called nozzle, is moved according to these instructions through a toolpath and drops the material to leave only the finished work piece. The concept also extends to non-cutting tools such as forming or burnishing tools, photo-plotting, additive methods such as 3D printing, and measuring instruments.

An example of G-code commands is shown in Figure 1.

G-Code words			
G0	Rapid Linear Motion	G59.2	Select Coordinate System 8
G1	Linear Motion at Feed Rate	G59.3	Select Coordinate System 9
G2	Arc at Feed Rate	G80	Cancel Modal Motion
G3	Arc at Feed Rate	G81	Canned Cycles – drilling
G4	Dwell	G82	Canned Cycles – drilling with dwell
G10	Set Coordinate System Data	G83	Canned Cycles – peck drilling
G17	X-Y Plane Selection	G85	Canned Cycles – boring,no dwell, feed out
G18	Z-X Plane Selection	G86	Canned Cycles – boring, spindle stop, rapid out
G19	Y-Z Plane Selection	G88	Canned Cycles – boring, spindle stop, manual out
G20	Length Unit inches	G89	Canned Cycles – boring, dwell, feed out
G21	Length Unit milimeters	G90	Set Distance Mode Absolute
G28	Return to Home	G91	Set Distance Mode Incremental
G30	Return to Home	G92	Coordinate System Offsets
G53	Move in Absolut Coordinates	G92.1	Coordinate System Offsets
G54	Select Coordinate System 1	G92.2	Coordinate System Offsets
G55	Select Coordinate System 2	G92.3	Coordinate System Offsets
G56	Select Coordinate System 3	G93	Set Feed Rate Mode units/minutes
G57	Select Coordinate System 4	G94	Set Feed Rate Mode inverse time
G58	Select Coordinate System 5	G98	Set Canned Cycle Return Level
G59	Select Coordinate System 6	G99	Set Canned Cycle Return Level
G59.1	Select Coordinate System 7		

Fig. 1.: G-code commands.

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.

II. Description of Design Method

We used Line Segment Intersection method because it's much easier to expand to 3D-version code. This way, we also don't need to worry about the complicated boundary cases in voxels marking method where they have to consider the case that the two lines are so close but not touch together.

We actually were going to use the voxel marking method at the beginning of our working when the project was just released. Because we thought it will be much simpler if we mark each visited voxels and detect if it being visited again later on in the running process. Then we found out that Verilog is not like any of the normal object oriented languages we are used to build program with. It too much to think and too hard to do to store all those voxels and their visited status in memory module.

Then we decide to use line segment intersection method. The main motivation is because that we found professor provided a C++ code to check if two lines are intersected. We then think it should be very easy to translate the C++ to Verilog. We did it, and it works! After a short period of happiness, we found that the C++ code is only for how to detect intersection between two and only two lines. Since the project asked for more, which is to compare a line with many other lines, it drove us crazy to try to figure it out.

We first used the for loop to make one versus many intersection comparison, and we successfully reached the checkpoint 2 requirement just before the deadline, which is detect intersection with 2D cases. However, after we got the grade result, we are disappointed. We didn't get full marks and we even have a little distance towards full marks. We were pretty sure that we passed all 2D test cases though. While we are still confused, we noticed that we mark all the "L" case as not intersected. We thought that "L" won't be consider as an intersection because you have to have some line attaching together if you want to do a perfect closed 3D print work. Nevertheless, we have to take the result to refine our code. After fixing the "L" case misunderstanding, we decided to go for 3D case to waive the final.

Not soon after this try, we gave up. We were not giving up to go for 3D case, but we think we have to give up our code for 2D. There are many aspects that not allow or at least make it much harder to expand it to also detect 3D intersection.

We tried so hard in five days, we did it! The code works for all test cases, but we then found out the code has to be synthesized too. There is only four days to deadline. We are so stressful, and we quickly found that there are so many error while we trying to synthesize the code. We changed the structure again, for example, replace for loops, to make it synthesizable. However, it takes half of the day time to synthesize. We were think we are good but professor lessen the synthesize time requirement, which that we have to finish synthesize in the length of one lecture. We tried hard to lessen the synthesis time to 10 minutes. At the time we are about to celebrate our success. We found out that the code has a small bug in one of the DIY test cases. We went through the code again and again, we then found that we mistyped a point variable value in line 383.

cross_product u_8(p1p2,p1p1,vz);

should be

cross_product u_8(p1p2,q1q2,vz);

It is important to be conscientious especially when design such a complicated program.

Let's know talk about the design in detail with some important part of code.

We have created a lot of modules.

module CollisionDetect

This is the main module. We create the memory modules of storing 63 lines even though we know the maximum number of line is 50 to be safe. We have the variable linecount to count the line number. This is very important because even if we detect the intersection, we need to keep update the linecount. Otherwise, we cannot transfer our design into correct result. The synthetization took long previously because every time we read one line, right after store it into memory, we try to detect if it has intersection with other lines already in memory. We think that if we want to reduce the synthesize time, we have to store all lines into memory first, and then detect all the potential intersections. So at each line read, we store the line into memory without any calculation and detection of intersection.

We used two variable *i* and *j* to indicate two lines. we change a for loop to detect line intersections into an if statement in collision detect module.

module DetectTwoLines

This this module, we detect if two lines are having intersection. It basically has the input of four points. Output 1 means intersection, which 0 means no intersection. The *i-j* for statement mentioned above is rely on this module.

module max2

module min2

These are two modules that we actually did not use at all in our final version of code. They simply judge the greatness of two numbers.

These modules are mandatory for our mathematical formula of detect intersection of line segments.

module vector_p

Vextor_p is a subtraction calculator. Is minus a vector by another.

module vector_v

Vector_v is also a subtraction calculator which invoke vector_p.

module cross_product

Cross product calculated the multiplication of two vectors.

module dot_product

Dot product is to calculate the multiplication of a point to a vector.

module hybrid_product

Hybrid_product take three vectors, first calculate the product and take the result to do a dot product. This is to check if the lines are on the same plane.

module add_vec

This is simply add vectors together.

module is_vertical

This is actually not used in our final version of code. We were think that "T" case means vertical lines, but then we notices it means the end point of one line is lying on the other line. They don't need to be vertical to form a "T" csse.

module on_segment

This take three points and check if two points are on the the line. This replaced is_vertial module after we grasp the true meaning of the "T" case.

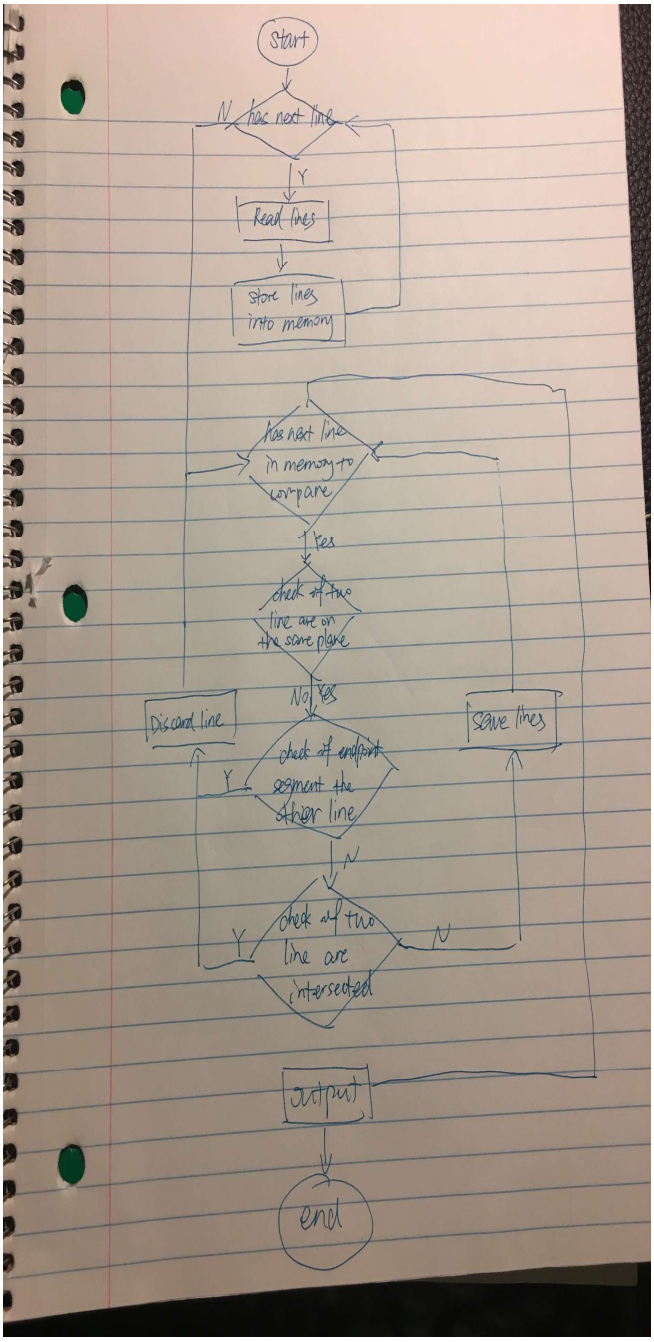
module is_intersect

This module is simply check if two lines are intersect, but it's not simple at all. This is one of the most important modules that contains the core concept of our work. See the comment section of this code. That output the correct result but it won't synthesize successfully. So we eliminate it and write a new is_intersect module.

III. Verilog Software Flow Chart and Description

We start the program by reading the input files line by line, and then storing all lines in memory module first.

Then we read the lines from memory module, anytime we read a line, we compare it with line above. If it's intersected with any of lines above, it got eliminated.



module is_intersect(

.....

vector_v u_1(p1, p2, p1p2);
vector_v u_2(q1, q2, q1q2);
vector_v u_3(p1, q1, p1q1);
vector_v u_4(p1, q2, p1q2);
vector_v u_5(q1, p1, q1p1);
vector_v u_6(q1, p2, q1p2);

hybrid_product u_7(p1p2, q1q2, p1q1, hybrid[0]);

cross_product u_8(p1p2, q1q2, vz);

```
    add_vec u_9(p1, vz, z);
    add_vec u_10(q1, vz, z1);
    vector_v u_11(p1, z, p1z);
    vector_v u_12(q1, z1, q1z1);

    hybrid_product u_13(p1p2, p1q1, p1z, hybrid[1]);
    hybrid_product u_14(p1p2, p1q2, p1z, hybrid[2]);
    hybrid_product u_15(q1q2, q1p1, q1z1, hybrid[3]);
    hybrid_product u_16(q1q2, q1p2, q1z1, hybrid[4]);

    assign delta1 = hybrid[1] * hybrid[2];
    assign delta2 = hybrid[3] * hybrid[4];

    assign o = (hybrid[0] != 64'b0) ? 1'b0 : (($signed(delta1)
<= 0 && $signed(delta2) <= 0) ? 1'b1 : 1'b0);

endmodule
```

The method we used for detecting line intersection is we first check the two z points to see if two lines are on the same level. Then we check the boundary case by check if any of the endpoint of a line segment the other line. If there are, it's the boundary case, so we discard the line and check the next line. If there are not, we use is_intersect module to check if two line are intersecting. If not, we go on to the next line. If there is, we discard the line.

IV. Simulation Result

Based on our implementation of the above design method, we were able to simulate the output result for the given g-code files with the expected correct output. Given below are the simulation results for the various test cases that were provided over the course of the semester. The first simulation result is for the first g-code file which was used to check for 2D collision detection. The second simulation result below depicts the output simulation waveforms for the 3D g-code file provided to check for collisions in the three-dimensional space. These were the simulation results that we were expected to achieve up until checkpoint 1 in our project deadlines. However, we were only able to simulate the output depicted in figure 2 until then, and tried to simulate the results in figure 3 later.

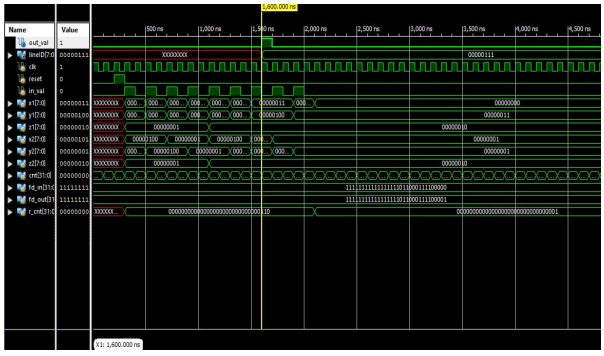


Fig. 2.: Simulation result for gcode.txt with collisions detected in 2D space.

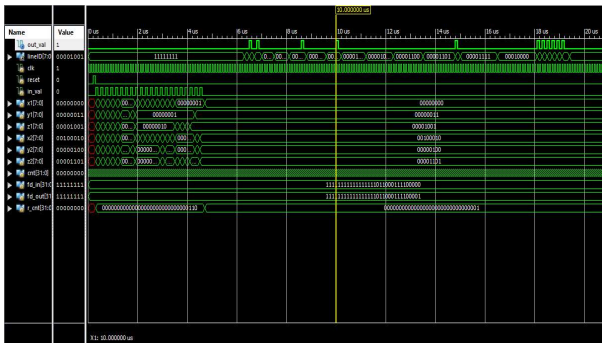


Fig. 3.: Simulation result for gcode_3d.txt with collisions detected in 3D space.

For checkpoint 2, we were provided with more test cases and g-code files to test our program against. We were asked to generate the correct simulation results for gcode_cp2test_2d.txt and gcode_test_3d.txt, and the correctness of our program was checked when compared to the simulation outputs of these two files. The boundary cases provided were to check for cases where the line segments are duplicated, and when there are cases where two line segments are intersecting at one of the line segment's end point, that is, two lines that form a "T" or "L". We were able to generate the correct output for the two given files, after checkpoint 2, which is shown in the simulation outputs below.

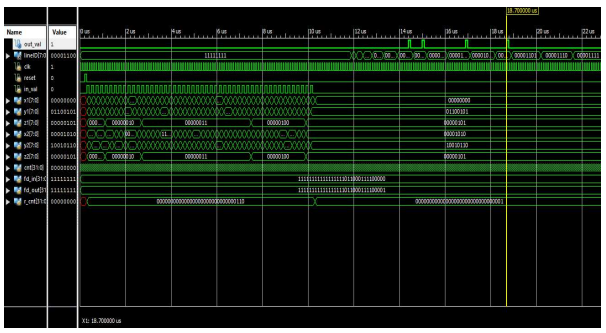


Fig. 4.: Simulation result for gcode_cp2test_2d.txt with collisions detected in 2D space.

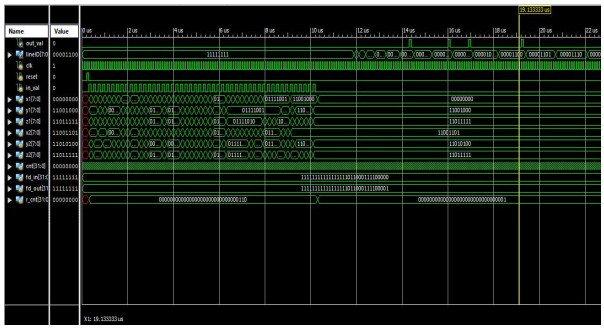


Fig. 5.: Simulation result for gcode_test_3d.txt with collisions detected in 3D space.

Based on the simulation results that we have generated for all the different test cases, we can infer that our program works correctly in all situations to detect wherever there is a collision occurring in both two-dimensional and three-dimensional space.

For all the known test cases and boundary cases, our implementation of the proposed design method works without any errors, or incorrect simulation results.

V. DISCUSSION

In this project, we implemented the line segment intersection method for detecting collisions in 2D and 3D space. By performing the following operation of detecting whether there is any line segment in our g-code file which intersects with any line segment coming before it, we were able to understand how the actual process of 3D printing takes place. We understood that the printer will take the g-code file as input, plot the lines that are present in the order in which they occur, and will continue in the same manner until the end of the file is reached. However, if a line segment is found in the input file such that there is a collision occurring, the printer will not be able to print flawlessly.

The main objective of this project was to ensure that there are no line segments in the input file, such that it will cause an error in the execution of the 3D printer. In the case that such a line segment does appear, it will be needed to discarded and marked as malicious. The rest of the line segments will then be stored in the memory module in the order of appearance.

The key concepts that were needed to be considered over the duration of the project were the concept of computational geometry, the memory constraints of the FPGA system (for synthesis) and the amount of time and computing power it would require to perform all of these operations. Firstly, we were required to consider the method that we would use to detect such malicious lines. The two options that were provided to us were the voxel marking method, and the line segment intersection method.

We chose to implement the line segment intersection method to detect for the occurrence of any collision. The basic idea behind this was that if there was an intersection between two given line segments in the two-dimensional space, there would be a collision at that point. We decided to first try to solve this problem by working on it in the 2D-space before extending it to 3D space. By doing so, we were able to understand the topic of computational geometry, and how it can be used to detect for line segment intersections, by making use of vectors, orientations of the lines, collinearity and cross product of vectors.

Secondly, we needed to also keep an eye on the memory constraints of our system. Our initial implementation made use of loops to generate the required output. This ensured that our program would provide us with the correct simulation result, in a short amount of time. However, since we were using loops, our program was not feasible for synthesis. This meant that although our implementation was running smoothly and efficiently on our systems, it could not be made to work on the FPGA board, which means that it required much more computing power and memory than that was available to us. For this reason, we had to find an alternative method to using loops in our program.

Lastly, we also needed to work on the amount of time required for the complete execution of our program. For smaller g-code files, our initial program was fairly efficient but as the size of the g-code files increased, that is the number of lines to be compared increased, it was clear that our initial implementation would not work as well. Also, to extend our solution to the three-dimensional space would add more complexity and reduce our efficiency. So, we needed to adopt a slightly different approach and eliminate the loops altogether, and make use of vector geometry.

In the case of a collision detection system implemented from scratch, the issue of expected development time might be as important as the desired feature set. For example, games are often on tight budgets and time frames, and the delay of any critical component could be costly. In evaluating the ease of implementation it is of interest to look at not just the overall algorithm complexity but how many and what type of special cases are involved, how many tweaking variables are involved (such as numerical tolerances), and other limitations that might affect the development time.

Several additional issues relate to the use of the collision detection system. For example, how general is the system? Can it handle objects of largely varying sizes? Can it also answer range queries? How much time is required in the build process to construct the collision-related data structures? For the latter question, while the time spent in preprocessing is irrelevant for runtime performance it is still important in the design and production phase. Model changes are frequent throughout development, and long preprocessing times both lessen productivity and hinder experimentation. Some of these problems can be alleviated by allowing for a faster, less optimized data structure construction during development and a slower but more optimal construction for non-debug builds.

REFERENCES

[1] Sungsuk Kim and Sun Ok Yang. Transforming algorithm of 3d model data into g-code for 3d printers in distributed systems. In *International Conference on Computer Science and its Applications*, pages 1074–1078. Springer, 2016.

[2] Amanatides, John. Andrew Woo. "A Fast Voxel Traversal Algorithm for Ray Tracing." *Proceedings of Eurographics 1987*, pp. 3–10, 1987.

[3] Barequet, Gill. Sariel Har-Peled. "Efficiently Approximating the Minimum-volume Bounding Box of a Point Set in Three Dimensions,"Proceedings of theTenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 82–91, 1999.

[4] Van den Bergen, Gino. *Collision Detection in Interactive 3D Environments.* Morgan Kaufmann Publishers, 2003.

[5] Christer Ericson, *Real-Time Collision Detection,* Morgan Kaufmann Publishers, 2005.

[6] https://users.cs.fiu.edu/~giri/teach/UoM/7713/f98/stara/stara.html

[7] https://en.wikipedia.org/wiki/3D_printing

[8] http://mashable.com/2013/03/28/3d-printing-explained/#f_Pr4SNjkuq6

[9] https://en.wikipedia.org/wiki/3D_printing_processes

[10] Project Description Report prepared by Professor Wenyao Xu