G-Code Re-compilation and Optimization for Faster 3D Printing

Xiaoming Li

University of Delaware, Newark DE 19716, USA
xli@udel.edu

Abstract: The 3D printing technology has seen increasingly wider application in industrial manufacturing and the general public domain. The normal working flow of 3D printing, i.e., from Computer Aided Design (CAD), to 3D model description, and last to 3D printers, essentially uses languages such as STL (Standard Tessellation Language or STereoLithography) and G-code to pass information between the work phases from designing to manufacturing. However, the languages are produced and used literally (like using XML for only data representation), and there has not been much discussion on how these de-facto programming languages can be compiled or optimized. In this paper, we present our preliminary work that tries to improve 3D printing's efficiency at the backend of the working flow. We re-compile the G-code into a higher-level IR, design a number of physics and graphics driven optimizations, and re-generate G-code from optimized IR representation. We test our G-code compiler on several popular 3D models and show upto 10.4% speedup or save more than 16 hours on printing complex models.

1 Introduction

The 3D printing is an emerging technology for product development and manufacturing. It is young, but it has passed the stage of being only a hobby activity, and has seen rapidly growing applications in industry. Some of the high-profile products that are 3D printed include Space-X's SuperDraco engine [2], Airbus aircraft parts [3] and many.

A standing-out issue for 3D printing is its speed [5]. Compared to the traditional manufacturing technologies such as casting or forging, which handle volume of material in batches, 3D printing builds up shapes with elementary forms of material such as powders or filaments. Its speed depends on both the volume of product and also the shape's complexity. It is quite common to take days to print even a seemingly simple 3D shape. For example, NASA is developing 3D printing technology for the first manned lunar base. In its latest demo, a simple chair took about 2 weeks to print [7].

This paper presents an initial and exploratory effort to improve 3D printing's speed using compiler-derived techniques. To understand the speed issue of 3D printing, we need to examine its complete workflow. 3D printing usually starts with a Computer Aided Design (CAD) software such as AutoCAD. The outcome of CAD is usually the 3D geometrical representation of a product. Thereafter, the 3D geometrical representation is processed by a type of software called "slicers" to be transformed into solid models and further been translated into commands that can be understood by 3D printers. 3D printers will load the commands, which are movements of servos and settings for printing such as temperatures or speeds. The embedded controller on 3D printers will execute the commands and directly control servos to move the printing nozzles and extruders.

Clearly, we can attempt to improve 3D printing's performance at any phases of the workflow. In this paper, we particularly look into the interface to the embedded

3D printer controller. The commands are usually represented in an industry standard format called G-Code. Our key observation is that while being sufficient to control the movement of servos, G-code is primitive and loses high-level information about the product being printed. The current technique that translates the 3D geometrical model to the elementary movements in G-code is direct and does have printing speed in mind. There has been a lack of proper intermediate representation for any optimization work to became possible.

The main contributions of this paper are a higher-level printing IR for G-Code, new optimization techniques that respect the physical contraints of 3D printing but reduce the total printing time, and a G-code generator that re-generates improved G-code from this IR. We evaluate our G-code compiler on a number of popular 3D printing models and achieve up to 10.4% speed improvement.

2 Background and Overview

The 3D printing technology is a new manufacturing technology to build 3D shapes. Generally speaking, traditional manufacturing methods such as forging ,casting or injection molding manipulate volume of material into desired shapes. In contrast, 3D printing builds up a shape by gradually adding minuscule amount of material into place piece by piece and layer by layer. That is why 3D printing is also referred as "additive manufacturing" in many contexts.

The 3D printing workflow typical involves three phases: CAD, slicer, and 3D printer. CAD software such as AutoCAD, Fusion 360 or OpenSCAD can be used to design 3D models. CAD interfaces with slicer programs with the mesh description of 3D model. One of the industry's defacto standard format for such mesh description is STL (Standard Tessellation Language or STereoLithography).

STL files cannot be directly printed because they only describe the surface geometry of a three-dimensional object without any model attributes. Slicer programs such as *slic3r* or *Simplify3D* translate STL files into printable description of 3D model. The translation involves two main tasks: figuring out the movements of extruder to implement the 3D object, and conforming to a 3D printer's specific physical capability. Slicers also need to make sure the movements are legitimate for a specific printer, and at the same time try to maintain the print quality of the final product. Therefore, slicers will specify the physical attributes of the movements such as the extruder temperature, the building bed temperature, and the speeds of servo involved in movements.

The output of slicers is the description of all the movements, together with the specification of the printer's setup and the physical attributes of movement. All of these are written in the G-code file to be send to the printer.

2.1 System Overview

The proposed G-code compiler has three main components: (1) the geometric IR, which contains the same semantic information as the G-code, but stores it in a graph-like data-structure to facilitate the analysis and transformation; (2) front-end and backend, which convert between the G-code and the geometric IR; and (3) compiler transformations that currently include only a few essential passes including the preprocessing passes and the speed optimization passes.

Figure 1 shows the overall working flow of the compiler. The geometric IR and the front/back ends are described in Section 3 and the compiler passes in Section 4.

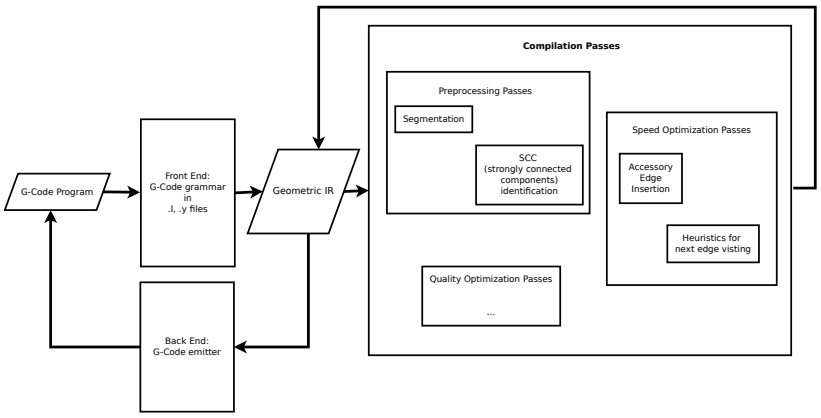


Fig. 1: Working Flow of the G-Code Compiler.

3 Intermediate Representation for G-Code

3.1 Design Consideration

What is G-Code? G-code is a numerical control language for programming Computer Numerical Control (CNC) devices. It was born out of the MIT Servomechanisms Lab in 1950s, and has since been extended by numerous standard institutions and companies. Originally the language is designed to describe the control and movement of cutting tools, i.e., essentially telling the servos where to move and when. Since its beginning, the 3D printing technology also adopt this concept. Today G-code also becomes the de-facto programming language for 3D printers.

Just like the CNC domain, the 3D printing community has also developed numerous variants of G-code such as the Marlin, RepRap or MakerBot dialects. While the variants are largely semantically compatible, they differ in the representation of numbers and the settings of printer. Here we briefly describe the stem of the G-code that is more-or-less common among all variants.

From the programming language perspective, G-code is extremely simple. Most G-code variants don't contain any control structures such as conditional branch or loop. The language has two basic command sets—G commands that start with the letter "G", and M commands that start with the letter "M". G commands basically describe movements. For example G1 (also represented as G01) specifies a linear interpolated movement. G3 represents a counterclockwise, circular interpolated movement. M commands basically specifies settings including both movement settings and machine settings. For example, M205 sets the jerk rate (max acceleration) on the XY axis. Most M commands are modal commands, which means that the effect of the commands stays in effect until being replaced.

Geometrical Movements The requirement for the IR to store the movements is easy to implement. The majority of a G-code file is the commands describing the servos' movements. The number of dimension in movement, or the degree of freedom in servo motion, is essentially the number of servos in a printer. The basic printers have 4 motors, and more advanced varieties have more. Usually the 3D geometrical space is consisting of the X, Y and Z dimensions and is controlled by three servos. There are two types of mapping from the three servos to the X/Y/Z dimensions. In the Cartesian style of 3D printers, each servo controls the motion along one dimension. The mapping is direct and linear. For the Delta 3D printers [4],

all three servos participate in the motion control along all three dimensions. The motor movements are translated into 3D space movements through trigonometric remapping.

In addition to the three servos that control the spatial movements, 3D printers also have extruders that additively accumulate material. Normally one extruder is directly controlled by a servo. So the number of material handling servos equals, in most cases, the number of extruder equipped in a printer. An extruder servo mostly moves in one direction, that is, extruding/adding material. On the other hand, printing an object involves move the extruder motor backward, i.e., for retracting, from time to timer. For example, to avoid the leakage of material when the extruder moves from one section to another disconnected one, the corresponding extruder servo will retract and when arriving at the destination location, extruding again.

Movement Setting and System Setting Up to this point, the IR for G-code is an abstract geometric space. Hypothetically speaking, transformations using it won't guarantee the correct printing of the transformed G-code. The reason is that a movement needs to be accomplished with proper settings such as temperatures and speeds for it to be printed correctly and with good quality. The setting is also highly contextualized, i.e., the properness depending on what happens before the movement. Therefore, a practical IR must incorporate information the printer setting, and equally importantly maintain the dependency of the settings.

The main settings relevant to movement are temperature and speed. Movements are printed with different temperatures. The reasons for doing that include to guarantee good bed adhesion, reduce material warping or improve surface quality of object. Therefore, when our compiler parses the G-code, it needs to deduce the temperatures for all the movements, and attaches the information to the edges in the Cartesian space.

A 3D printer also constantly changes printing speed in the process of printing an object. For example, when printing a small section, the speed might be reduced to give material sufficient cooling time. As another example, when the movement is across a large unsupported section, the speed might be increased to avoid sagging of the material (a.k.a "bridging" mode). Clearly the speed setting of a movement is context-sensitive, i.e., that the appropriate speed setting depends on what happens before it.

In addition to the settings related to movement, G-code also contains commands pertaining to features that are specific to slicer or printer. Those commands usually appear at the prolog or the epilog parts of G-code. The commands are usually not tied to specific movements. The compiler's frontend will recognize the start and the end of those sections, and re-emit them when transforming back into G-code.

Encoding Printer's Physical Constraints By this point, the IR design has become capable of representing the geometrical space that a printer's motor can reach, arbitrary movements in the space, and the settings of the printer's auxiliary equipment such as heat bed or fans. One thing that needs attention is the resolution of number in the IR. The space and movements are *not* continuously reachable by the printer. The servos are mostly step-driven. That means, they can only rotate as multiples of the minimum amount of rotation, and can't go below the rotation resolution. In other words, the numbers should be discretized according to the printer's capability.

For example, a key specification parameter of 3D printer is the minimum layer height. The parameter is usually linked to the rotation resolution of the Z-axis servo.

If the minimum height is $0.1mm$, any representation of the Z-axis position in the IR should be a multiple of $0.1mm$. Imaging that if a transformation introduces a new vertex in the space with $Z = 3.55mm$, the newly created vertex, however, is meaningless because it simply won't be able to be reached precisely by the printer. Practically the printer's controller might still accept such a value in the G-code, but where it actually goes is unpredictable, and may likely create printing quality problems such as the separation of layer.

Therefore, we make the resolution of number in the IR explicitly visible in the IR. The resolution constraints are not only maintained when G-code is parsed or re-generated, but also are mandated when transformations are applied on the IR.

3.2 Definition of the Geometric IR

We have so far discussed the main points for consideration when designing the IR for G-code. Taking all these into account, we can put together a formal definition of the G-code IR. The IR has two parts, a graph representation that describes the geometric information in the G-code, and the decorating properties that store the settings for the geometric movements and the printer.

The geometric basis of the IR is a N-dimensional *graph*, N being the number of servos in the 3D printer that the IR is representing. The graph is *undirectional*, because material printing can be done on either one of the two directions of the movement. So this geometric information can be encoded as $G = (V, E)$, V is the vertex set and E is the edge set. For $\boldsymbol{v} \in V$, $\boldsymbol{v}$ is a n-dimensional vector, where each element $\boldsymbol{v}_i$ represents the rotational position of the i_th servo. For $E = e_i | e_i = \boldsymbol{v_{i_0}} - \boldsymbol{v_{i_1}}$, each edge in the set represents the linear interpolated movements of servos between $\boldsymbol{v_{i_0}}$ and $\boldsymbol{v_{i_1}}$. The edges are undirectional.

The graph is decorated with three categories of property: vertex property, edge property and environmental property. The vertex property in the current IR contains the position vector $\boldsymbol{v}^N$ of the vertex, and several flags that can be set by analysis or transformation passes to facilitate future processing of the graph. The flags include whether the vertex is introduced new in the segmentation pass, and the strongly connected component index that the vertex belongs to.

The edge property contains the settings for correctly printing the edge. The settings include temperature, line speed of movement, and starting/ending actions. The temperature and the speed settings are self explanatory. The starting/ending actions of an edges are those M instructions in the G-code that do not directly affect movements but still need to be done before or after the printing of the edge. Examples of such actions include $M207$—setting jerk rate, or $M212$—setting bed offset for the auto-leveling feature, etc.

The environmental property describes the printer system constraints such as the minimum resolution for servos or the sensors' precision. Even though part of such information can be reasoned from G-code, our current implementation manually provide the environmental property as an external configuration for the compiler.

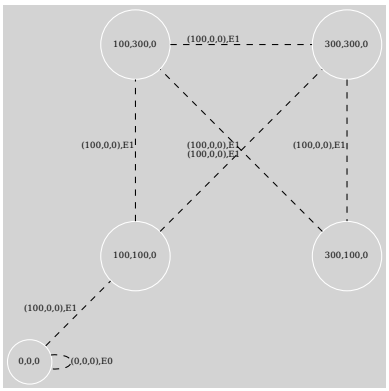
3.3 Example of IR

Here we illustrate the proposed geometric IR with an example segment of G-code. The example is simplified for this illustration because in real G-code, the dimensionality is at least 4—one servo each for the X, Y, and Z dimensions and one servo for extruding. It is hard to visualize a 4 dimensional graph. So in this example, we only use the X and Y axis and one extruder dimension in the G-code program.

Figure 2a shows the example G-code segment, and Figure 2b shows the visualization of the corresponding IR. As we can see, each G-instruction is translated into

```
G1 X100 Y100 Z0 E1 F100
G1 X100 Y300 E2
G1 X300 Y300 E3
G1 X300 Y100 E4
G1 X100 Y300 E5
G1 X100 Y100 E5
G1 X300 Y300 E6
```

(a) G-code



(b) IR visualization

Fig. 2: G-code example and its corresponding IR visualized.

one edge, and the *M*-instructions preceding an edge will be attached to the starting action property of the edge.

4 G-Code Optimization

The G-code IR provides a holistic representation of information contained in the G-code output of slicer. The distinctive characteristic of the IR, compared with the raw G-code format, is its high-level semantic. The IR is naturally geometrical and the raw G-code is sequential. For example, movements in a G-code file might be totally independent to each other. However, it is hard to tell that in the original form, as the G-code mandates an unnecessary order between the movements. On the other hand, the IR representation of the same movements can be easily analyzed for their dependency or even spatial locality, and further be reordered for faster printing.

The higher-level semantic of the G-code IR opens the door for 3D printing transformations. That is, transforming a G-code IR representation of an object into other equivalent IR representation, and when feeding the transformed IR to a 3D printer, still produces the same object. The concepts involved in this G-code compilation and optimization are similar to those for the typical computer programming languages. However, as we can imagine, computers compute, but 3D printers perform a very different kind of job in a very different way. We need to redefine what kind of transformation is legal, what are the optimization goals, and how to model performance (i.e., speed, quality, etc.) for the specific problem of compiling for 3D printer.

4.1 Compilation Constraints

A transformation of the G-code IR changes both the geometric description and also the order of movements or extrusion. First we need to find an operable definition of what is a valid transformation. Eventually, any valid G-code transformations should be able to print the same object that the original G-code program intends to make. The question is how this correctness requirement can be translated into a series of legality tests, like the dependency test for our computer compilers?

The correctness requirement for 3D printing basically means two things: (1) a valid transformation must extrude material exactly as the original extrusion movements do in the IR. The transformation cannot extrude more, and cannot extrude

less. (2) The order of extrusion must be feasible for the 3D printer. We can use a simple example to demonstrate what is the "feasibility" here. When moving from the position $(x, y, z) = (100, 100, 2)$ to $(100, 200, 2)$, for example, there should not be any place along the moving path where the height of the printed part is higher than 2. Otherwise, the movement will damage the printed portion.

Beyond that a valid transformation must print out the same shape, like a compiler transformation should produce the same results, 3D printing has requirment on the quality of printout. Still using compiler transformations as example, in a computer program, transforming "1+1" into "1*2" will be legal but may carry different speed characteristics. The term "performance" in 3D printing not only means printing speed, but also printing quality. When the movements in a printing job are reordered, in many cases, the outcome can have drastically changed quality. It is because materials that 3D printer handle, such as PLA, ABS or Nylon, exhibit different physical properties such as layer adhesion when they are printed with different speeds, or with printing direction from the layer below, etc.. Therefore, when we transform G-code, we need to respect this additional quality constraint.

4.2 Optimization For Printing Speed

Just like that our computer compiler transformations can be tuned for different goals such as speed or code size, G-code can also be transformed to improve on different 3D printing metrics such as printing speed or product quality. In this paper, we describe our exploratory study of printing speed optimization using the G-code IR.

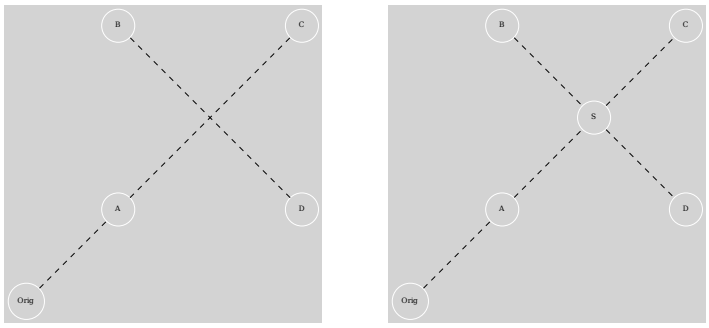
Pre-processing with IR Given an IR representation of a 3D printing task, the basic job is to traverse all the edges exactly once. As we will discuss later, this job sounds like an Euler Tour problem. But before we start looking into how to walk through the graph, we want to point out that the raw IR representation hides potential optimization opportunities. And these opportunities can be made available for later transformation by adding a preprocessing pass after the compiler front-end.

In this study we implement two preprocessing passes: graph segmentation and connected component identification.

The purpose of graph segmentation is to facilitate the route searching in the Euler Touring based optimization. Here is an example of how graph segmentation can help. As Figure 3 shows, in the original G-code representation, two movements $A - C$ and $B - D$ intersect in space. However, because they are originally represented as two separate edges, the Euler Touring algorithm might not be able to take advantage of the intersection to find more efficient tours. If we segment the graph, i.e., introduce a dummy vertex for the intersection point, the touring algorithm would be able to move only parts of the original movements and find more efficient touring path. Thanks to this added flexibility of touring, after graph segmentation, the Euler Touring algorithm will find the shortest tour that uses the additional connectivity of the dummy vertex "S".

Graph segmentation itself is a conceptually very simple processing. A naive algorithm will try to intersect an edge with all other edges. If any two edges intersect, introduce the intersection point as a new vertex and break up the original two edges into four based on the intersection point. In this case, the complexity of the naive segmentation will be $O(E^2)$, where E being the number of edges. Because the newly created edges might also intersect with other edges, the iteration will continue until no further changes are made.

The naive graph segmentation algorithm won't work in practice. A typical 3D object will involve millions of edges. The complexity of $O(E^2)$ simply make the naive



(a) Original Graph (b) Segmented Graph

Fig. 3: IR Pre-processing: Graph Segmentation

algorithm not viable. We use two pruning techniques to accelerate the algorithm. The first is to further preprocess the graph IR to identify connected components. So that we only need to test intercepting with a component. We use a DFS-based approach to find all connected components. Since connected components can still be huge, we further re-organize a component into layers and only try interception test with in the layer that the edge belongs to and the layer below and the layer above, where are the only place that any potential intercepting edges can reside.

4.3 Printing Speed Optimization

As previously discussed, the proposed G-code IR supports a variety of optimization goals such as speed, printing quality or physical strength. This paper describes our preliminary result of optimizing speed. Under this set up, the goal function is simply to cover all edges in the original graph exactly once, and minimize the overall *time*.

The goal sounds very much like an Euler Tour problem, i.e., finding a path in a finite graph that visits every edge exactly once. The main challenges to adapt the Euler Tour problem in the solving of the G-code printing speed problem lie in the subtle but fundamental problem setup differences. The differences are derived from how 3D printers work and perform.

The starting point of our preliminary optimization is based on the Hierholzer's algorithm [6,8] with solutions to address the specifics of the G-code optimization setup. Our main effort is spent on addressing the differences between the Euler Tour problem and our optimization problem. Next we describe the differences and our solution thereof.

– Goal: The Hierholzer algorithm finds *an* Euler tour in a graph, and that's it. There is not any optimization criteria built-in. The only guarantee is that every edge is visited exactly once. But no effort is made to find the tour that minimize the total distance or other graph metrics. Note that the total distance in an Euler tour is not fixed for a graph, as accessory edges need to be introduced in order to guarantee the existence of an Euler tour.
  Our solution is to include heuristics at several places in the algorithm to optimize the total tour time. The places include the construction of accessory edges, the choice of next edge and the choice of next vertex to visit. The final heuristics also consider the next couple of challenges, and will be detailed in Section 4.4.

– Complexity: The computation complexity of the Hierholzer's Algorithm is $O(V+E)$, V being the number of vertices and E being the number of edges. Real-world G-code, when transformed into the proposed IR, can contain millions of vertices and/or edges. It is impractical to blindly apply the Hierholzer's Algorithm. Our solution is motivated by a type of compiler pass, i.e., the Strongly Connected Components (SCC) passes. That is, we apply the optimization algorithm on every connected component that has been found. Also when the work on one component is finish, we use the same heuristic that finds the next vertex to identify the next component to process.
– Performance of edge visit: In the setup of the Euler Tour problem, the weights of edges are constant. In our G-code optimization problem, the weights are the distance of movement, and they are indeed constant, too. However, the time to travel through the edge is not. This is because for a 3D printing to move, the setting of the movement must be ready, which introduce overhead. Also the change of moving direction lead to acceleration, and the involved servos need to do extra work to handle the G-force. Overall, the time to travel through an edge is contextually dependent on the previous edge. Our solution is to build a physical performance model, quite rough at this stage, for the edge traversal, and incorporate the model into the optimization heuristics.

4.4 Optimization Heuristics

Accessory edges: In order to find an eulerian tour, accessory edges need to be introduced in a graph to connect pairs of vertices with odd degrees. We need to minimize the total distance of the introduced accessory edges to optimize the total time for traveling the eulerian tour. That is, if a graph has n odd-degree vertices, we need to find the division scheme that minimize $\sum_{\forall v} dist(v_i, v_j)$, This is another well-known algorithmic problem called the *Pairwise Optimization* problem. We use a simple heuristic to solve the problem. We build a matrix of all the pairwise distances any two of odd-degree vertices, and use Dynamic programming to iteratively remove the next shortest pair, until all odd-degree vertices are covered. We want to point out that our heuristic is *not* globally optimal.

Next edge to visit: In the Hierholzer's Algorithm, if a just-visited vertex has multiple un-visited outbound edges, a random choice is made. In the case of G-code IR, the choice of the next edge carries significance with regard to the edge traversal time. There are two reasons. First, if the next edge has different setting, e.g., speed or temperature, the printer need to change setting first before it can drive the servos to make the movement. That introduces overhead. Second, change of moving direction introduces G-force in servos. This is call "jerk rate" in the 3D printing terminology. Without going into too much physic details, the short conclusion is that the lower the G-force, the faster the printing. Using the example in Figure 3b, if the current vertex is S, and the previous edge is $B - S$, the best next edge is $S - D$ but not $S - A$ or $S - C$, as $S - D$ is mostly aligned with the previous edge and will incur the least G-force.

We develop a simple heuristic here. We first check the printing setting of the edge candidates, and if possible, only choosing from the ones that have the same setting as the current edge, or if not possible, involving the least setting changes. If there are still multiple candidates, which are the majority of the cases, choose the one the involves the minimum G-force to travel.

5 Experiment and Evaluation

The G-code compiler and the printing speed optimization are evaluated with 3D models. There have been no public available compiler/optimization work on G-code, and as the result there is no "standard" benchmark for the kind of evaluation we want to do. Fortunately, due to the increasing populaty of 3D printing technology, there are multiple websites for people to share 3D printing models—sort of like github for 3D models. We use `http://www.thingiverse.com` (Thingiverse), one of the most widely used 3D model sharing site, and use several of the most popular models on that site as the benchmarks. The models are "Baby Groot", "Benchy", "Printer Test", and "Mid Castle". Table 1 shows the benchmark models, download links and the total number of downloads as reported by Thingiverse.

Table 1: 3D Model Benchmarks

Benchmarks	*URL*	*# of Downloads*
Baby Groot	`https://www.thingiverse.com/thing:2014307`	32402
Benchy	`https://www.thingiverse.com/thing:763622`	42329
3D Printer Test	`https://www.thingiverse.com/thing:2656594`	32235
Medieval Castle	`https://www.thingiverse.com/thing:862724`	18701

All the models are downloaded as STL files. We use Simplify3D [1], a widely used commercial Slicer to generate G-code for the STL. The 3D printer we use is JGAurora A8, and its controller firmware is Marlin, a Linux-based software that is widely used as the operating system in 3D printers.

The G-code output from Simplify3D is the input to our compiler and optimizer, and our output is also G-code. We measure the printing time of the before/after versions of the G-code. Actually Simplify3D also reports estimated printing time based on its own G-code output, and in almost all cases, its estimation is spot on. In this paper, we report the actual printing time.

Table 2 shows the before and the after printing time of the benchmark modes. We also collect statistics of the model before vs. after, including the number of vertices, number of edges, total distance traveled. As the result shows, our optimization achieves upto 10.4% speed or about 973 minutes for the model "Medieval Castle" that has the highest number of edges (9.08 million). On simpler models, our speed-ups are around 5%.

Table 2: Before/After Comparison and Speedups.

Benchmarks	*Edges*	*Vertices*	Total Distance (mm)	Time (minutes)	Speedup
Baby Groot	3.39M/3.42M	3.58M/3.65M	183.536K / 174.227K	632.945/605.71	4.7%
Benchy	2.05M/2.43M	2.43M/2.67M	5.34M/5.22M	1866.1/1766.75	5.3%
3D Printer Test	190.9K/194.2K	211.37K/213.3K	872.1K / 829.3K	380.301 362.723	4.6%
Medieval Castle	9.08M/9.76M	16.81M/17.33M	29.1M/27.51M	9358.84/8385.93	10.4%

6 Conclusion

In this paper we present the preliminary design of a G-code compiler. Particularly we introduce an appropriate IR that captures all information in G-code and in addition makes it easily to retract higher-level graphic and physical information. Furthermore, we discuss the legal test for G-code transformation on the IR and several heuristics for improving the printing performance. The evaluation using several popular 3D models shows up to 10% speedup on complex and long printing jobs.

References

1. Simplify3d. https://www.simplify3d.com/
2. Spacex uses dmls to 3d print inconel superdraco engine chamber. https://additivemanufacturingtoday.com/spacex-uses-dmls-to-3d-print-inconel-superdraco-engine-chamber
3. Bridging the gap with 3d printing. (2018), https://www.airbus.com/newsroom/news/en/2018/04/bridging-the-gap-with-3d-printing.html
4. Bell, C.: 3D Printing with Delta Printers. Apress, USA, 1st edn. (2015)
5. Gibson, I., Rosen, D.W., Stucker, B.: Additive Manufacturing Technologies: Rapid Prototyping to Direct Digital Manufacturing. Springer Publishing Company, Incorporated, 1st edn. (2009)
6. Hierholzer, C., Wiener, C.: Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren (March 1873)
7. Staedter, T.: Ai spacefactory wins nasa's 3d-printed extraterrestrial habitats challenge. In: IEEE Spectrum. IEEE (May 2019)
8. Torrubia, G.S., Blanc, C.T., Navascués-Galante, L.: Eulerpathsolver : A new application for fleury ' s algorithm simulation (2009)