Programmatically generating 3D models in C

Zak Edwards* and Cameron Mason†

May 7, 2015

Date Performed: April, 2015
Module: CS132 *Computer Organisation and Architecture*
Instructor: Steven Wright

Abstract

3-dimensional physical models constructed by issuing a series of G-codes to a print head are shown to be constructable, to varying degrees of efficacy, in a wholly programmatic manner using the C language. This programmatic replication of 3-dimensional model production lends itself towards a concise, interpretable and logical manner of 3D printing. The physical structure of greatest complexity ultimately intended to be printed is that of the *Lego brick*; this is shown to be a feasible endeavour, by virtue of building upon elementary geometric shapes generated by implemented C methods and calling these methods in a certain sequential manner. Considering geometric relations of proportionality, these structures, and thus the final brick structure, are shown to be generable and printable contingent upon user-specificied dimensional parameters.

1 Introduction

The 3-dimensional geometric constructions mechanically generated and printed via the utilisation of *G-codes* (*cf.* 1.1) are reconstructable programmatically in C. Implemented methods in C provide a basis for a sequence of commands, issued to the 3D printer and interpreted as a series of G-codes, that generate and print elementary geometric forms. Constituted by a composite of these elementary forms, attempts are made to determine that a relatively complex geometric structure can effectively and efficiently be produced. Pertinent to these aims, a number of objectives are realised:

*Z.Edwards@warwick.ac.uk, Department of Computer Science, University of Warwick
†C.G.Mason@warwick.ac.uk, Department of Computer Science, University of Warwick

I

To *construct programmatic methods* in `C` for a number of geometric shapes generated by G-codes implemented in `Marlin`[1].

II

To *optimise* these programmatic generations in order to produce the most *optimally constructed* and *efficiently printed* physical results.

III

To *combine* a series of successfully generated geometric shapes to ultimately compose a 'Lego brick' model.

1.1 Nomenclature

G-codes individual commands issued to the 3D printer by `RepRap`[2] firmwares in order to achieve a mechanical action. A series of G-codes performs a sequence of actions; files containing such series have the extension `.g`.

Pronterface the graphical interface used to provide a relationship between inputted G-codes and the 3D printer. The software belongs to the `Printrun` suite of G-code sending applications.[3]

2 Requirements Analysis

In order to construct effective and efficient control software in `C` that will allow the above objectives to be accomplished, a number of requirements ought be specified.

A necessary minimal requirement is that each generated *elementary* shape should be *connected* and *robust* – that is to say, there should exist no gaps or weaknesses in the perimeters of generated models. Further, *filled* shapes should be *complete, i.e.,* there should also exist no gaps or areas of overt vulnerability to the extent that is possibly preventable, and *uniform* in dimension. The latter condition of uniformity is necessary to ensure that printed shapes are not affected by contortion caused by the protrusion of extruded material; methods implemented to fill areas must thus avoid any overlapping and conform to the restrictions imposed by the inherent physical properties of the 3D printer.

A further fundamental requirement concerns the ability for generated shapes to be compositable, *i.e.,* shapes should be able to be built upon without immediate difficulty and composed to form shapes of greater complexity. A less

[1] The 3D printer firmware used; see `https://github.com/MarlinFirmware/Marlin` for more information.

[2] *cf.* Jones et al. (2011).

[3] For more information concerning `Pronterface` and related softwares, see `https://github.com/kliment/Printrun`.

basal but nonetheless important requirement concerns the *efficient* generation of shapes – such a requirement ensures that *optimally minimal* material is used, to the extent that robustness and physical stability is not significantly compromised. This requirement proves to be of great relevance and significance to the industrial aspects and applications of 3D printing technology.

The ultimate objective of generating a composite brick demands that various primitive geometric shapes are foremostly generable. A cursory observation of the composite brick structure reveals that generability of the *cuboid* and *cylinder* shapes is a certain requirement; these shapes are further deconstructable, however, namely into the *cube* and *circle* shapes, and further into the *square* and *line* forms. Thus, an effective, practical programmatic implementation of each of these geometric forms must be realised.

3 Design

As shown in *Fig.* 1, one can obtain the desired composite Lego-type brick by building upon a series of increasingly complex geometric forms, the *line* being the most primitive of these forms.

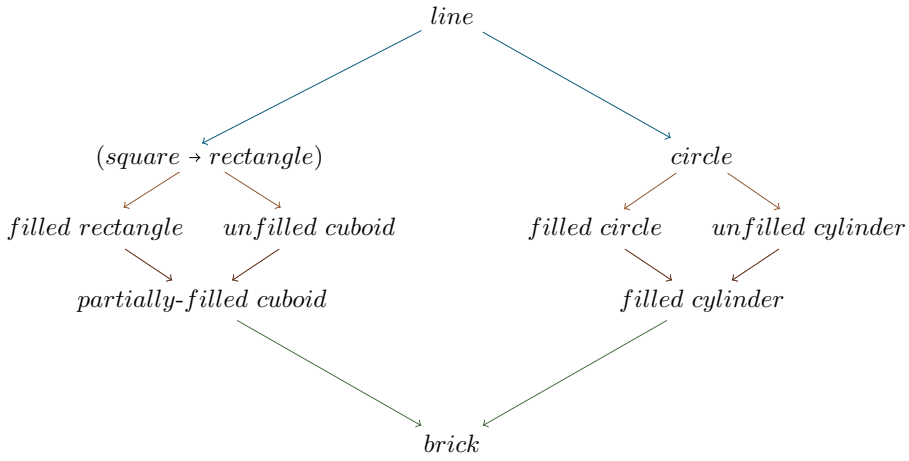

Figure 1: Design scheme for composite geometric shapes.

Immediately generable by calling upon an implemented *line* method are the *square* and *circle* shapes. The (unfilled) square method consists obviously of four iterations of the line function; the circle form can feasibly be constructed by repeatedly drawing appropriately small lines about an origin with a given radius, effectively generating a polygonic perimeter. The most immediately viable method of filling in these shapes is by decrementing the area of the unfilled shape and generating the smaller shape on the same z-plane; thus, the outer

shape effectively contains smaller iterations of itself. The necessary quantity of these iterations, relative to the geometric form in question, can be calculated algebraically.

Rectangular forms can be generated with simple modifications to the square method; if the archetypal rectangular basis of the desired composite brick is taken to be of a $1 : 2$ relation in the $x : y$ dimensions, one would need only to programmatically double the inputted value for the x dimension to obtain the measurements for a printable rectangular shape. The composite brick is constituted largely in part by a *partially-filled cuboid;* that is, a predominantly hollow cuboid, generated by successive iterations of an unfilled rectangle, with a single layer composed of a filled rectangle. Such a cuboid is therefore *partially-filled* insofar as its top layer is constituted by a *filled* rectangle. The remainder of the brick's composition is constituted wholly by filled cylinders, geometric forms which, in precisely the manner aforementioned, can be generated by successive iterations of filled circles along the z-axis. A diagrammatic representation of a composite brick system, composed of subsystems of lesser complexity, is shown in *Fig.* 2 (here, the brick has a *cube* basis as opposed to the canonical *cuboid* basis).

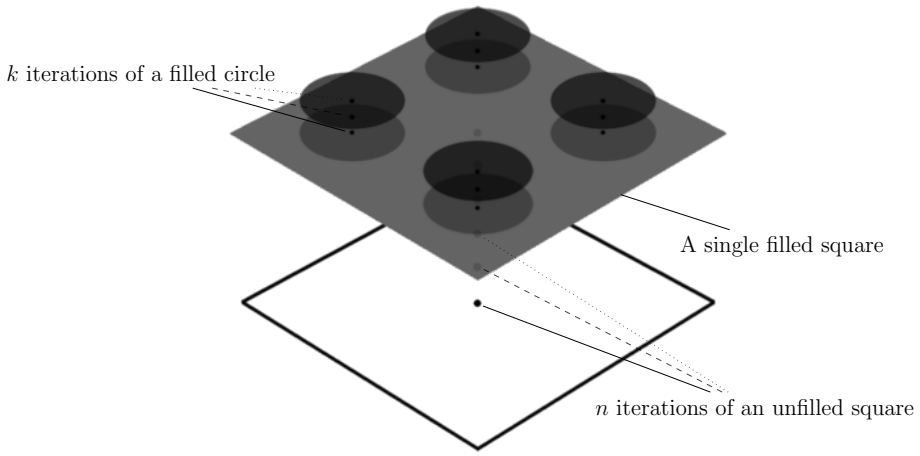

Figure 2: A diagrammatic deconstruction of the *cube* brick's components.

Therefore, bearing in mind these considerations, the generation of a Lego-type brick model is wholly feasible, precisely by considering foremostly the geometric forms of greater fundamentality.

4 Implementation

4.1 Line

```
1  void print_line(position_t* from, const position_t* to) {
2
3          double ext = hypot(from->x - to->x, from->y - to->y)/8;
4
5          printf("G1 X%f Y%f E%f\n", to->x, to->y, ext);
6          *from = *to;
7
8          }
```

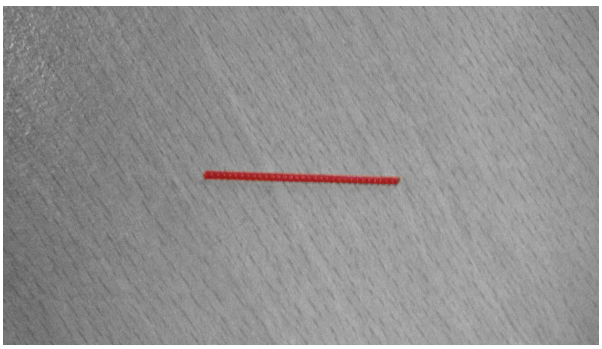

Figure 3: First successful attempt at generating a line.

4.2 Square

4.2.1 Unfilled square

As shown in the below implemented method, the unfilled square is generated in the immediately obvious manner of drawing four lines by calling upon the `print_line` method. That is, for each line, the positional parameter `pos` is assigned the position of the end point of the previous line, and its x or y value is altered according to a single addition or subtraction of the parameter distance `size`.

```
1  void print_square(position_t* pos, double size) {
2
3          position_t pos2;
4          init_position(&pos2);
5
6          pos2 = *pos;
7          pos2.x = pos->x + size;
8
9          print_line(pos, &pos2);
10
```

```
11          *pos = pos2;
12          pos2.y = pos->y + size;
13
14          print_line(pos, &pos2);
15
16          *pos = pos2;
17          pos2.x = pos->x - size;
18
19          print_line(pos, &pos2);
20
21          *pos = pos2;
22          pos2.y = pos->y - (size - 0.5);
23
24          print_line(pos, &pos2);
25
26          }
```

A difficulty here lies in ensuring the print head does not extrude plastic over the origin point - to countenance this physical property, the final drawn line in the square is of length 0.5mm less than all other drawn lines, as this distance is precisely the uniform width of the extruded plastic.

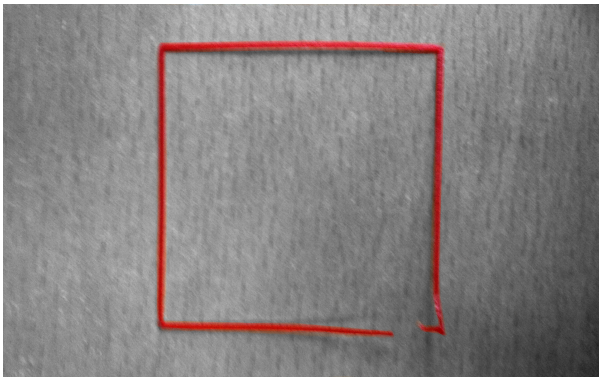

Figure 4: First successful attempt at generating a square perimeter.

4.2.2 Filled square

Fig. 5 concerns a geometric analysis towards determining the number of iterations required to fill the square; that is, the number of individual, unfilled squares that, when appropriately decremented, constitute a filled square when composed.

Ascertaining the relevant diagonal lengths with regard to both the square of dimensions $size \cdot size$ and the uniform extrusion width of 0.5mm, one can determine that the required number of iterations is

$$\sqrt{\tfrac{(size)^2}{2}} \Big/ \sqrt{2 \cdot (0.5)^2} \quad \text{or} \quad size^2.$$

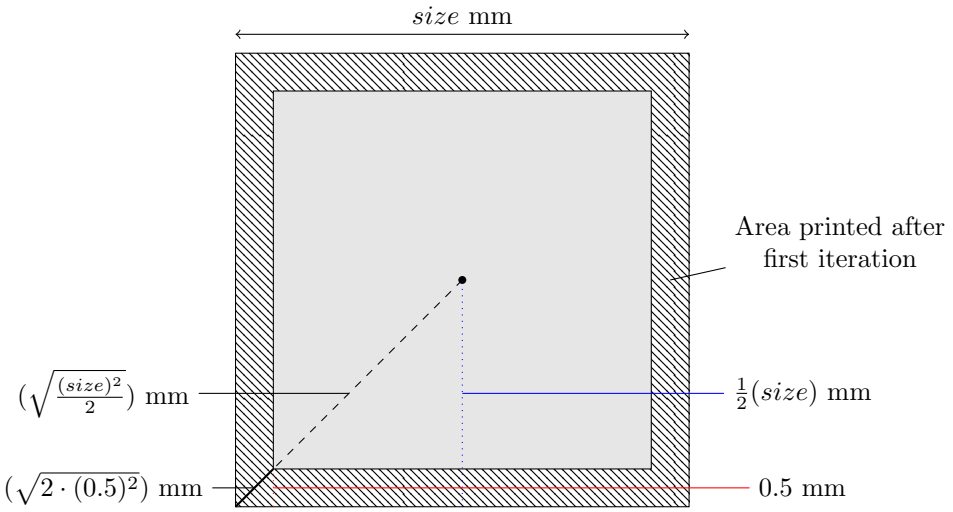

Figure 5: Determining the number of iterations required to generate a filled square.

Thus, implementing this limit simply entails specifying `size*size` as the strict limit of an iterative `for`-loop, within which the size is decremented subsequent to printing the preceeding square.

```
1  void print_filled_square(position_t* pos, double size) {
2
3          printf(" ; Printing Filled Square at (%f,%f,%f) of size %f
               .\n", pos->x, pos->y, pos->z, size);
4
5          position_t pos2;
6          init_position(&pos2);
7
8          pos2 = *pos;
9
10         for (int i = 0; i < (size*size); i++) {
11
12              print_square(pos, size);
13              size--;
14
15             *pos    = pos2;
16              pos2.x = pos->x + 0.5;
17              pos2.y = pos->y + 0.5;
18
19              }
20
21         }
```

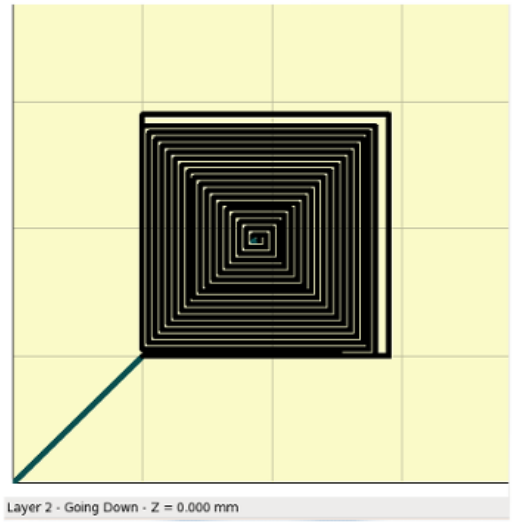

Figure 6: `Pronterface`-generated preview for a filled square.

4.3 Rectangle

4.3.1 Unfilled rectangle

The unfilled rectangle differs from the unfilled square only insofar as the length is specified to be necessarily *twice* the width, thus generating an archetypal rectangle. Implementation, therefore, entails multiplying the variable `size` by a factor of 2 when shifting y-coordinates.

```
1  void print_rectangle(position_t* pos, double size) {
2
3          position_t pos2;
4          init_position(&pos2);
5
6          pos2 = *pos;
7          pos2.x = pos->x + size;
8
9          print_line(pos, &pos2);
10
11         *pos = pos2;
12         pos2.y = pos->y + 2*size;
13
14         print_line(pos, &pos2);
15
16         *pos = pos2;
17         pos2.x = pos->x - size;
18
19         print_line(pos, &pos2);
```

```
20
21          *pos = pos2;
22          pos2.y = pos->y - (2*size - 0.5);
23
24          print_line(pos, &pos2);
25
26          }
```

4.3.2 Filled rectangle

The required number of iterations to fill a rectangle, consistent with that of the square, is `size*size`. In order to avoid the problems detailed in Section 5, however, the rectangle is instead filled simply by generating two filled squares, possible by virtue of the rectangle's length being necessarily defined as twice its width.

```
1   void print_filled_rectangle(position_t* pos, double size) {
2
3           printf(" ; Printing filled rectangle at (%f,%f,%f) of width
                %f and height %f.\n", pos->x, pos->y, pos->z, size, 2*
               size);
4
5           position_t pos2;
6           init_position(&pos2);
7
8           pos2 = *pos;
9
10      /* Generate a filled rectangle by generating a composite of two
            filled squares */
11          print_filled_square(pos, size);
12      *pos = pos2;
13          pos2.y = pos->y + size;
14      print_filled_square($pos2, size);
15
16          }
```

4.4 Cube

4.4.1 Unfilled cube

Both unfilled and filled cubes are remarkably straightforwardly generable, precisely by initially printing the relevant square, iterating along the z-axis by shifting a height of 0.5mm (due to the uniform thickness of the extruded material being 0.5mm), and regenerating an identical square. Thus, the unfilled cube consists of multiple layers (precisely 10 in the below listings) of unfilled squares; analogously, the filled cube consists of multiple layers of filled squares.

```
1   void print_cube(position_t* pos, double size) {
2
```

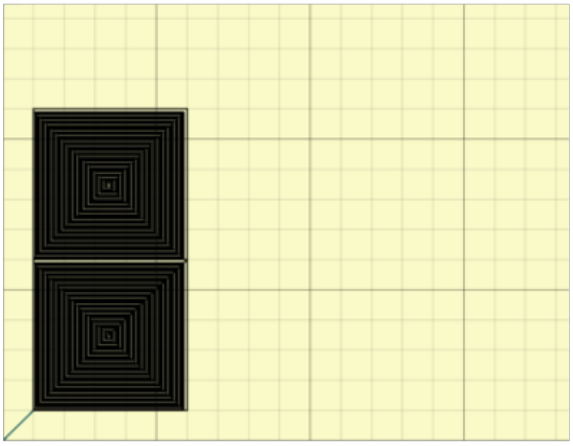

Figure 7: `Pronterface`-generated preview for a filled rectangle.

```
3           printf(" ; Printing Cube at (%f,%f,%f) of size %f.\n", pos
                ->x, pos->y, pos->z, size);
4
5           position_t pos2;
6           init_position(&pos2);
7
8           pos2 = *pos;
9
10          for (int i = 0; i < 10; i++) {
11
12               print_square(pos, size);
13              *pos    = pos2;
14               pos2.z = pos->z + 0.5;
15
16                  }
17
18          }
```

4.4.2 Filled cube

```
1   void print_filled_cube(position_t* pos, double size) {
2
3           printf(" ; Printing filled Cube at (%f,%f,%f) of size %f.\n
                ", pos->x, pos->y, pos->z, size);
4
5           position_t pos2;
6           init_position(&pos2);
```

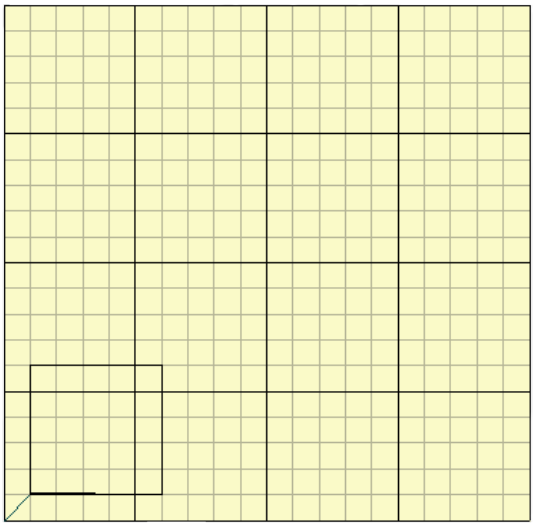

Figure 8: `Pronterface`-generated preview for an unfilled cube.

```
7
8           pos2 = *pos;
9
10          for (int i = 0; i < 10; i++) {
11
12               print_filled_square(pos, size);
13              *pos    = init_pos2;                                    //
                    Set position to the outer perimeter, thus reseting
                    the nozzle for continued iterations
14               pos2.z = pos->z + 0.5;
15
16                  }
17
18          }
```

4.5 Cuboid

With the ultimate composition of the Lego brick borne in mind, the *required* cuboid remains unfilled, as the brick is designed to be hollow. Thus, the generation of a filled cuboid would be redundant, if certainly feasible; the filled cuboid has its obvious analogue in the filled cube, with multiple filled rectangle functioning as the layers rather than squares.

By iterating along the z-axis for $^{size}/_{2}$ iterations, one obtains the general cuboid form that serves as a basis for the composite brick. This is possible precisely

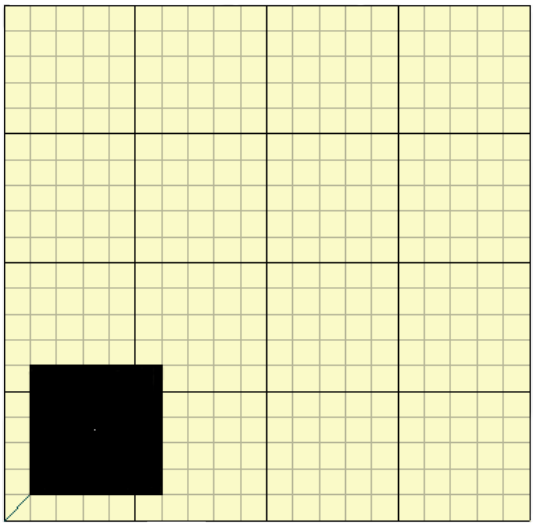

Figure 9: `Pronterface`-generated preview for a filled cube.

in the manner in which a hollow cube is generable, with this iterative limit specified:

```
1  void print_cuboid(position_t* pos, double size) {
2
3          printf(" ; Printing cuboid at (%f,%f,%f) of width %f,
               length %f and height %f.\n", pos->x, pos->y, pos->z,
               size, 2*size, size);
4
5          position_t pos2;
6          init_position(&pos2);
7
8          pos2 = *pos;
9
10         for (int i = 0; i < size/2; i++) {
11
12             print_rectangle(pos, size);
13            *pos    = pos2;
14             pos2.z = pos->z + 0.5;
15
16                 }
17
18         }
```

4.6 Circle

4.6.1 Filled circle

The mathematical principle implemented in order to generate a filled circle concerns finding all point in 2D space within a circle's perimeter, given a radius.

The distance between a relative origin (x_c, y_c), and some points (x_p, y_{cp}) on the perimeter is given by the Pythagorean theorem

$$d = \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2}.$$

Thus, the point (x_p, y_p) is contained within the circle if $d < r$, or alternatively, if $d^2 < r^2$. This principle can be implemented in `C` by using nested `for`-loops, identifying all such points that satisfy the condition `((x1*x1 + y1*y1) <= radius*radius)` and printing lines between these points iteratively:

```
1  void print_filled_circle(position_t* pos, double radius, int i, int
       j) {
2
3          printf(" ; Printing circle at (%f,%f,%f) of radius %f.\n",
               pos->x, pos->y, pos->z, radius);
4
5          position_t pos2;
6          var_position(&pos2, i, j);
7
8          pos2 = *pos;
9
10         for (int y1 = -radius; y1 <= radius; y1++) {
11              for (int x1 = -radius; x1 <= radius; x1++) {
12                          if ((x1*x1 + y1*y1) <= radius*radius) {
13                          *pos    = pos2;
14                           pos2.x = i + x1+x1;
15                           pos2.y = j + y1+y1;
16                            print_line(pos, &pos2);
17                             }
18                      }
19              }
20         }
```

4.7 Cylinder

4.7.1 Filled cylinder

The filled cylinder is constructed from filled circles in the manner aforedescribed for the filled cube; for the composite brick in particular, the number of iterations is assumed to be 3, an appropriate limit in relation to the measurements of the cuboid.

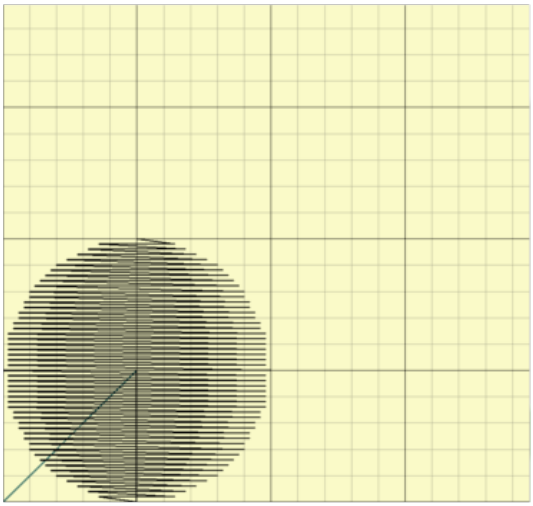

Figure 10: `Pronterface`-generated preview for a filled circle.

```
1  void print_filled_cylinder(position_t* pos, double radius) {
2
3          printf(" ; Printing filled cylinder at (%f,%f,%f) of radius
               %f.\n", pos->x, pos->y, pos->z, radius);
4
5          position_t pos2;
6          init_position(&pos2);
7
8          pos2 = *pos;
9
10         for (int i = 0; i < 3; i++) {
11
12              print_filled_circle(pos, radius);
13             *pos    = init_pos2;
14              pos2.z = pos->z + 0.5;
15
16                 }
17
18         }
```

4.8 Composite brick

The composite brick shown in 2 is constructed from a partially-filled *cube*; the desirable, archetypal brick is alternatively constructed from a partially-filled *cuboid*. Due to proportionality, the cylindrical layers of the cuboid brick can

be constructed by calling the same method that produces the cylindrical layers of the cube brick – that is, by virtue of the cuboid having the dimensions $size \times 2size$.

```
1  void print_lego_brick(position_t* pos, double size) {
2
3        printf(" ; Printing lego brick at (%f,%f,%f) of width and
             height %f, length %f.\n", pos->x, pos->y, pos->z, size,
             2*size);
4        position_t pos2;
5        init_position(&pos2);
6
7        pos2 = *pos;
8        print_cuboid(pos, size);
9        print_filled_rectangle(pos, size);
10
11       for (int i = 1; i < 2*size; i + 2) {
12                for (int j = 1; j < size; j + 2 ) {
13
14                     *pos    = pos2;
15                          pos2.x = pos->((j*size)/4);
16                          pos2.y = pos->((i*size)/4);
17                          print_filled_cylinder(pos, radius);
18
19                          }
20                }
21       }
```

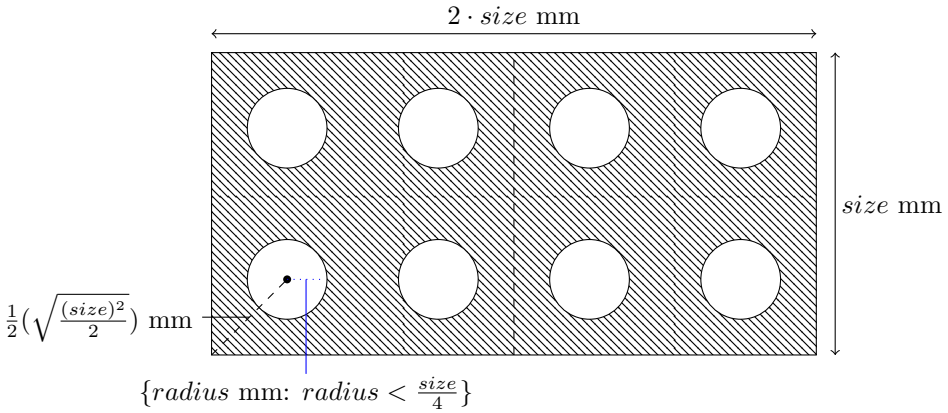

Figure 11: Determining the point of origin and radius constrictions required to generate a composite brick.

5 Testing

The most basic analysis undertaken of the performance and useability of a single generated geometric form concerned the unfilled square. An early printing

attempt revealed that, with no programmatic considerations for the physical properties of the printer equipment, the origin point wherefrom a square is constructed is visited precisely twice, and thus has approximately twice the depth of any other point on the square upon completion. If left unconsidered, this fact would compromise the degree of useability of forms built based upon the square. Thus, to accomodate this physical property, the final drawn line in the square is of length 0.5mm less than all other drawn lines, as this distance is precisely the uniform width of the extruded plastic.

```
1     *pos = pos2;
2          pos2.y = pos->y - (size - 0.5);
```

Concerning the relationship between individual methods, a number of errors were encountered before an effective solution was obtained. The models observable in *Fig.* 12 and *Fig.* 13, for instance, resulted from a misapplication of the iterative process required to fill a square in the manner set forth in the design scheme.

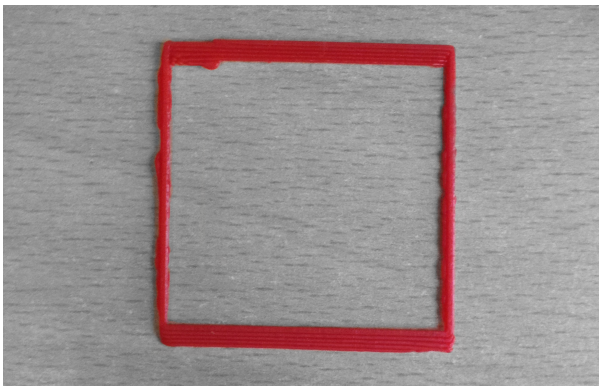

Figure 12: Unsuccessful attempt at producing a filled square.

When the succesful method used to generate a filled square was applied to the rectangle without alteration, an impractical and undesirable pattern was generated, as shown in *Fig.* 14.

Considered to a significant extent was the method used in order to generate circles. The initial consideration assumed the form of earlier shapes' generation processes, insofar as the perimeter was to be drawn firstly, decremented in area by some appropriate degree and in re-drawn about the origin, repeating this process until a filled circle was obtained. Such an implemented method would thus accept the desired radius of the circle as a parameter. One such advantage to this considered method concerns its elements of simplicity; for instance, one can observe with no difficulty that the required number of iterations to generate a filled circle is precisely $2 \cdot radius$. This property can be inspected in *Fig.* 15.

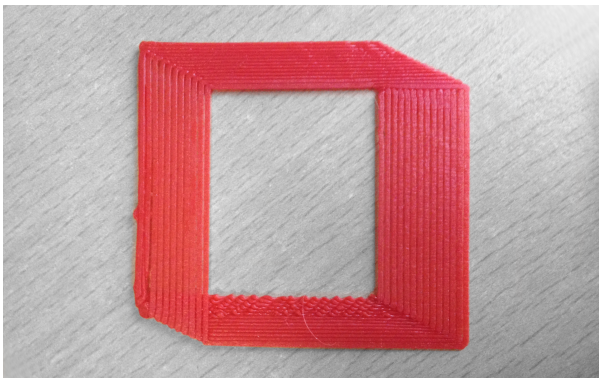

Figure 13: Another unsuccessful attempt at producing a filled square.

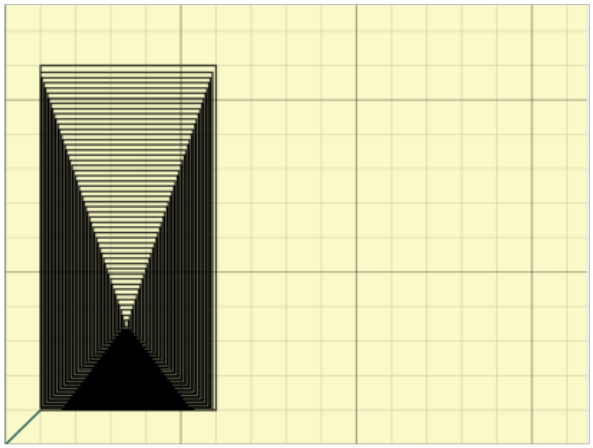

Figure 14: `Pronterface`-generated preview for an unsuccessfully filled rectangle.

This endeavour was ultimately discarded for the principle reason of complexity; generating an unfilled square, *i.e.,* calculating precisely those points that lie on the desired circle's perimeter, appeared to be a task of higher difficulty and complexity that calculating those points that lie *within* the circle's perimeter. Conducive to this observation, the geometric form of the unfilled cylinder, and thus by extension the unfilled circle, is not a necessary form for the construction of the cmoposite brick – thus, any such method to generate an unfilled circle (`print_circle`) was ultimately discarded.

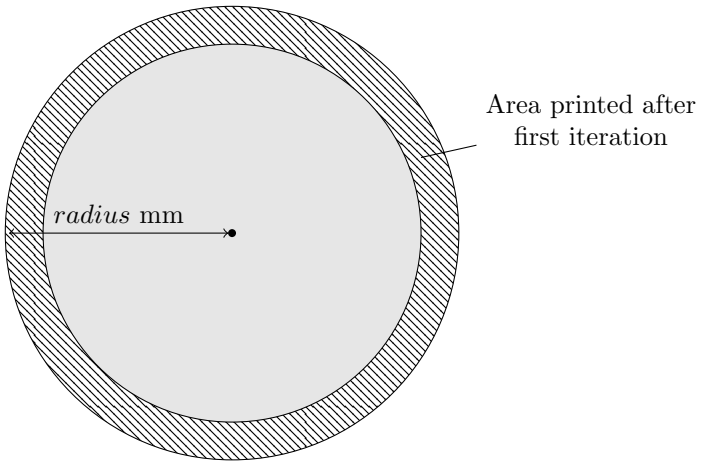

Figure 15: Determining the number of iterations required to generate a filled circle.

Considered for its efficiency, a further attempted method of generating circles employed the *midpoint algorithm,* an extension of *Bresenham's line algorithm* (*cf.* Sarcar et al. (2008), *pp.* 124–125). Although attempted to be implemented, this method was also discarded, foremostly for its ostensible inefficacy (*Fig.* 16) in producing the desired form but also for its for complexity and unmanageability.

The ultimately chosen method, therefore, concerns a 'brute-force' generation of a filled square by drawing very small lines, effectively points, about a specified origin contingent with a given user-inputted radius. Such an approach appears to be the least complex of the considered approaches.

In this regard, therefore, the initial design plans were not wholly adhered to, as testing the circle unit betrayed a weakness of design, in both efficiency and simplicity.

6 Evaluation

With concern to the aforestated requirements (2), the system generally performed favourably.

The unforeseen technical errors that introduced delay to the project's progression were largely constituted by the physical properties of the 3D printer technology, in contrast to the logistically independent dimension of programmatic generation. For instance, initial attempts to generate an unfilled square were ineffective, insofar as the printer head would extrude material over the initial

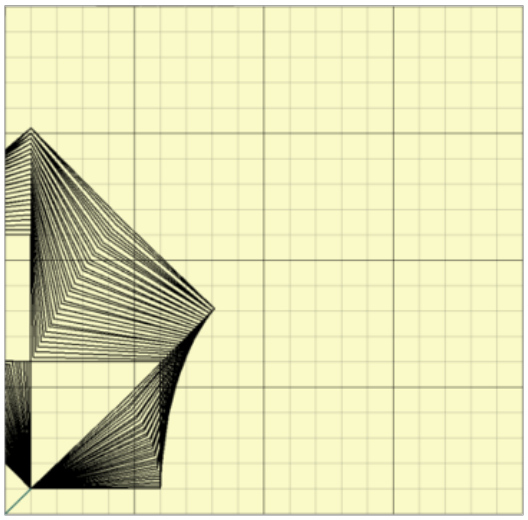

Figure 16: `Pronterface`-generated preview for an unsuccessful circle based on the midpoint algorithm.

starting point of the printer. This problem was circumvented via the methods described in Section 5.

Errors that existed solely as a result of poor implementation of methods in `C` include those shown in *Fig.* 11 and *Fig.* 12 as printed objects. Multiple ultimately ineffective efforts were made in attempt to generate a filled square, stemming from a misunderstanding of the integration of the *line* and *square* methods; these efforts are detailed in Section 5.

A principal weakness of the system concerns the *efficient* generation of shapes. As shown in *Fig.* 6, the implemented method for filling areas contained with shapes' perimeters functions by progressively decrementing the size of the initial shape, based on some user-inputted measurement, the iteration terminating when the area is filled to the extent possible. This method is sub-optimal as – whilst ultimately producing a robust and resilient filled shape – an unnecessarily large quantity of material is extruded, compromising efficiency and contributing also the weight of the model, a distinct disadvantage from an industrial perspective.

This compromise in efficiency could be eschewed by favouring lattices over the above described iterations; for instance, the lattice-based methods discussed in Wang et al. (2013) and the honeycomb fractals discussed in Lu et al. (2014) would provide a lightweight, efficient alternative, should these methods be implemented algorithmically in `C`.

Also in this regard, however, the generated shapes are of no lesser resilience, uniformity and completion than those lattice-based shapes generable via the utilisation of fractal algorithms. Thus, the system can be said to satisfy the requirement criteria pertinent to these aforementioned properties.

7 Conclusion

A programmatic generation of a composite brick is believed to have been theoretically achieved, independent of a physical model thereof having been printed; it is therefore determined to be a feasible practical endeavour, by virtue of building upon elementary geometric shapes generated by implemented `C` methods and calling these methods in a certain sequential manner.

The designed – and subsequently programmatically implemented – composite brick lacks the form of archetypal Lego bricks insofar as the bricks are not able to be stacked or otherwise joined in the conventional manner. Thus, a possible further improvement on the detailed results could be alter the present composite brick in order to comply to this requirement, by generating and printing a section positioned at the lower portion of the brick that could connect to the present cylindrical section atop the brick. This could feasibly be accomplished in a manner almost identical to the current sequence of programmatic operations responsible for generating the protruding cylindrical sections featured on the present brick; this sequence of operations would be issued at the base level of the brick's construction, excepting the operation that increments the print header's respective z-axis. Another dissimilarity between this possible advancement and the present construction concerns the nature of the generated cylinders; the new cylinders would need to be unfilled to accomodate the protruding cylindrical sections of other printed bricks, and, due to the physical dimensional properties of the extruded material, would similarly be required to have a greater radius in order for this accomodation to be feasible.

As presently designed, the methods concerned with generating rectangles (and thus cuboids) assume the length of the rectangle to be precisely twice the inputted width. This design aspect is conceivably undesirable, and thus a possible development would be specify an additional parameter accepted by these methods that should thus generate a dimensionally variable – if atypical – rectangle or cuboid model. Ostensibly, the optimal way to approach this task would be to restrict the inputted length to some multiple of $^{size}/_{2}$ – where $size$ represents the inputted width – in order to ensure the brick's rectangular surface can practicably accomodate some columnal multiple of cylinders.

Further, *arbitrary shapes* could be considered; that is, contrary the the present constant iteration of geometrically homogeneous shapes, the dimensions of shapes generated along the$z$ axis could be variable as such to create an arbitrary composite. One could imagine a four-sided pyramid structure being generable in this

fashion by modifying the existant iterative method responsible for production of *cube* structures.

The generated 3-dimensional structures, both actualised and theoreticised, bear practical significance to real-world systems engineering. It has been shown interpretable sequences of G-codes can be produced from concise, logical and user-readable `C`-based interfaces. These developments therefore contribute towards an increasingly more accessible, manageable and utilisable approach to 3-dimensional printing.

References

Jones, R. et al. (2011). RepRap – The Replicating Rapid Prototyper. *Robotica*, 29(01):177–191.

Lu, L. et al. (2014). Build-to-Last: Strength to Weight 3D Printed Objects. *ACM Transactions on Graphics (TOG)*, 33(4):97.

Sarcar, M. M. M., Rao, K. M., and Narayan, K. L. (2008). *Computer Aided Design and Manufacturing*. PHI Learning Pvt. Ltd.

Wang, W. et al. (2013). Cost-effective Printing of 3D Objects with Skin-frame Structures. *ACM Transactions on Graphics (TOG)*, 32(6):177.

A
Program Listings

```
1  #include "common.h"
2  #include "line.h"
3
4  #include <math.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  int line_main(int argc, char** argv) {
9
10         printf(" ; Generating a line\n");
11
12         if (argc != 6) {
13                 fprintf(stderr, "Invalid number of arguments (%d)\n
                       ", argc);
14                 return 1;
15                 }
16
17         position_t from, to;
18
19         // C structs need to be initialised to some default values.
20         // Here we set them to 0.
21         init_position(&from);
22         init_position(&to);
23
24         // Convert the string provided on the command line to a
               double.
25         from.x = strtod(argv[2], NULL);
26         from.y = strtod(argv[3], NULL);
27         to.x = strtod(argv[4], NULL);
28         to.y = strtod(argv[5], NULL);
29
30         print_start();
31         print_move_to(&from);
32         print_line(&from, &to);
33         print_end();
34
35         return 0;
36
37         }
38
39 void print_line(position_t* from, const position_t* to) {
40
41         double ext = hypot(from->x - to->x, from->y - to->y)/8;
42
43         //printf("G1 Z0.45\n");
44         printf("G1 X%f Y%f E%f\n", to->x, to->y, ext);
45
46         // Update the current location of the nozzle.
47         *from = *to;
48
49         }
```

Listing 1: `line.c`

```
1  #include "common.h"
2  #include "line.h"
3  #include "square.h"
4
5  #include <math.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8
9  typedef void (*printfn_t)(position_t*, double);
10
11 static int square_main_common(int argc, char** argv, printfn_t
       print_function) {
12
13         if (argc != 5) {
14                 fprintf(stderr, "Invalid number of arguments (%d)\n
                       ", argc);
15                 return 1;
16                 }
17
18         position_t pos;
19         init_position(&pos);
20
21         pos.x = strtod(argv[2], NULL);
22         pos.y = strtod(argv[3], NULL);
23         const double size = strtod(argv[4], NULL);
24
25         print_start();
26         print_move_to(&pos);
27         print_function(&pos, size);
28         print_end();
29
30         return 0;
31
32         }
33
34 int square_main(int argc, char** argv) {
35
36         printf(" ; Generating a square\n");
37         return square_main_common(argc, argv, &print_square);
38
39         }
40
41 int filled_square_main(int argc, char** argv) {
42
43         printf(" ; Generating a filled square\n");
44         return square_main_common(argc, argv, &print_filled_square)
               ;
45
46         }
47
48 void print_square(position_t* pos, double size) {
49
50         position_t pos2;
51         init_position(&pos2);
52
53         pos2 = *pos;
54         pos2.x = pos->x + size;
```

```
55
56          print_line(pos, &pos2);
57
58      *pos = pos2;
59          pos2.y = pos->y + size;
60
61          print_line(pos, &pos2);
62
63      *pos = pos2;
64          pos2.x = pos->x - size;
65
66          print_line(pos, &pos2);
67
68      *pos = pos2;
69          pos2.y = pos->y - (size - 0.5);
70
71          print_line(pos, &pos2);
72
73          //printf("G1 X%f Y%f Z%f E%f\n", pos->x, pos->y, pos->z,
                size);
74
75          // You can assume that the nozzle is at the position
                provided.
76          // *pos contains coordinates the nozzle is located at the
                end of the function.
77
78          }
79
80  void print_filled_square(position_t* pos, double size) {
81
82          printf(" ; Printing Filled Square at (%f,%f,%f) of size %f
                .\n", pos->x, pos->y, pos->z, size);
83
84          position_t pos2;
85          init_position(&pos2);
86
87          pos2 = *pos;
88          init_pos2 = pos2;
89
90          for (int i = 0; i < (size*size); i++) {
91
92               print_square(pos, size);
93               size--;
94
95              *pos    = pos2;
96               pos2.x = pos->x + 0.5;
97               pos2.y = pos->y + 0.5;
98
99               }
100
101         }
```

Listing 2: square.c

```
1   #include "common.h"
2   #include "line.h"
3   #include "square.h"
```

```
4
5   #include <math.h>
6   #include <stdlib.h>
7   #include <stdio.h>
8
9   typedef void (*printfn_t)(position_t*, double);
10
11  static int cube_main_common(int argc, char** argv, printfn_t
        print_function) {
12
13          if (argc != 5) {
14                  fprintf(stderr, "Invalid number of arguments (%d)\n
                        ", argc);
15                  return 1;
16                  }
17
18          position_t pos;
19          init_position(&pos);
20
21          pos.x = strtod(argv[2], NULL);
22          pos.y = strtod(argv[3], NULL);
23          const double size = strtod(argv[4], NULL);
24
25          print_start();
26          print_move_to(&pos);
27          print_function(&pos, size);
28          print_end();
29
30          return 0;
31
32          }
33
34  int cube_main(int argc, char** argv) {
35
36          printf(" ; Generating a cube\n");
37          return cube_main_common(argc, argv, &print_cube);
38
39          }
40
41  int filled_cube_main(int argc, char** argv) {
42
43          printf(" ; Generating a filled cube\n");
44          return cube_main_common(argc, argv, &print_filled_cube);
45
46          }
47
48  void print_cube(position_t* pos, double size) {
49
50          printf(" ; Printing Cube at (%f,%f,%f) of size %f.\n", pos
                ->x, pos->y, pos->z, size);
51
52          position_t pos2;
53          init_position(&pos2);
54
55          pos2 = *pos;
56
57          for (int i = 0; i < 10; i++) {          // 10 iterations is
```

```
                an approximation for the height of the cube
58
59              print_square(pos, size);
60             *pos    = pos2;
61              pos2.z = pos->z + 0.5;
62
63                 }
64
65          }
66
67  void print_filled_cube(position_t* pos, double size) {
68
69          printf(" ; Printing filled Cube at (%f,%f,%f) of size %f.\n
                ", pos->x, pos->y, pos->z, size);
70
71          position_t pos2;
72          init_position(&pos2);
73
74          pos2 = *pos;
75
76          for (int i = 0; i < 10; i++) {
77
78               print_filled_square(pos, size);
79              *pos    = init_pos2;                                    //
                    Set position to the outer perimeter, thus reseting
                    the nozzle for continued iterations
80              pos2.z = pos->z + 0.5;
81
82                 }
83
84          }
```

Listing 3: cube.c

```
1   #pragma once
2
3   #include "common.h"
4
5   int rectangle_main(int argc, char** argv);
6   int filled_rectangle_main(int argc, char** argv);
7
8   void print_rectangle(position_t* pos, double size);
9   void print_filled_rectangle(position_t* pos, double size);
10
11  int cuboid_main(int argc, char** argv); int Lego_brick_main(int
        argc, char** argv);
12  void print_cuboid(position_t* pos, double size); void
        print_Lego_brick(position_t* pos, double radius);
13
14  int lego_brick_main(int argc, char** argv);
15  void print_lego_brick(position_t* pos, double radius);
```

Listing 4: rectangle.h; explain here

```
1   #include "common.h"
2   #include "line.h"
3   #include "square.h"
```

```
4  #include "rectangle.h"
5
6  #include <math.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9
10 typedef void (*printfn_t)(position_t*, double);
11
12 static int rectangle_main_common(int argc, char** argv, printfn_t
       print_function) {
13          if (argc != 5) {
14              fprintf(stderr, "Invalid number of arguments (%d)\n",
                    argc);
15              return 1;
16              }
17
18          position_t pos;
19          init_position(&pos);
20
21          pos.x = strtod(argv[2], NULL);
22          pos.y = strtod(argv[3], NULL);
23          const double size = strtod(argv[4], NULL);
24
25          print_start();
26          print_move_to(&pos);
27          print_function(&pos, size);
28          print_end();
29
30          return 0;
31
32          }
33
34 int rectangle_main(int argc, char** argv) {
35
36          printf(" ; Generating a rectangle\n");
37          return rectangle_main_common(argc, argv, &print_rectangle);
38
39          }
40
41 int filled_rectangle_main(int argc, char** argv) {
42
43          printf(" ; Generating a filled rectangle\n");
44          return rectangle_main_common(argc, argv, &
                print_filled_rectangle);
45
46          }
47
48 void print_rectangle(position_t* pos, double size) {
49
50          position_t pos2;
51          init_position(&pos2);
52
53          pos2 = *pos;
54          pos2.x = pos->x + size;
55
56          print_line(pos, &pos2);
57
```

```
58          *pos = pos2;
59          pos2.y = pos->y + 2*size;
60
61          print_line(pos, &pos2);
62
63          *pos = pos2;
64          pos2.x = pos->x - size;
65
66          print_line(pos, &pos2);
67
68          *pos = pos2;
69          pos2.y = pos->y - (2*size - 0.5);
70
71          print_line(pos, &pos2);
72
73          }
74
75  void print_filled_rectangle(position_t* pos, double size) {
76
77          printf(" ; Printing filled rectangle at (%f,%f,%f) of width
                 %f and height %f.\n", pos->x, pos->y, pos->z, size, 2*
                size);
78
79          position_t pos2;
80          init_position(&pos2);
81
82          pos2 = *pos;
83
84      /* Generate a filled rectangle by generating a composite of two
             filled squares */
85          print_filled_square(pos, size);
86      *pos = pos2;
87          pos2.y = pos->y + size;
88      print_filled_square($pos2, size);
89
90          }
```

Listing 5: rectangle.c

```
1   #include "common.h"
2   #include "line.h"
3   #include "square.h"
4   #include "rectangle.h"
5
6   #include <math.h>
7   #include <stdlib.h>
8   #include <stdio.h>
9
10  typedef void (*printfn_t)(position_t*, double);
11
12  static int cuboid_main_common(int argc, char** argv, printfn_t
        print_function) {
13          if (argc != 5) {
14              fprintf(stderr, "Invalid number of arguments (%d)\n",
                    argc);
15              return 1;
16              }
```

```
17
18          position_t pos;
19          init_position(&pos);
20
21          pos.x = strtod(argv[2], NULL);
22          pos.y = strtod(argv[3], NULL);
23          const double size = strtod(argv[4], NULL);
24
25          print_start();
26          print_move_to(&pos);
27          print_function(&pos, size);
28          print_end();
29
30          return 0;
31
32          }
33
34  int cuboid_main(int argc, char** argv) {
35
36          printf(" ; Generating a cuboid\n");
37          return rectangle_main_common(argc, argv, &print_cuboid);
38
39          }
40
41  void print_cuboid(position_t* pos, double size) {
42
43          printf(" ; Printing cuboid at (%f,%f,%f) of width %f,
                length %f and height %f.\n", pos->x, pos->y, pos->z,
                size, 2*size, size);
44
45          position_t pos2;
46          init_position(&pos2);
47
48          pos2 = *pos;
49
50          for (int i = 0; i < size/2; i++) {
51
52              print_rectangle(pos, size);
53             *pos    = pos2;
54              pos2.z = pos->z + 0.5;
55
56                  }
57
58          }
```

Listing 6: cuboid.c

```
1   #pragma once
2
3   #include "common.h"
4
5   int circle_main(int argc, char** argv);
6   int filled_circle_main(int argc, char** argv);
7
8   //void print_circle(position_t* pos, double radius);
9   void print_filled_circle(position_t* pos, double radiu, int i, int
        j);
```

```
10
11 int cylinder_main(int argc, char** argv);
12 int filled_cylinder_main(int argc, char** argv);
13
14 //void print_cylinder(position_t* pos, double radius);
15 void print_filled_cylinder(position_t* pos, double radius, int j,
       int k);
```

Listing 7: `circle.h`

```
1  #include "common.h"
2  #include "line.h"
3  #include "circle.h"
4
5  #include <math.h>
6  #include <stdlib.h>
7  #include <stdio.h>
8
9  typedef void (*printfn_t)(position_t*, double);
10
11 static int circle_main_common(int argc, char** argv, printfn_t
       print_function) {
12
13         if (argc != 5) {
14             fprintf(stderr, "Invalid number of arguments (%d)\n",
                   argc);
15             return 1;
16             }
17
18         position_t pos;
19         init_position(&pos);
20
21         pos.x = strtod(argv[2], NULL);
22         pos.y = strtod(argv[3], NULL);
23         const double radius = strtod(argv[4], NULL);
24
25         print_start();
26         print_move_to(&pos);
27         print_function(&pos, radius);
28         print_end();
29
30         return 0;
31
32         }
33
34 int filled_circle_main(int argc, char** argv) {
35
36         printf(" ; Generating a filled circle\n");
37         return circle_main_common(argc, argv, &print_filled_circle)
               ;
38
39         }
40
41 void print_filled_circle(position_t* pos, double radius, int i, int
        j) {
42
43         printf(" ; Printing circle at (%f,%f,%f) of radius %f.\n",
```

```
            pos->x, pos->y, pos->z, radius);
44
45          position_t pos2;
46          var_position(&pos2, i, j);
47
48          pos2 = *pos;
49
50          for (int y1 = -radius; y1 <= radius; y1++) {
51               for (int x1 = -radius; x1 <= radius; x1++) {
52                         if ((x1*x1 + y1*y1) <= radius*radius) {
53                         *pos    = pos2;
54                          pos2.x = i + x1+x1;
55                          pos2.y = j + y1+y1;
56                           print_line(pos, &pos2);
57                            }
58                    }
59               }
60          }
```

Listing 8: `circle.c`

```
1   #include "common.h"
2   #include "line.h"
3   #include "circle.h"
4   #include "cylinder.h"
5
6   #include <math.h>
7   #include <stdlib.h>
8   #include <stdio.h>
9
10  typedef void (*printfn_t)(position_t*, double);
11
12  static int cylinder_main_common(int argc, char** argv, printfn_t
        print_function) {
13
14          if (argc != 5) {
15              fprintf(stderr, "Invalid number of arguments (%d)\n",
                    argc);
16              return 1;
17              }
18
19          position_t pos;
20          init_position(&pos);
21
22          pos.x = strtod(argv[2], NULL);
23          pos.y = strtod(argv[3], NULL);
24          const double radius = strtod(argv[4], NULL);
25
26          print_start();
27          print_move_to(&pos);
28          print_function(&pos, radius);
29          print_end();
30
31          return 0;
32
33          }
34
```

```
35  int cylinder_main(int argc, char** argv) {
36
37          printf(" ; Generating a cylinder\n");
38          return cylinder_main_common(argc, argv, &print_cylinder);
39
40          }
41
42  int filled_cylinder_main(int argc, char** argv) {
43
44          printf(" ; Generating a filled cylinder\n");
45          return cylinder_main_common(argc, argv, &
                print_filled_cylinder);
46
47          }
48
49  void print_filled_cylinder(position_t* pos, double radius, int j,
        int k) {
50
51          printf(" ; Printing cylinder at (%f,%f,%f) of radius %f.\n"
                , pos->x, pos->y, pos->z, radius);
52
53          position_t pos2;
54          init_position(&pos2);
55
56          pos2 = *pos;
57
58          for (int i = 0; i < 3; i++) { // 3 iterations is an
                approximation for the height of the cylinder
59
60               print_circle(pos, radius, j, k);
61              *pos    = pos2;
62               pos2.z = pos->z + 0.5;
63
64               }
65
66          }
```

Listing 9: `cylinder.c`

```
1   #include "common.h"
2   #include "line.h"
3   #include "square.h"
4   #include "rectangle.h"
5   #include "circle.h"
6
7   #include <math.h>
8   #include <stdlib.h>
9   #include <stdio.h>
10
11  int lego_brick_main(int argc, char** argv) {
12
13          printf(" ; Generating a Lego brick\n");
14          return cuboid_main_common(argc, argv, &print_Lego_brick);
15
16          }
17
18  void print_lego_brick(position_t* pos, double size) {
```

```
19
20        printf(" ; Printing Lego brick at (%f,%f,%f) of width and
              height %f, length %f.\n", pos->x, pos->y, pos->z, size,
              2*size);
21        position_t pos2;
22        init_position(&pos2);
23
24        pos2 = *pos;
25        print_cuboid(pos, size);
26        print_filled_rectangle(pos, size);
27
28            /* the radius is assumed to be (size/6) for neatness, and
                   to prevent overlap */
29            double radius = (size/6);
30
31        for (int i = 1; i < 2*size; i + 2) {
32                  for (int j = 1; j < size; j + 2 ) {
33
34                      *pos    = pos2;
35                           pos2.x = pos->x + ((j*size)/4);
36                           pos2.y = pos->y + ((i*size)/4);
37                           print_filled_cylinder(pos, radius, pos2.x,
                                 pos2.y);
38
39                           }
40                  }
41        }
```

Listing 10: `lego_brick.c`

```
1  #include "line.h"
2  #include "square.h"
3  #include "circle.h"
4  #include "cylinder.h"
5  #include "rectangle.h"
6  #include "cube.h"
7  #include "cuboid.h"
8
9  #include <stdio.h>
10 #include <string.h>
11
12 int main(int argc, char** argv) {
13         if (argc > 1) {
14                 const char* const name = argv[1];
15
16                 if (strcmp(name, "line") == 0) {
17                         return line_main(argc, argv);
18                         }
19                 else if (strcmp(name, "square") == 0) {
20                         return square_main(argc, argv);
21                         }
22                 else if (strcmp(name, "filled-square") == 0) {
23                         return filled_square_main(argc, argv);
24                         }
25                 else if (strcmp(name, "cube") == 0) {
26                         return cube_main(argc, argv);
27                         }
```

```
28                  else if (strcmp(name, "circle") == 0) {
29                          return circle_main(argc, argv);
30                          }
31                  else if (strcmp(name, "cylinder") == 0) {
32                          return cylinder_main(argc, argv);
33                          }
34                  else if (strcmp(name, "rectangle") == 0) {
35                          return rectangle_main(argc, argv);
36                          }
37                  else if (strcmp(name, "filled-rectangle") == 0) {
38                          return filled_rectangle_main(argc, argv);
39                          }
40                  else if (strcmp(name, "cuboid") == 0) {
41                          return cuboid_main(argc, argv);
42                          }
43                  else if(strcmp(name, "lego-brick") == 0) {
44                          return lego_brick_main(argc, argv);
45                          }
46                  else {
47                          fprintf(stderr, "Unknown shape '%s'\n",
                                name);
48                          }
49                  }
50          else {
51                  fprintf(stderr, "Incorrect number of parameters
                        provided.\n");
52                  }
53
54          return 0;
55
56          }
```

Listing 11: `main.c`

```
1   #include "common.h"
2
3   #include <stdio.h>
4
5   void init_position(position_t* pos) {
6           pos->x = pos->y = pos->z = 0;
7           }
8
9   void var_position(position_t* pos, double x, double y) {
10          pos->x = x;
11          pos->y = y;
12          }
```

Listing 12: `common.c` (fragment)

The midpoint algorithm was implemented as follows:

```
1   /**
2    * Implementation of the Midpoint circle algorithm, a
         generalisation
3    * of Bresenham's line algorithm
4    **/
5   void print_filled_circle(position_t* pos, double radius) {
```

```
6
7           printf(" ; Printing filled circle at (%f,%f,%f) of radius %
                f.\n", pos->x, pos->y, pos->z, radius);
8
9           position_t pos2;
10          init_position(&pos2);
11
12          pos2 = *pos;
13
14          int x1 = radius;
15          int y1 = 0;
16          int radiusError = (1 - x1);
17
18          while (x1 >= y1) {
19
20                *pos = pos2;
21
22           /**
23            * First quadrant: print lines from the origin to all
                  points
24            * on the circumference of the circle's first quadrant,
                  thus
25            * effectively filling in the circle
26            **/
27
28                 pos2.x = ( x1 + pos2.x );
29                 pos2.y = ( y1 + pos2.y );
30                 print_line(pos, &pos2);
31
32                 pos2.x = ( y1 + pos2.x );
33                 pos2.y = ( x1 + pos2.y );
34                 print_line(pos, &pos2);
35
36           /* Second quadrant */
37
38                 pos2.x = (-x1 + pos2.x );
39                 pos2.y = ( y1 + pos2.y );
40                 print_line(pos, &pos2);
41
42                 pos2.x = (-y1 + pos2.x );
43                 pos2.y = ( x1 + pos2.y );
44                 print_line(pos, &pos2);
45
46           /* Third quadrant */
47
48                 pos2.x = (-x1 + pos2.x );
49                 pos2.y = (-y1 + pos2.y );
50                 print_line(pos, &pos2);
51
52                 pos2.x = (-y1 + pos2.x );
53                 pos2.y = (-x1 + pos2.y );
54                 print_line(pos, &pos2);
55
56           /* Fourth quadrant */
57
58                 pos2.x = ( x1 + pos2.x );
59                 pos2.y = (-y1 + pos2.y );
```

```
60                print_line(pos, &pos2);
61
62                pos2.x = ( y1 + pos2.x );
63                pos2.y = (-x1 + pos2.y );
64                print_line(pos, &pos2);
65
66                y1++;
67
68                if  (radiusError < 0) {
69                     radiusError += 2 * y1 + 1;
70                     }
71                else {
72                     x1--;
73                     radiusError += 2 * (y1 - x1) + 1;
74                     }
75                }
76          }
```

Listing 13: Implementation of the midpoint algorithm in C