FlatCAD and FlatLang: Kits by Code

Gabe Johnson
Carnegie Mellon University

Mark D. Gross
Carnegie Mellon University

Abstract

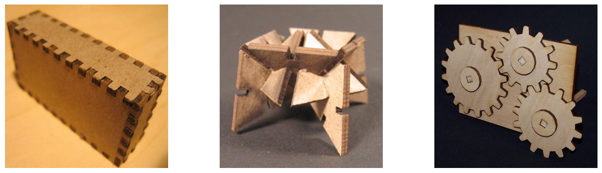
The FlatCAD system lets you create your own physical construction kits by coding in the LOGO-like FlatLang language. No longer must construction kit pieces be merely a product designed by someone else: if you can write a simple FlatLang program, you can design a kit. This paper describes our domain-specific language features and example output.

1 Introduction

Construction kits let you build complicated things out of simpler pieces. A typical set of LEGO bricks consist of plastic pieces that snap together vertically. Another popular kit, Tinker Toys, feature rigid struts that fit into round holes in hubs. Many more examples can be found at toy stores. Most kits present different types of pieces that vary in size, length, or the way they fit together. Despite the kits' simplicity, they support people in building creative, complex constructions.

In existing kits, individual pieces are immutable. But what if we were able to design new kinds of parts? Instead of building from the parts we are given, we could instead make new kinds of kits enabling us to work in different ways. To explore this question, we have developed FlatCAD, a prototype system that can be used to develop new kinds of construction kits.

FlatCAD is a design system for modeling and fabricating 3D objects using flat material like wood, acrylic, paper, or cardboard. These materials may be folded, layered, attached, and trimmed in various ways to create physical constructions. FlatCAD models are made by programming in a LOGO-like language called FlatLang. These models can then be 'printed' to rapid prototyping devices like laser cutters for manual assembly. Figure 1 shows several physical output examples.



(a) Parametric boxes whose faces attach with finger joints.

(b) A simple kit with notched pieces.

(c) Gears are one type of piece in mechanical kits.

Figure 1. Physical output of FlatCAD.

2 A Mechanical Construction Kit

Construction kits can be more than just existing physical pieces—they can be virtual as well. While a physical mechanical construction kit may come with four or five sizes of gears, a virtual mechanical kit can render any size gear we like. FlatCAD lets us bridge the virtual and physical by letting us easily fabricate our models.

Say we would like to make a toy ~~automata.~~ In particular, we ~~are going~~ to make a vehicle with a little puppet 'driving' it. We want the puppet to move up and down as the vehicle rolls on the floor. We must convert the wheel's radial motion to harmonic linear motion to move the puppet. We could build this out of existing parts, but the resulting automata would likely not be as nice as we would like. Instead, we can use FlatCAD and a library of parametric construction kit pieces and get exactly what we want.

One way of converting from angular to linear motion is by using a piston wheel, like the one shown in Figure **??**. ~~Such a part~~ has an off-center axle attached to a rigid strut. When the other end of the strut is constrained to move along a straight path, rotating the wheel causes the strut to move back and forth linearly. A program for expressing this mechanism is shown in Figure 2.

The simple toy automaton program uses default values. Sometimes the default values will produce something that won't work. Using defaults in this example, the strut is facing the wrong way. ~~This is~~ correct~~ed~~ by inserting a `left(90)` instruction before creating the strut. We can

```
; wheel -- piston_wheel -- wheel
;               |
;             strut
;               |
;             puppet

wheel_1 = wheel()
wheel_2 = wheel()
piston = piston_wheel()
strut = link()
piston.offcenter_object = strut

coaxial(wheel_1, piston, wheel_2)
```

Figure 2. A mechanism for a toy automaton using default dimensions and angles.

also change the parameterization of the parts to get behavior different from the default, for example changing the wheel's `radius` member variable.

The mechanism resulting from the program in Figure 2 will move the puppet, but its speed and the amount of displacement are constant. We can parameterize these properties by modifying the program. If we add gears, we can change the puppet's speed in relation to the forward motion of the vehicle. If we lengthen the distance between the piston wheel center and it's off-center axle, the puppet's vertical displacement will be greater. A revised version of this program is shown in Figure 3.

Of course, a finished toy automaton consists of more than mechanisms. ~~There must be~~ a chassis to hold the wheels and moving parts, which ~~can be made~~ by augmenting the program further. ~~In the end,~~ the FlatLang program lets us make more than just one particular automaton. By changing ~~numbers~~ we may easily produce the parts list for an entire class of toys.

3 Why Not Use Illustrator?

Typically people use commercial design software to make models ~~for production~~ on a laser cutter. ~~A fair question might be,~~ "Why not use Illustrator?" After all, designers use commercial systems all the time. Such systems let designers specify exact dimensions and angles. SolidWorks lets designers establish constraints ~~like~~ "X is halfway between A and B"—regardless of how A or B are manipulated, the system ~~will~~ ensure that X is between them. ~~Environments like~~ Maya or SketchUp provide scripting capabilities, so programs can directly generate or modify models. ~~Where scripting is available, there are good development environments.~~

Interaction modes found in these design environments can be separated into two groups: graphical and program-

```
; wheel -- gear -- wheel
;           |
;          gear
;           |
;      piston_wheel
;           |
;         strut
;           |
;         puppet

define parametric_automaton
  (g1_teeth, g2_teeth, offset)

  wheel_1 = wheel()
  wheel_2 = wheel()
  gear_1 = gear(g1_teeth)
  gear_2 = gear(g2_teeth)
  piston = piston_wheel(offset)
  left(90)
  strut = link()
  piston.offcenter_object = strut

  coaxial(wheel_1, mesh(gear_1,
          coaxial(gear_2, piston_wheel)),
          wheel_2)
done
```

Figure 3. A parametric version of the previous automaton.

matic. ~~Often a single~~ environment ~~can use~~ both of these interaction modes. Each mode has its ~~own~~ strengths and weaknesses.

In the graphical mode designers interact with the model primarily with the mouse. This involves the designer drawing lines, selecting and manipulating objects, or establishing relationships between them. This mode allows the designer to see their model in 2D or 3D as they work, and affords random access to anything already on the screen.

In the ~~programmatic~~ mode, people typically use scripts provided by the environment or third parties. Occasionally designers write their own scripts, but this requires programming skills. Depending on the language, learning to program may be seen as a lot of work for little payoff. Many environments use existing general purpose languages (or variants), providing access to the GUI's functionality via an API. For example, AutoCAD uses a variant of Lisp, SketchUp uses Ruby, and Maya uses MEL, a scripting language bearing a strong resemblance to Perl. Geometry in scripting environments is usually expressed in absolute terms.

In all of these cases, programming is supplemental to the primary graphics mode of interaction. In FlatCAD, programming is primary mode of interaction. It may be theoretically possible for any of these languages to do the same things. However, from the perspective of a human designer, there may be a significant difference between the expressive power of one language over another. Our language was designed for procedurally generating shapes for production on rapid prototyping machines. FlatLang uses differential (turtle) geometry, which is often easier to work with than absolute geometry. The language and the small set of built-in functions allow a designer to quickly design physical parts and systems of parts by writing short programs.

4 Related Work

The LOGO language provides a “microworld” to explore programming [6]. While LOGO is a complete programming language that works without graphics, it is particularly known for the ease that novice programmers can make visual output. LOGO’s graphics are generated with the use of two-dimensional “turtle geometry” [1], where graphics are drawn by instructions an on-screen “turtle” to move or turn various amounts. A recent LOGO-like language called FormWriter used a “flying turtle” that operated in 3D [3]. In addition to drawing lines, FormWriter had primitive drawing functions for creating 3D objects such as cones, cylinders, and boxes.

~~A main feature of~~ FlatCAD is ~~its use for~~ designing models for physical fabrication. The price of rapid prototyping machinery continues to drop while quality improves. ~~It is plausible that~~ soon most elementary schools will have prototyping machines such as 3D printers and laser cutters. The usefulness of these machines is limited by the software for creating models they will produce. Current CAD software is made for professional designers and may be inaccessible to most people. However, there is a growing community of researchers interested in making accessible software to empower ~~non-designers~~ to use prototyping machines.

Triskit pieces are simple wafer-like sheets cut from acrylic sheets and fit together at the edges with finger joints [4]. Users can design new Triskit pieces with arbitrary dimensions using a Java applet. The Furniture Factory and the Designosaur capture freehand sketch input that is used to generate dollhouse furniture and wooden dinosaur skeletons, respectively [5].

~~While the above~~ systems generate static kits, the MachineShop lets users design moving mechanical systems such as toy automata [2]. These systems are made from parts like gears and cams. Rather than directly editing the parameters or shape of such components, MachineShop users indicate behavioral qualities such as the distance a cam follower moves as the cam rotates. The system then generates a cam providing the desired behavior.

```
dihedralAngle = 125

define triangle(size)
  repeat(3)
    forward(size)
    left(120)
  done
done

define goNext(size)
  roll(-dihedralAngle)
  forward(size)
  right(90)
  roll(dihedralAngle)
done

roll(dihedralAngle)
repeat(4)
  triangle(3)
  goNext(3)
done
```

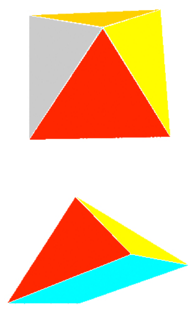


Figure 4. Low-level FlatLang code and graphics for a pentahedron, shown from two perspectives.

5 FlatLang Programming

FlatLang is a dynamically typed, interpreted language. The interpreter is written in Java, using ANTLR [7] to parse FlatLang source code. Its syntax resembles Python’s, though indentation is not significant. Execution begins at the top of input and proceeds from there–no `main()` function is required as in C or Java.

Figure 4 shows a simple FlatLang program for making a pentahedron. In this example, the `dihedralAngle` is declared and initialized to 125°. The variable is implicitly numeric. Next, the code declares two functions. `triangle` produces an equilateral triangle of parametric size. The `goNext` routine ‘rolls’ and positions the turtle for the next operation. The end of the code sample loops four times, explicitly creating four of the five faces of our square pyramid. The fifth face (the base of the pyramid) was created implicitly from the turtle’s path.

While low-level FlatLang code such as the pentahedron uses basic operations that control the turtle (like `forward`, `left` and `roll`), high-level FlatLang lets us write commands to generate complex objects subject to one-way constraints.

We have used FlatLang to develop a set of parametric mechanical parts such as gears and n-bar linkages. For example we may write `coaxial(gear(12), gear(24))` to make two gears that share an axis, where one gear has twelve teeth and the other with twice that number. Figure **??** shows a short FlatLang program for part of a mechanical system

```
coaxial(gear(10), piston_wheel())
link(4).draw()
```

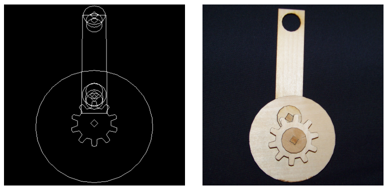


Figure 5. High-level FlatLang code, graphics, and physical output for a simple mechanism.

capable of driving a toy automata.

FlatLang models are intended to be ‘printed’ using rapid prototyping machines. The ~~on-screen representation~~ is different from the format used by computer-controlled fabrication machines like laser cutters. The on-screen representation shows how parts relate—the gear and piston wheel’s centers are at the same location in Figure **??**. However, when this is sent to a laser cutter, those parts must be separated and arranged to make reasonably efficient use of material. FlatLang provides the `part` command to begin a new logical collection of lines that are kept together when printing, but has no effect on the screen representation. This lets the programmer focus on creating systems of parts without manually separating them on the screen.

5.1 Turtle Tree

FlatCAD records the turtle’s activity in a data structure called the *Turtle Tree*. The nodes of the tree are turtle operations. A cursor stores the ‘current location’ that serves as the parent of the next operation.

There are three kinds of turtle operations: geometry, pen, and naming commands. Each operation is represented as a node in a tree structure. Geometry commands modify the turtle’s position or heading (e.g. `forward`, `left`, `roll`, and `pitch`). Pen commands (`up` and `down`) turn off and on the visual trail left behind when the turtle moves. Geometry and pen nodes may have at most one child node. Named nodes are inserted into the turtle tree with the `mark(s)` function. Name nodes may have any number of children.

To return to previously named position `s` the programmer uses the `backto(s)` command.

The behavior of the `backto` command could just as easily be done by reversing the turtle’s path. However, if the turtle has created a complicated shape (possibly using libraries written by other people), it may be very difficult for the programmer to know exactly how to undo the turtle’s path. The `backto` command allows designers to use subroutines that move the turtle without understanding exactly how they work.

5.2 Shapes

Procedures that generate a shape typically begin and end at the same locations. For example, the “triangle” function in Figure 4 always begins and ends at one corner. In most cases this is good enough, because we may never want to draw something in another way. However, we sometimes may want to draw a shape beginning from one particular location, or begin drawing subsequent parts from a different point. This commonly happens when we have a number of parts that can fit together in many ways.

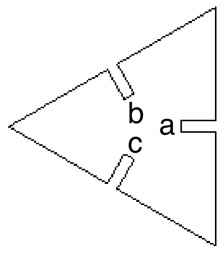
Consider the code in Figure 6. After the function definitions, it creates a globally visible shape called “tri”. The code in the `shape` block is executed but it is not appended to the model’s turtle tree. Instead, `shape` creates a separate circular list structure consisting of named nodes and geometric operations. The last node added to the list refers to the first node. We may use these named nodes as locations to begin drawing shapes using the `draw` and `from` commands.

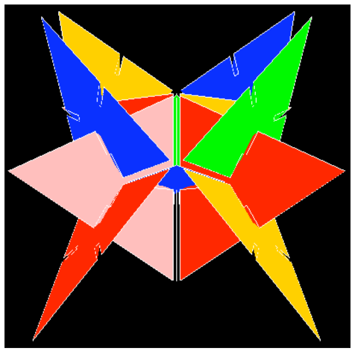
The `go` function first uses the `draw` command to draw a “tri” shape beginning from its ‘a’ location. This copies the sequence of operations from the “tri” list to the turtle tree. `draw` is helpful because it removes the need for the designer to know exactly which sequence of turtle operations is necessary to make the shape appear at the desired location. Next, the `from` command lets us position the turtle at the bottom of the other two notches in our triangle and draw additional pieces.

5.3 Absolute geometric commands

Most geometric turtle commands in FlatLang are interpreted relative to the turtle’s current position. Users may also use absolute geometric commands. The `pos` and `dir` commands return the absolute turtle position and directions. Programs may return the turtle to previously stored positions or direction with the `drawto` and `facedir` commands.

The absolute geometry commands are useful in cases when we are interested in connecting points but we are not able to (or do not care to) calculate the differential geometry between points. There are two primary difference between the `mark`/`backto` and `pos`/`drawto` approaches. The first is that geometric operations inherit their pen state from their parent in the turtle tree. While `backto` results in a new branch in the turtle tree, `drawto` does not. Second, `drawto` draws a line when the pen is down, but `backto` will never draw a line.





```
define notched_tri(len, dep, wid)
  angle = 360 / 3
  notch(len, dep, wid, "a")
  left(angle)
  notch(len, dep, wid, "b")
  left(angle)
  notch(len, dep, wid, "c")
  left(angle)
done

define go(s, ttl)
  draw(s, "a")
  from("b","c")
    pitch(90)
    left(180)
    if (ttl > 0)
      go(s, ttl - 1)
    done
  done
done

shape("tri")
  notched_tri(3, 0.4, 0.1)
done

go("tri", 3)
```

Figure 6. Recursive FlatLang code storing a shape called 'tri', drawing it from location 'a', and drawing subsequent 'tri' shapes from locations 'b' and 'c'. A single 'tri' is shown at top left, the graphic output of this program is shown at top right.

```
; make an n-sided polygon beginning and
; ending at the current turtle position.
;
define centered_polygon(sides, radius)
  angle = 360 / sides
  points = [] ; initialize empty list
  up()
  mark("middle")
  i = 0
  repeat(sides+1)
    backto("middle")
    left(i * angle)
    forward(radius)
    points = cons(pos(), points)
    i = i+1
  done
  down()
  repeat(points.n)
    p = first(points)
    points = rest(points)
    drawto(p)
  done
  backto("middle")
done
```

Figure 7. FlatLang showing absolute and differential geometry as well as `mark` and `backto`.

Figure 7 illustrates the use of `mark`/`backto` and `pos`/`drawto`. After lifting the pen up this code marks the "middle". Next, vertices are calculated by rotating the turtle counter-clockwise and moving forward. The first point is stored twice in order to complete the tour. Next the pen is lowered, and points are connected by repeated calls to `drawto`. This code makes equilateral polygons that are exactly centered at the initial position.

5.4 Objects

FlatLang lets designers make objects with instance members and methods. There is no notion of inheritance, however. The code in Figure 8 defines a 'collar' object with two instance variables with default values and one method. Objects provide an abstraction and help us work at a higher level. Instead of explicitly using turtle operations we may use object operations.

6 Future Work

The primary (indeed, the only) interaction mode currently available in FlatCAD is by programming in FlatLang. In order to reach a wider audience, we must make the development environment easier to use. We have begun adding support for a debugger. Syntax highlighting in the text editor would also help find errors.

```
define collar()
  c = object("collar")
  c.inner = 0.14
  c.outer = 0.4
  c.draw = collar_draw
  c
done

define collar_draw()
  part("collar")
  centered_polygon(4, inner)
  centered_polygon(14, outer)
done
```

Figure 8. A collar object, a structural part used in a mechanical construction kit.

Frequently, mechanical errors are discovered only as the parts are physically assembled. For example, the strut attached to a piston wheel conflicted with the wheel's structural collar. The strut could not freely rotate. If the program had an awareness of the desired behavior of a construction (in this case, that the strut must freely rotate) it could provide critical feedback before users invested time in fabricating a physical model.

It is also possible to generate models based on functional descriptions, as in MachineShop [2]. For example, desired high level description such as "translate radial motion into harmonic linear motion" could be translated into FlatLang code in a variety of ways (such as the code in Figure 2). High level descriptions may be articulated in any number of ways, such as a traditional WIMP GUI. We are particularly interested in the possibility of recognizing freehand sketches of mechanisms and inferring behavioral intent. This intend can then be used to generate FlatLang code that in turn generates a functional assembly.

7 Conclusion

We have introduced FlatCAD, an environment for programming physical shape. FlatCAD allows us to escape the immutable nature of construction kit pieces by providing a straightforward way to design and manufacture our own. The FlatLang language offers a number of ways for working with shape, letting designers choose the best method for the job. After designing individual parts we can program partial or complete assemblies by describing how the constituent parts fit together. We can see the assembly onscreen and fabricate it using a laser cutter. The process of designing mechanisms with code can be powerful.

8 Acknowledgments

This work was funded by NSF Grant ITR-0326054.

References

[1] H. Abelson and A. diSessa. *Turtle Geometry*. MIT Press, 1981.

[2] G. Blauvelt and M. Eisenberg. Computer aided design of mechanical automata: Engineering education for children. In *ICET 2006, The IASTED International Conference on Education and Technology*, Calgary, Alberta, 2006.

[3] M. D. Gross. Formwriter: A little programming language for generating three-dimensional form algorithmically. In *CAAD Futures*, pages 577–588, 2001.

[4] F. Martin, M. Meo, and G. Doyle. Triskit: A software-generated construction toy system. In *"Let's get Physical" Workshop at the 2nd International Conference on Design Computing and Cognition (DCC06)*, Eindhoven, The Netherlands, 2006.

[5] Y. Oh, G. Johnson, M. D. Gross, and E. Do. The Designosaur and the Furniture Factory: Simple software for fast fabrication. In *Second International Conference on Design Computing and Cognition*, Eindhoven, The Netherlands, 2006.

[6] S. Papert. *Mindstorms–Children, Computers, and Powerful Ideas*. Harvester Press, Brighton, 1980.

[7] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.