FlatCAD and FlatLang: Kits by Code

Gabe Johnson
Carnegie Mellon University
Computational Design Lab (codelab)

VL/HCC - September 2008

Hello. I'm Gabe Johnson from CMU. Today I'm going to be presenting my work on FlatCAD, a 3D modeling environment based on programming in a domain specific language called FlatLang. (0:11)

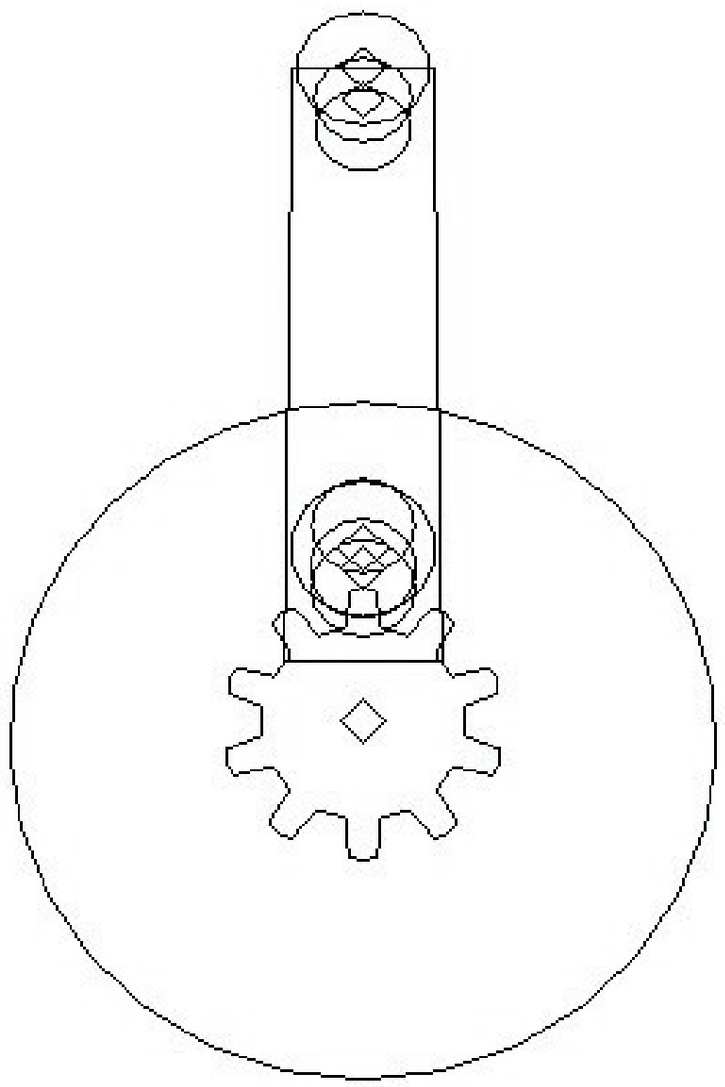
(a few) Ways of Designing

- Structured modeling tool (CAD)
- Freehand drawing (sketching)
- Give examples or analogies (human-human)
- ... many more!

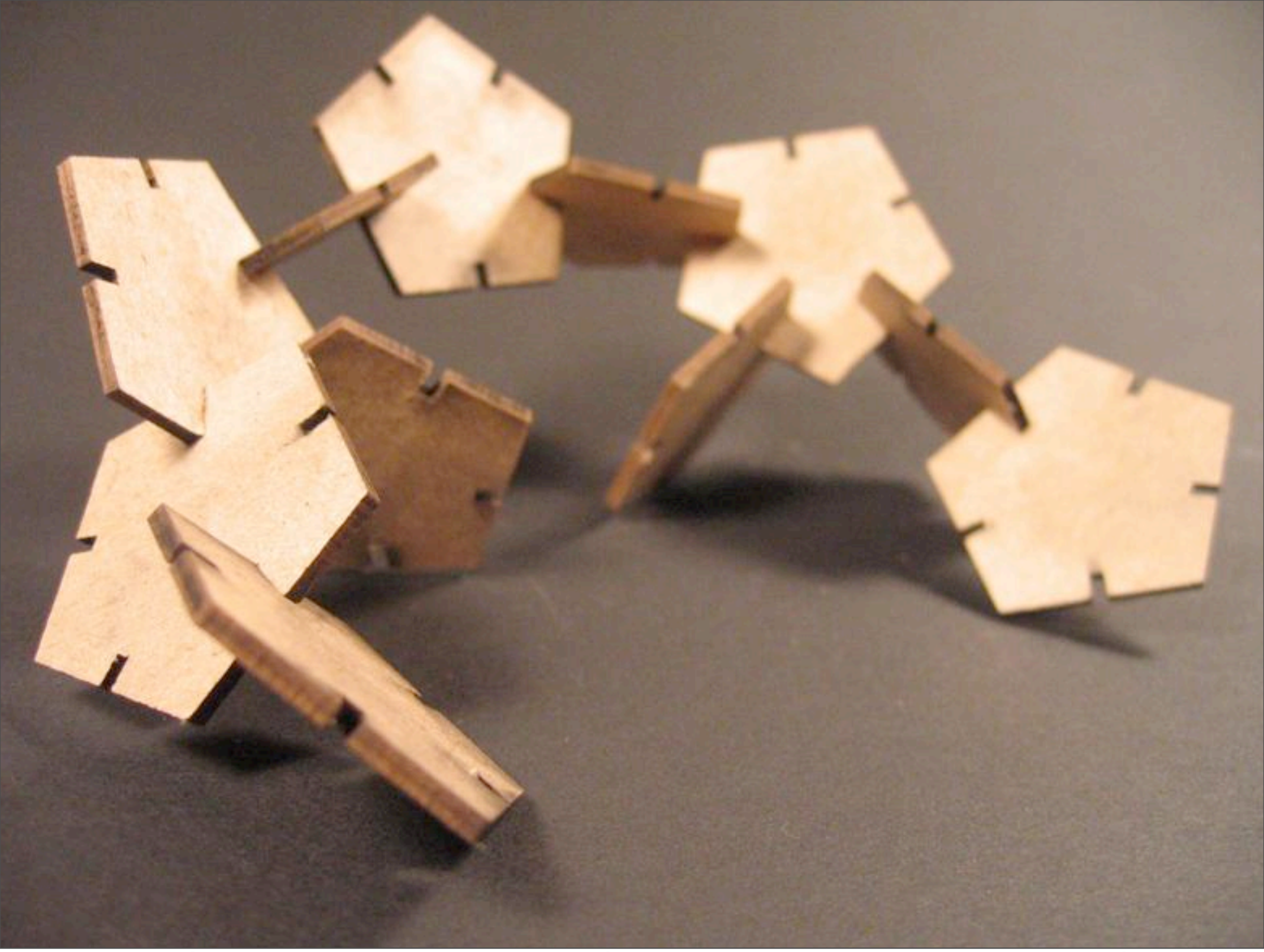
There are many ways we can design things. By designing, I mean generating and exploring ideas, and specifying how those ideas should be implemented. We could use a structured modeling tool like SketchUp. Or, we could make freehand drawings. Or if we're working with domain experts we can explain what we want to them by giving examples or analogies. Certainly there are many other ways of doing design. (0:25)

The FlatCAD Way: Code

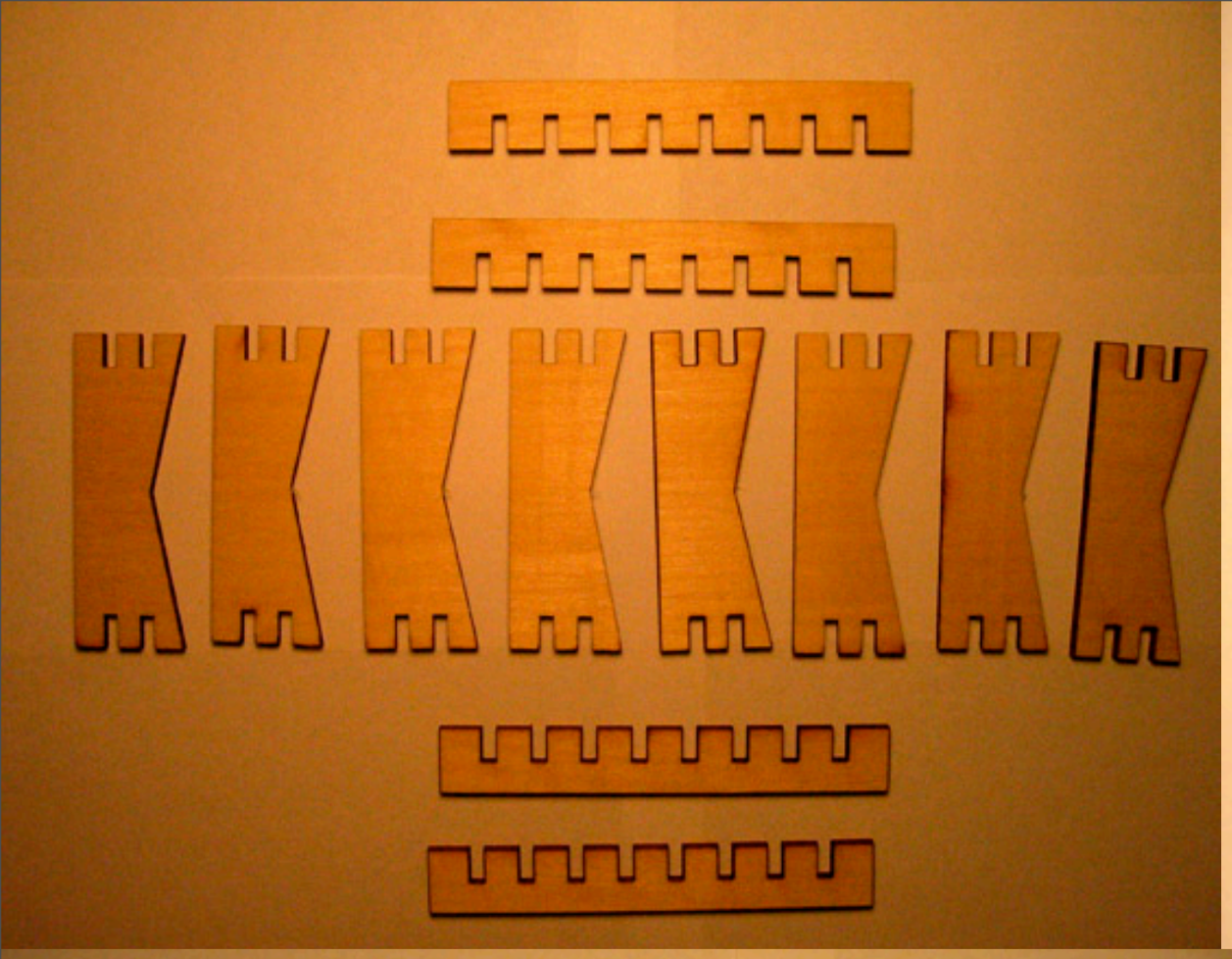
coaxial(gear(10), piston_wheel(5))

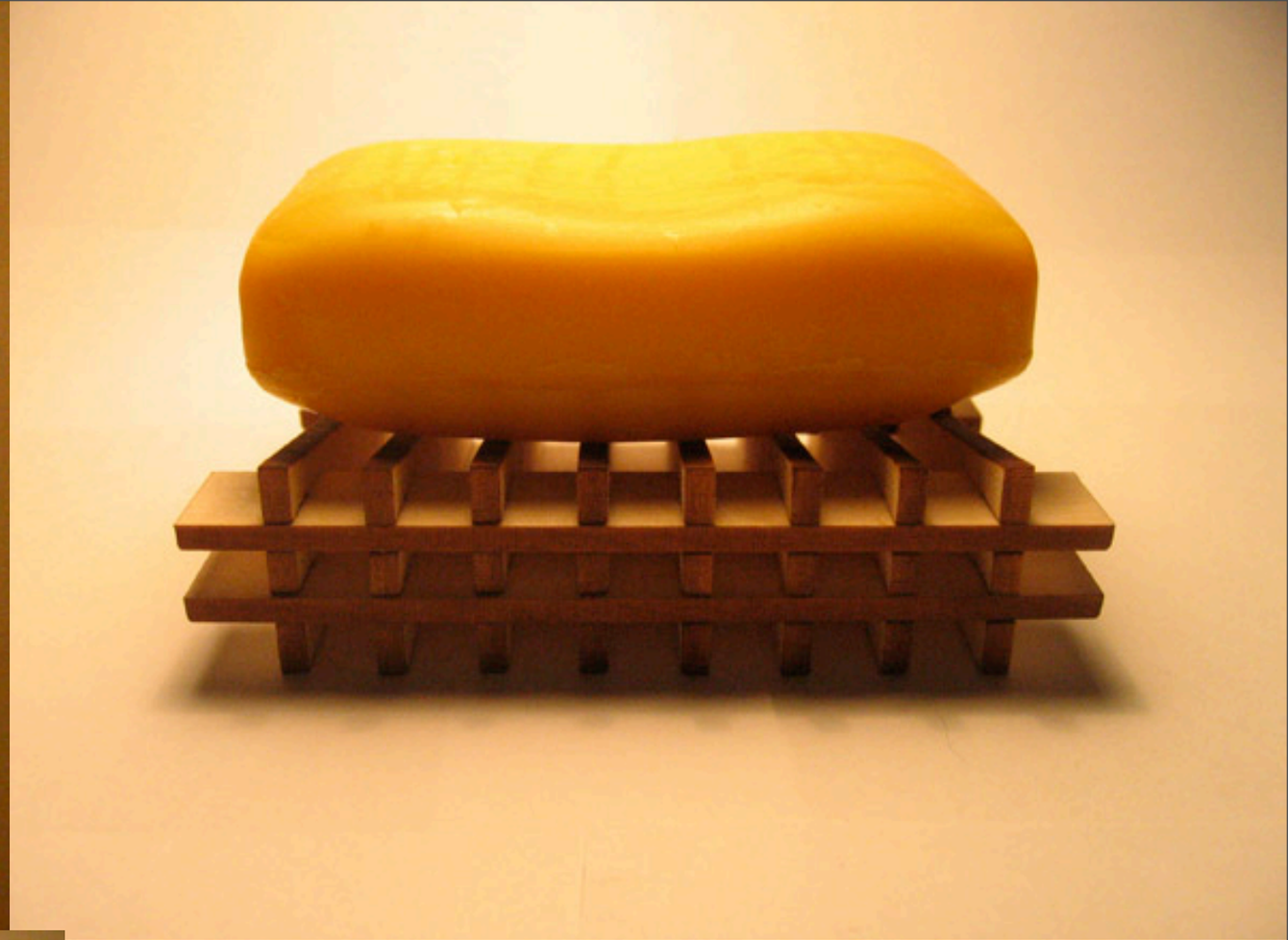


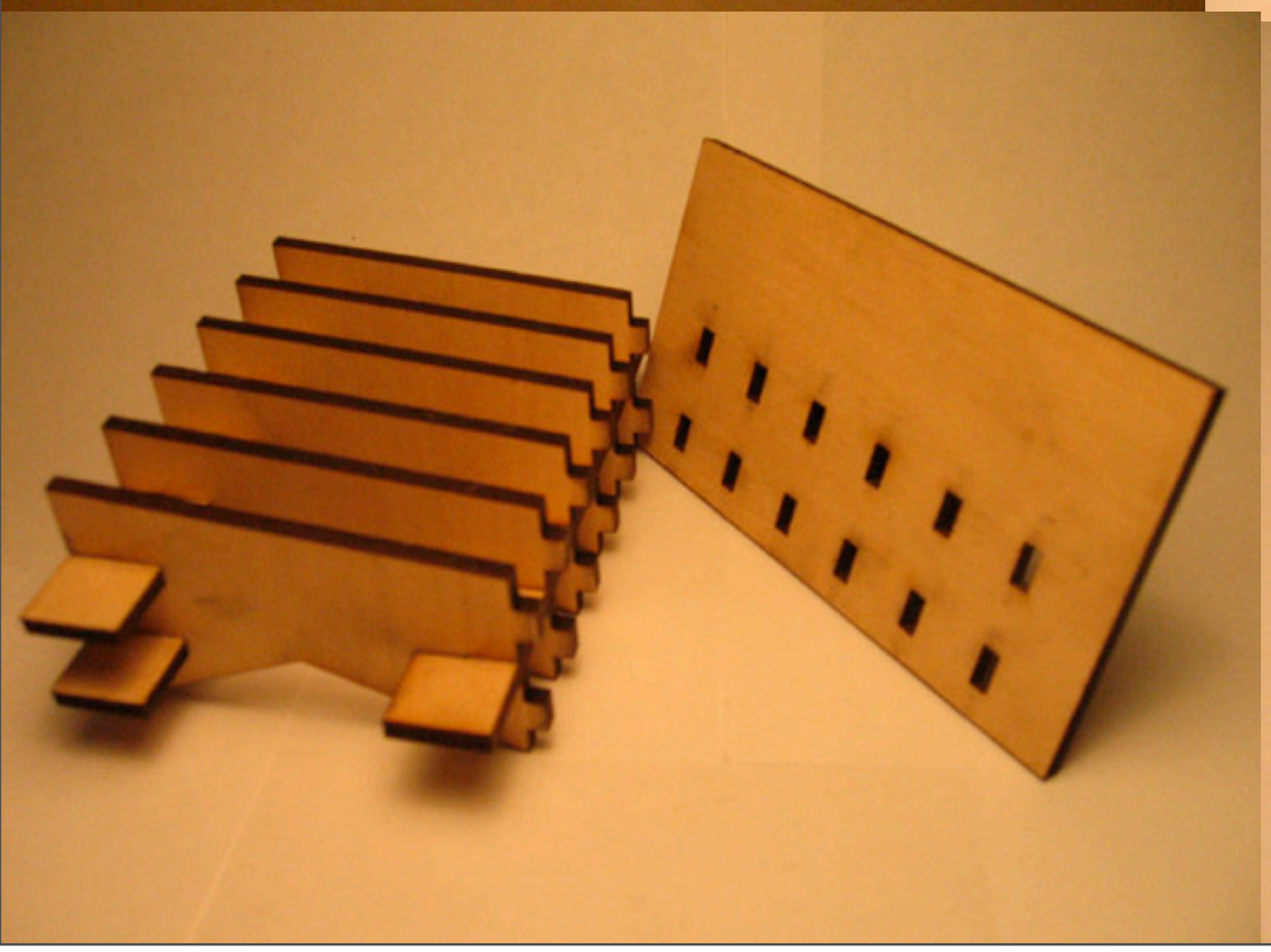
I’ve developed FlatCAD as a way of exploring programming as a way of designing. Here you can see a one-line program written in FlatLang. It creates a gear with ten teeth and a piston wheel that is five inches long. These items are arranged so they share a common axis. You can see the on-screen representation at left. Another important aspect of FlatCAD is that it supports manufacturing using laser cutters out of flat material such as wood or cardboard. (0:29)

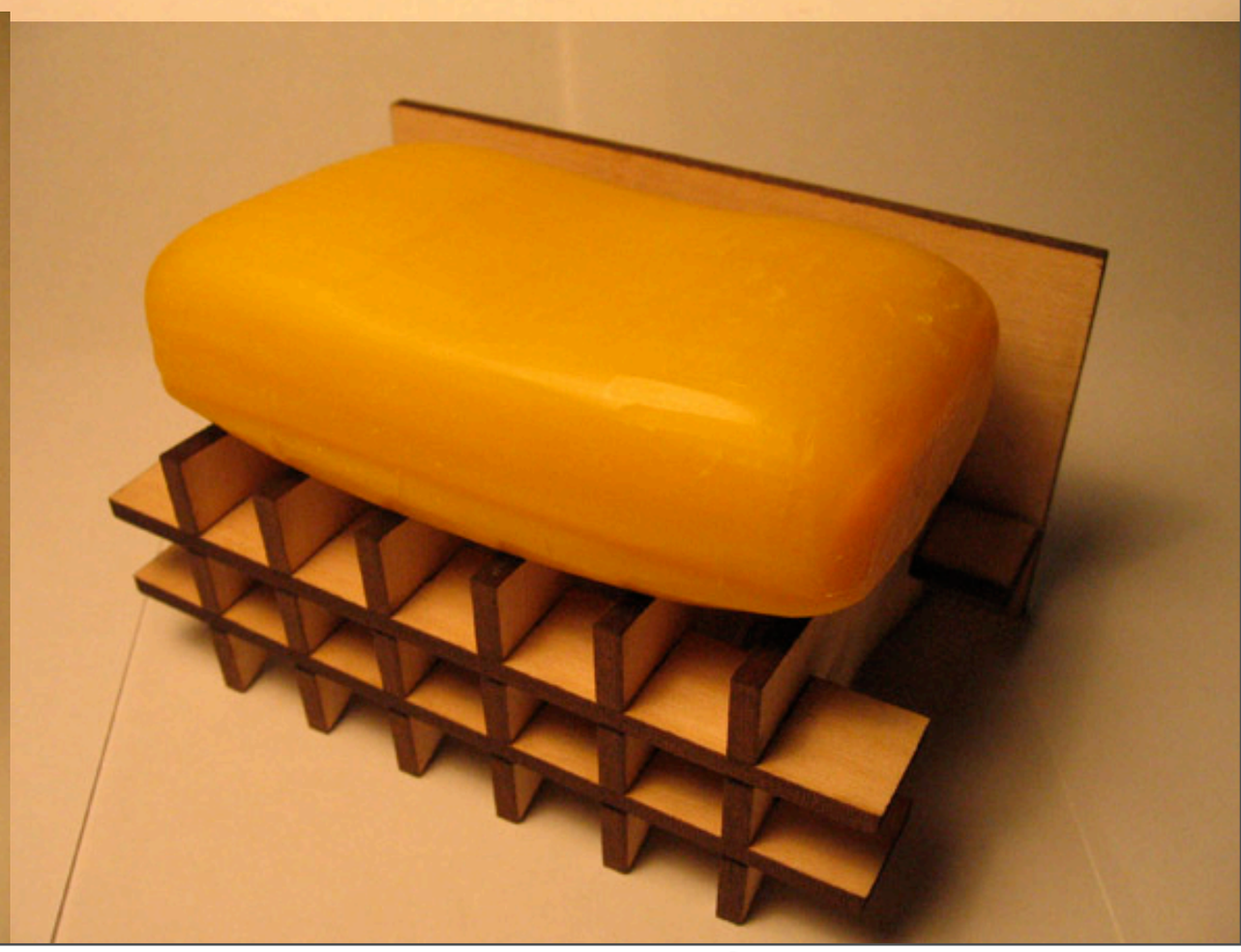


This and the following slides are examples of FlatCAD’s output. This is an example from a simple construction kit designed in FlatCAD. (0:07)







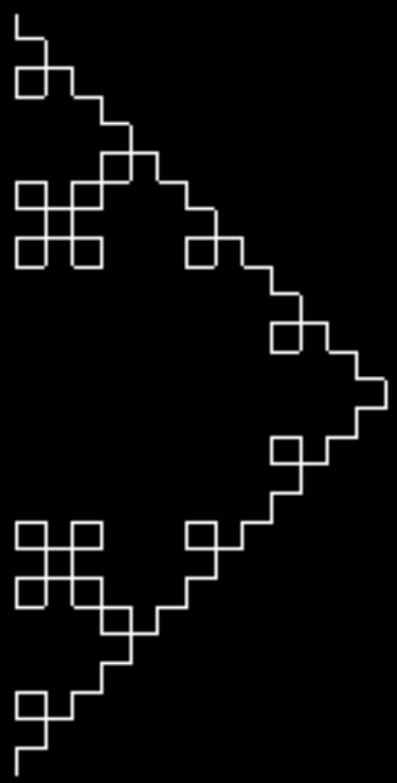


FlatCAD has also been used to make functional items as well. This is one of the soapdishes made with FlatCAD. (0:06)

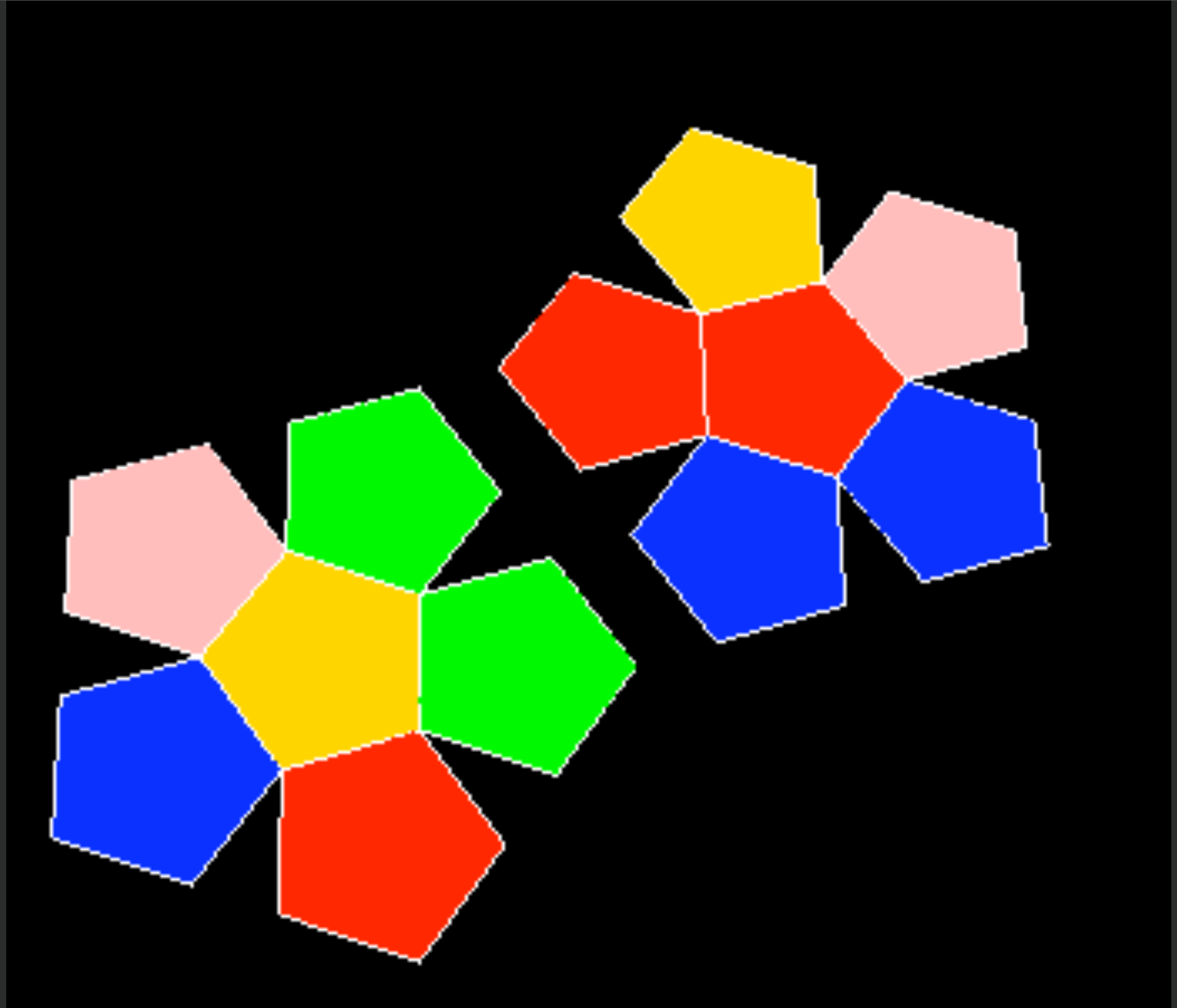
Along the same line as the soapdish, this is a toothbrush holder, which actually works quite nicely. (0:06)

FlatCAD

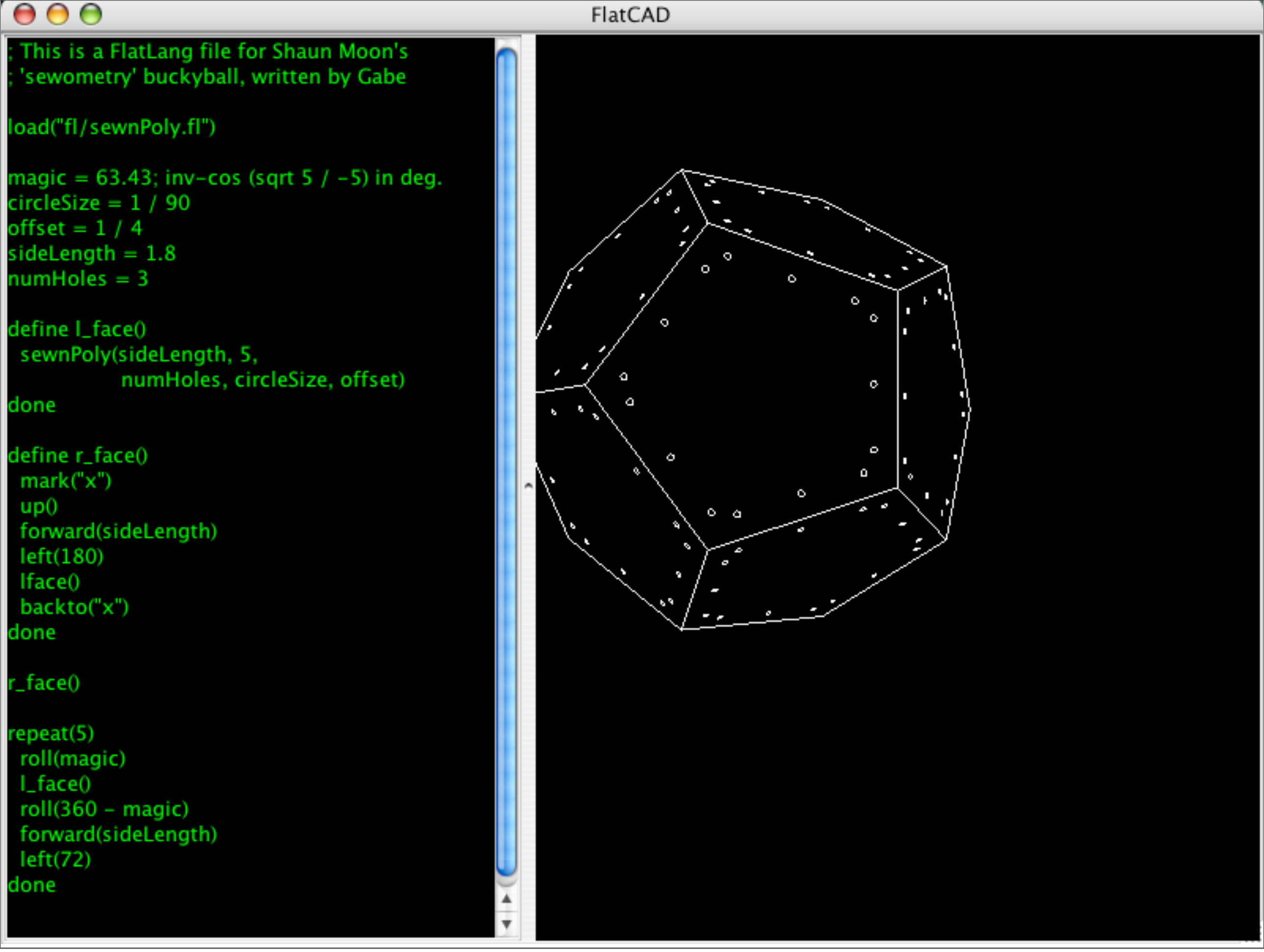
```
define F(x)
  mark ("start")
  forward (x)
  mark ("end")
done

define expand_F(x)
  F(x)
  right(90)
  F(x)
  left(90)
  F(x)
  left(90)
  F(x)
  right(90)
  F(x)
done

define expand(x)
  replaceAll("start", "end")
    expand_F(x)
  done
done

F(1)

amt = 0.5
expand(amt)
amt = amt / 2
expand(amt)
amt = amt / 2
expand(amt)
```



While physical output is certainly a hallmark of FlatCAD, it can also be used to make interesting screen drawings in the tradition of LOGO's turtle graphics. This is an example of using FlatLang's code rewriting feature to create a Lindenmayer system. (0:15)



FlatCAD can also be used to render animations. This video shows a pattern of pentagons rolling up to form a dodecahedron. (add more here to fill up the time) (0:20)



Here's a screenshot of the FlatLang code and visual output for another construction kit toy called Sewometry. This idea was from Shaun Moon, a colleague of mine. The pieces are intended to be sewn together using shoelaces or similar using tiny holes along each polygon's edge. (0:17)

Here's a photo of Sewometry once it has been printed and partially assembled. Shaun's daughter loved to play with Sewometry. (0:08)

FlatCAD: A programming-based 3D modeling tool.

FlatLang: FlatCAD's language.

Syntax: Python-like Semantics: LOGO-like

FlatCAD is the modeling environment, which handles drawing models to the screen, for generating laser cutter output files, and giving the user a simple development environment. It also analyses the programmer's model and provides feedback, for example identifying closed polygons and filling them in with colors.

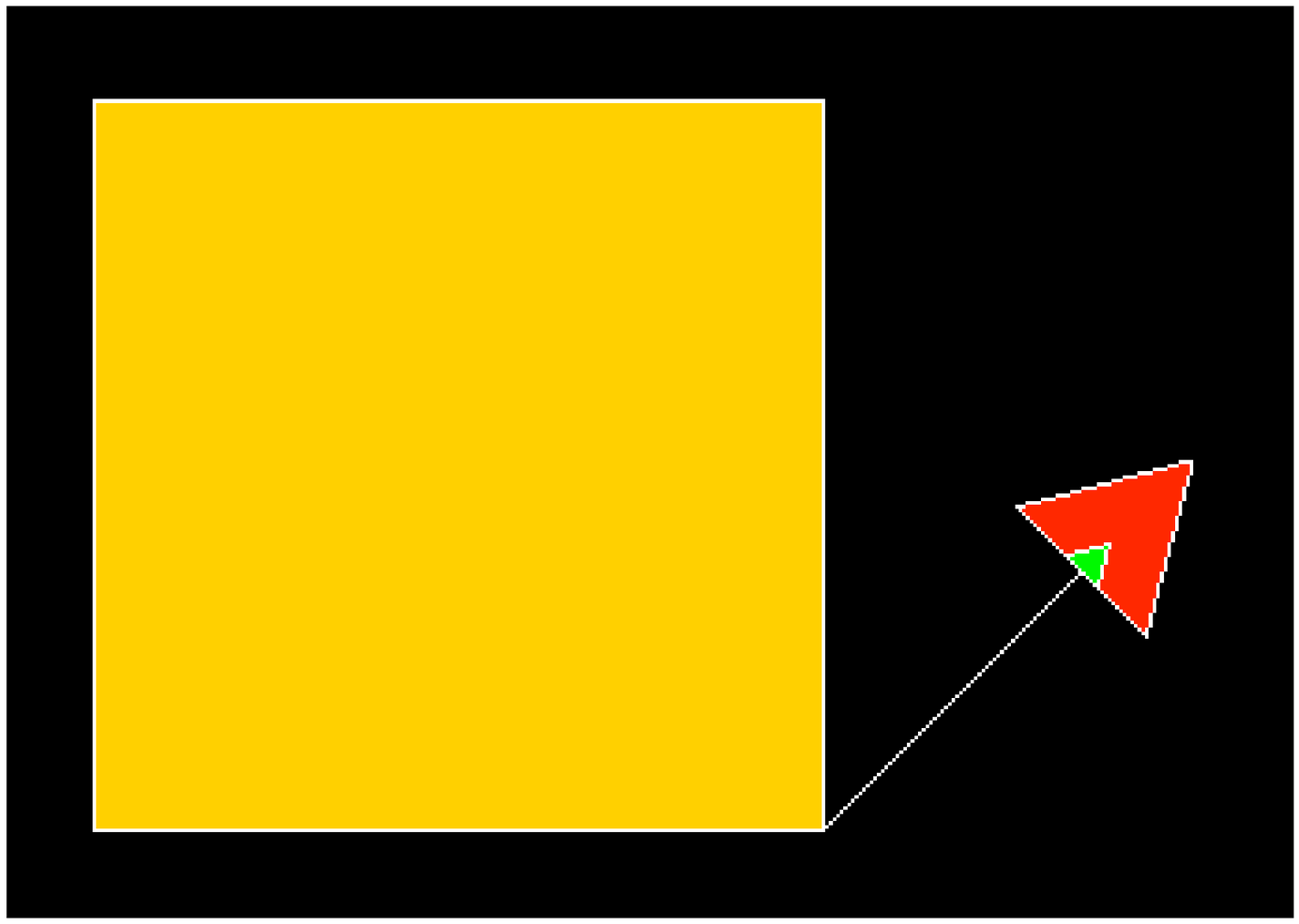
FlatLang is the domain-specific programming language I developed for use in FlatCAD. Its syntax looks something like Python, but its semantics are similar to the turtle graphics found in LOGO. (0:30)

FlatCAD: A programming-based 3D modeling tool.

FlatLang: FlatCAD's language.

Syntax: Python-like Semantics: LOGO-like

```
repeat(4)
  forward(2)
  left(90)
done

right(45)
forward(1)
```



FlatCAD is the modeling environment, which handles drawing models to the screen, for generating laser cutter output files, and giving the user a simple development environment. It also analyses the programmer's model and provides feedback, for example identifying closed polygons and filling them in with colors.

FlatLang is the domain-specific programming language I developed for use in FlatCAD. Its syntax looks something like Python, but its semantics are similar to the turtle graphics found in LOGO. (0:30)

Earlier Related Languages

- LOGO (1967): Differential ("turtle") geometry.
- FormWriter (2001): "Flying Turtle" for making 3D pictures.
- Triskit (2006): Turtle path drives laser cutter for physical production.

I've mentioned LOGO several times so far. As I'm sure many in this room are familiar with, it was originally developed by Wally Feurzeig and Seymour Papert at Bolt, Beranek and Newman in 1967. One of the things that popularized LOGO was the on-screen "turtle" graphics, whereby programmers would instruct a triangle-shaped turtle to move around on the screen relative to where it currently is.

There are many other uses of turtle geometry, but I'll mention two that influenced the design of FlatLang. FormWriter was a LOGO-like language that featured a "flying turtle" that let users arrange objects such as cubes or cylinders in 3D space. It was made by my advisor Mark Gross. Triskit, done by Fred Martin and students, used turtle geometry to generate laser cutter output files for building a construction kit. (0:51)

Earlier Related Languages

- LOGO (1967): Differential ("turtle") geometry.
- FormWriter (2001): "Flying Turtle" for making 3D pictures.
- Triskit (2006): Turtle path drives laser cutter for physical production.

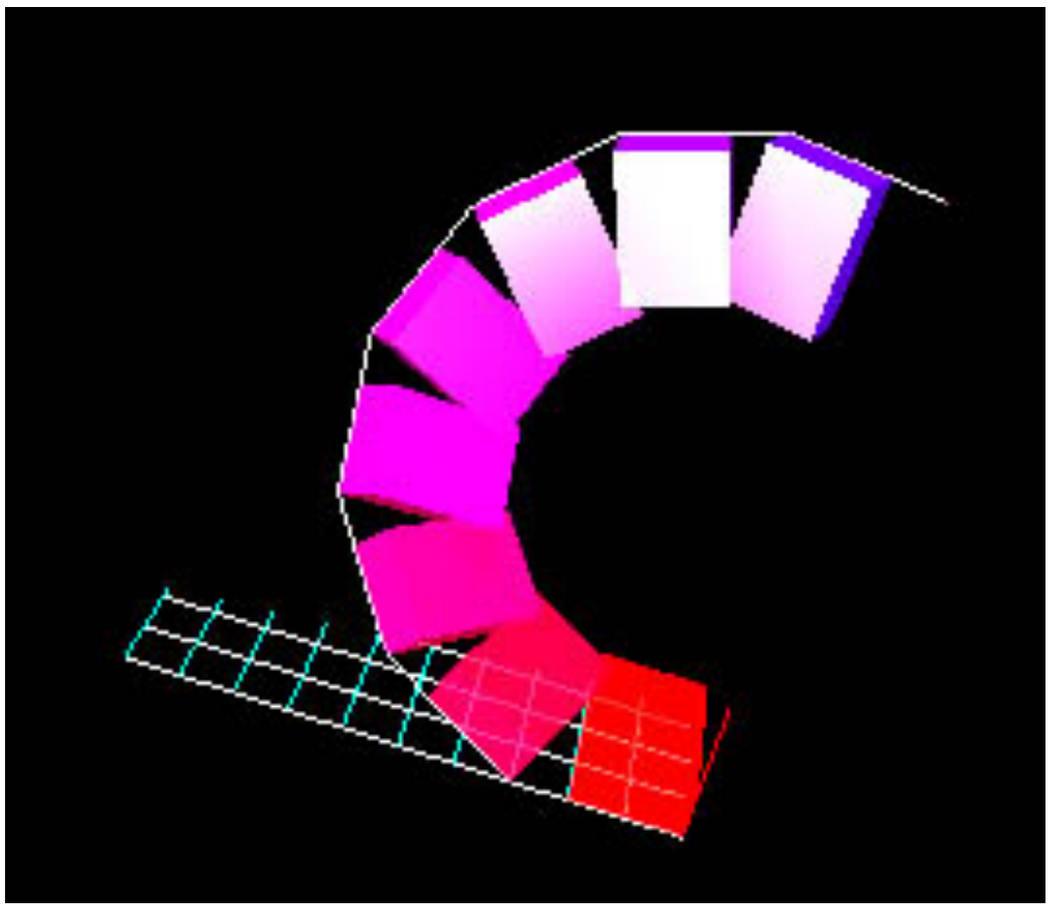
```
repeat 3 [forward 50 right 60]
```

I've mentioned LOGO several times so far. As I'm sure many in this room are familiar with, it was originally developed by Wally Feurzeig and Seymour Papert at Bolt, Beranek and Newman in 1967. One of the things that popularized LOGO was the on-screen "turtle" graphics, whereby programmers would instruct a triangle-shaped turtle to move around on the screen relative to where it currently is.

There are many other uses of turtle geometry, but I'll mention two that influenced the design of FlatLang. FormWriter was a LOGO-like language that featured a "flying turtle" that let users arrange objects such as cubes or cylinders in 3D space. It was made by my advisor Mark Gross. Triskit, done by Fred Martin and students, used turtle geometry to generate laser cutter output files for building a construction kit. (0:51)

Earlier Related Languages

- LOGO (1967): Differential ("turtle") geometry.
- FormWriter (2001): "Flying Turtle" for making 3D pictures.
- Triskit (2006): Turtle path drives laser cutter for physical production.



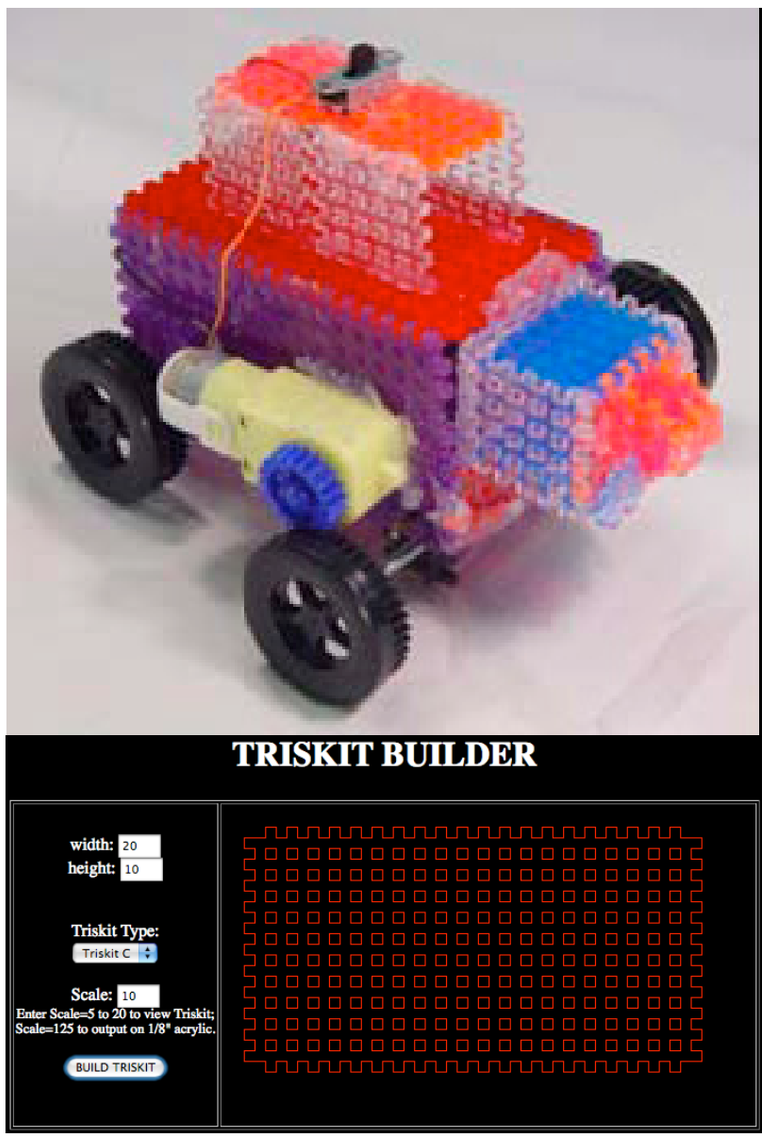
```
to box_move ()
 box (3,2,1)
 forward (3)
  right (20)
  pitch (20)
end
```

I've mentioned LOGO several times so far. As I'm sure many in this room are familiar with, it was originally developed by Wally Feurzeig and Seymour Papert at Bolt, Beranek and Newman in 1967. One of the things that popularized LOGO was the on-screen "turtle" graphics, whereby programmers would instruct a triangle-shaped turtle to move around on the screen relative to where it currently is.

There are many other uses of turtle geometry, but I'll mention two that influenced the design of FlatLang. FormWriter was a LOGO-like language that featured a "flying turtle" that let users arrange objects such as cubes or cylinders in 3D space. It was made by my advisor Mark Gross. Triskit, done by Fred Martin and students, used turtle geometry to generate laser cutter output files for building a construction kit. (0:51)

Earlier Related Languages

- LOGO (1967): Differential ("turtle") geometry.
- FormWriter (2001): "Flying Turtle" for making 3D pictures.
- Triskit (2006): Turtle path drives laser cutter for physical production.



I've mentioned LOGO several times so far. As I'm sure many in this room are familiar with, it was originally developed by Wally Feurzeig and Seymour Papert at Bolt, Beranek and Newman in 1967. One of the things that popularized LOGO was the on-screen "turtle" graphics, whereby programmers would instruct a triangle-shaped turtle to move around on the screen relative to where it currently is.

There are many other uses of turtle geometry, but I'll mention two that influenced the design of FlatLang. FormWriter was a LOGO-like language that featured a "flying turtle" that let users arrange objects such as cubes or cylinders in 3D space. It was made by my advisor Mark Gross. Triskit, done by Fred Martin and students, used turtle geometry to generate laser cutter output files for building a construction kit. (0:51)

FlatLang Features

- 3D turtle movement: left/right, roll, pitch
- Absolute or relative turtle instructions
  - ➡ “move by” vs. “move to”
  - ➡ “turn 90°” vs. “face north” vs. “face point X”
- Support for manufacturing
- “Turtle Trees” for making coding easier

Like FormWriter, FlatLang supports arbitrary 3D turtle navigation. Unlike FormWriter, FlatLang allows the programmer to define new polyhedra composed from individual edges. To make this easier, FlatLang supports both differential geometry as well as absolute geometry. For example, we can say “move by 10 units”, which is differential, or we may say “move to location X”, which is absolute. The same is true for changing direction. We can turn by 90 degrees or we can face a cardinal direction, or we can face an existing location.

To produce physical output, FlatLang has keywords for instructing FlatCAD to begin recording geometry for logical parts. This allows us to see an artifact onscreen as it will look when assembled, but generate laser cutter files for individual components of that assembly.

FlatLang also features “Turtle Trees”, which are special data structures that make programming 3D shapes easier. The rest of this talk focuses on this particular feature. There are details of other features in the paper. (1:04)

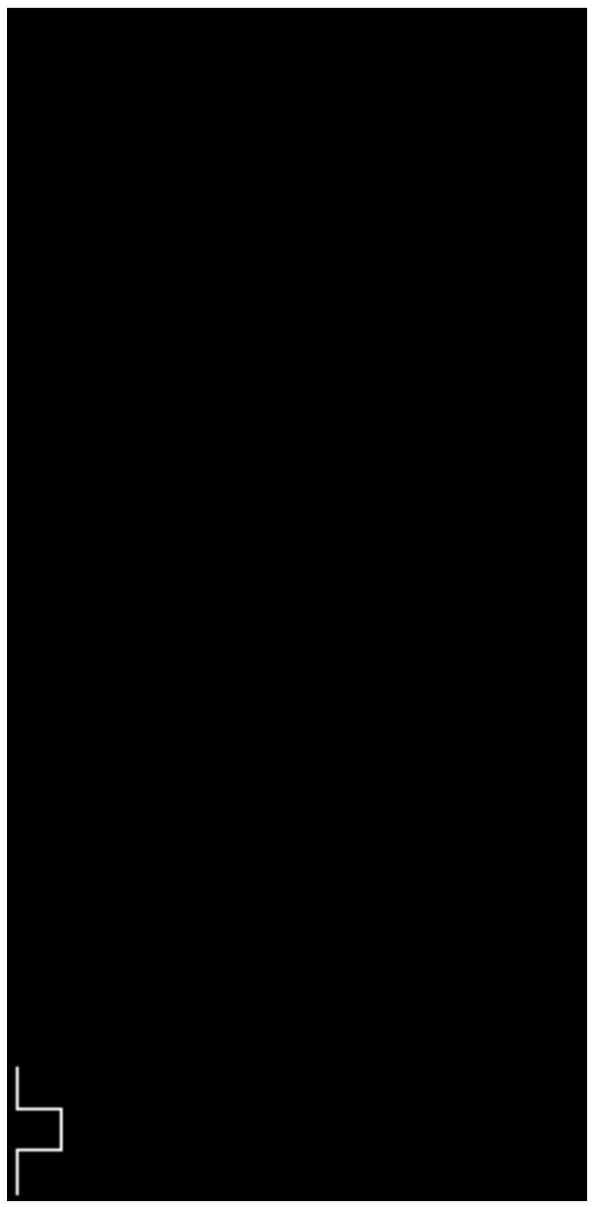
Turtle Trees

- A record of where the turtle has been and how it got there
- Record turtle operations for later playback
- Can return to locations easily
- Can manipulate turtle trees
  - ★ e.g. shape replacement

A turtle tree is a record of the operations that has taken the turtle from the starting position to its current location. This record can be used to create shapes and “play back” those shapes. It can also be used as a data structure that provides context for additional turtle instructions. Last, we can programmatically manipulate existing turtle trees. This is useful for performing shape replacement. This is how the L-system shown earlier was created. (0:30)

Turtle Trees

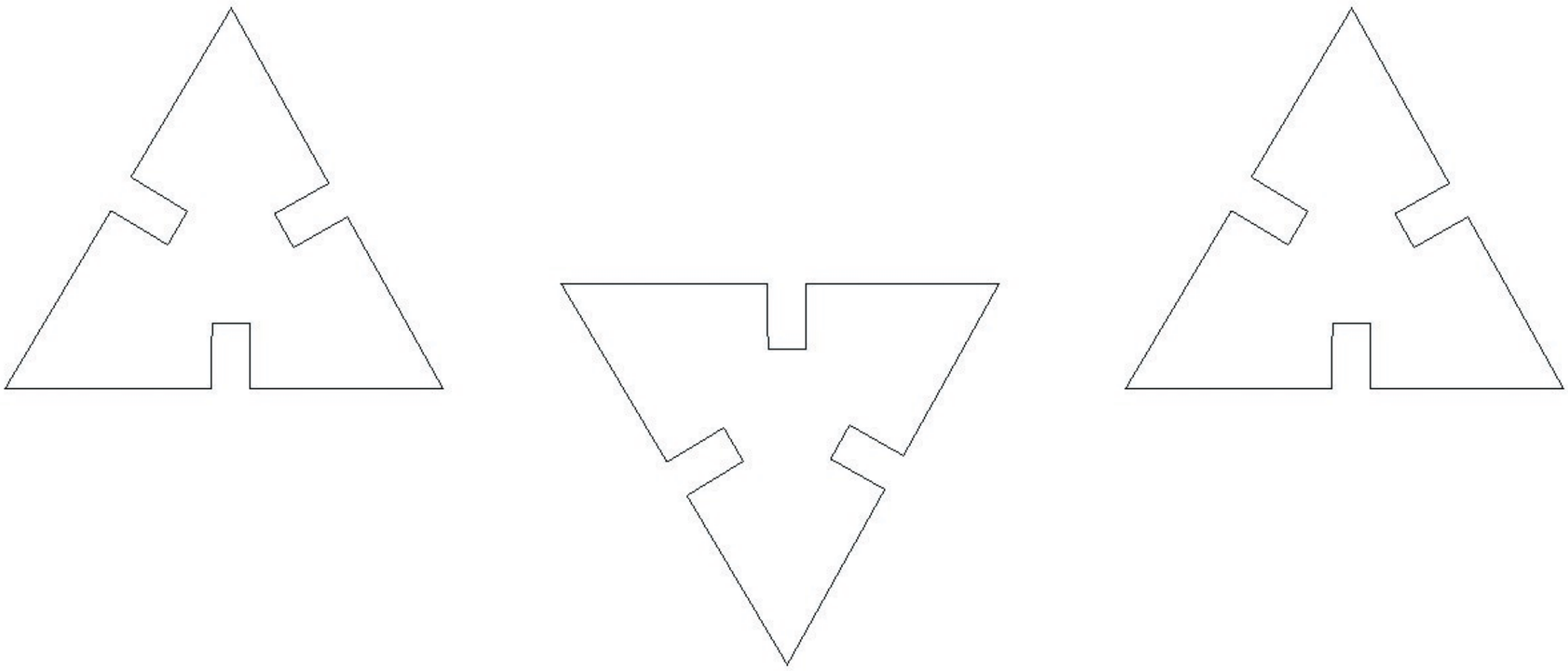
- A record of where the turtle has been and how it got there
- Record turtle operations for later playback
- Can return to locations easily
- Can manipulate turtle trees
  - ★ e.g. shape replacement



L-system; n=1..3

A turtle tree is a record of the operations that has taken the turtle from the starting position to its current location. This record can be used to create shapes and “play back” those shapes. It can also be used as a data structure that provides context for additional turtle instructions. Last, we can programmatically manipulate existing turtle trees. This is useful for performing shape replacement. This is how the L-system shown earlier was created. (0:30)

Turtle Tree Example: Triangle Construction Kit



Now I'm going to give an in-depth example of how Turtle Trees work. In this example I will be building a construction kit made out of triangular pieces that join together at these notches. (0:11)

Turtle Tree Example

We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

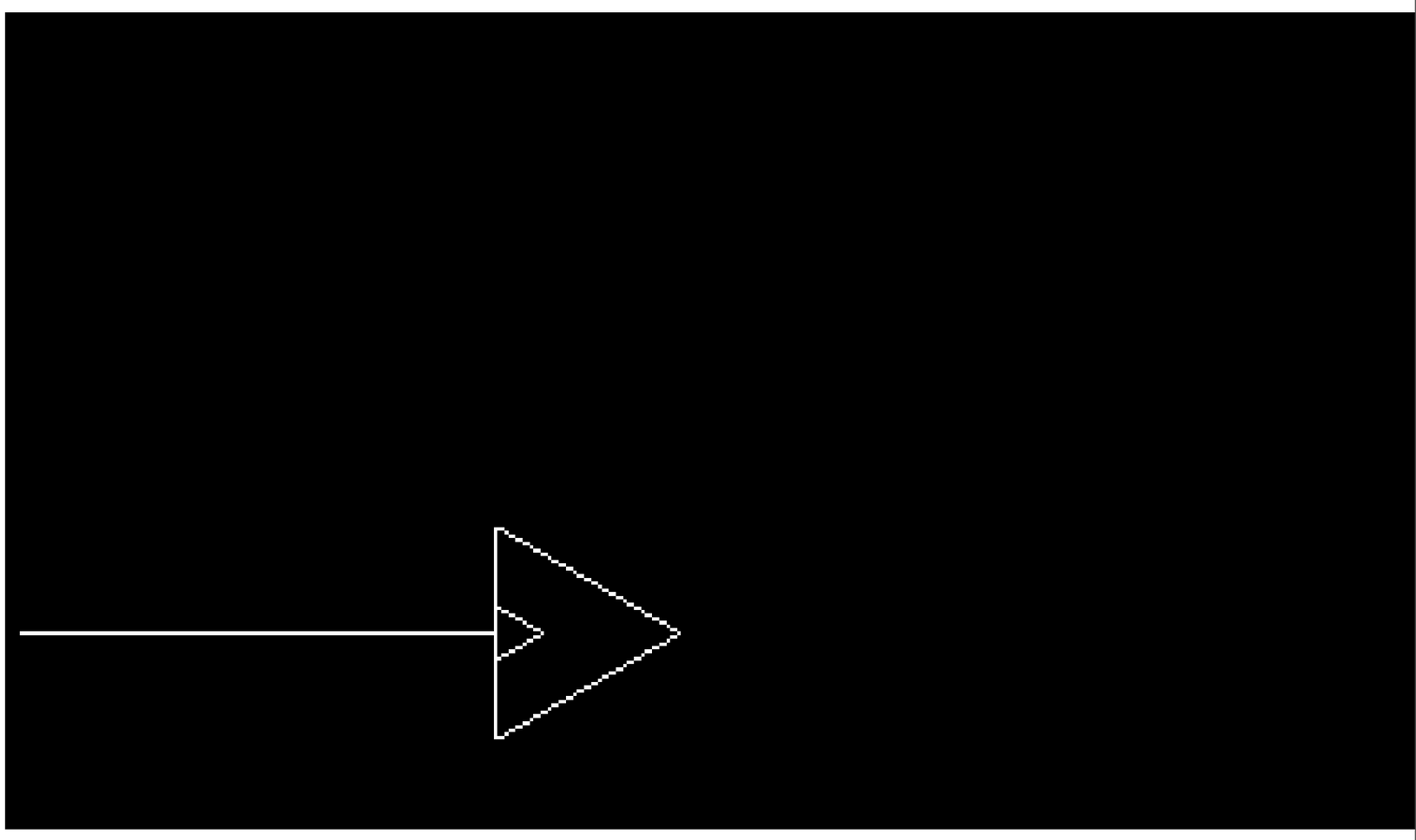
Turtle Tree Example

We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

forward(side_length / 2)

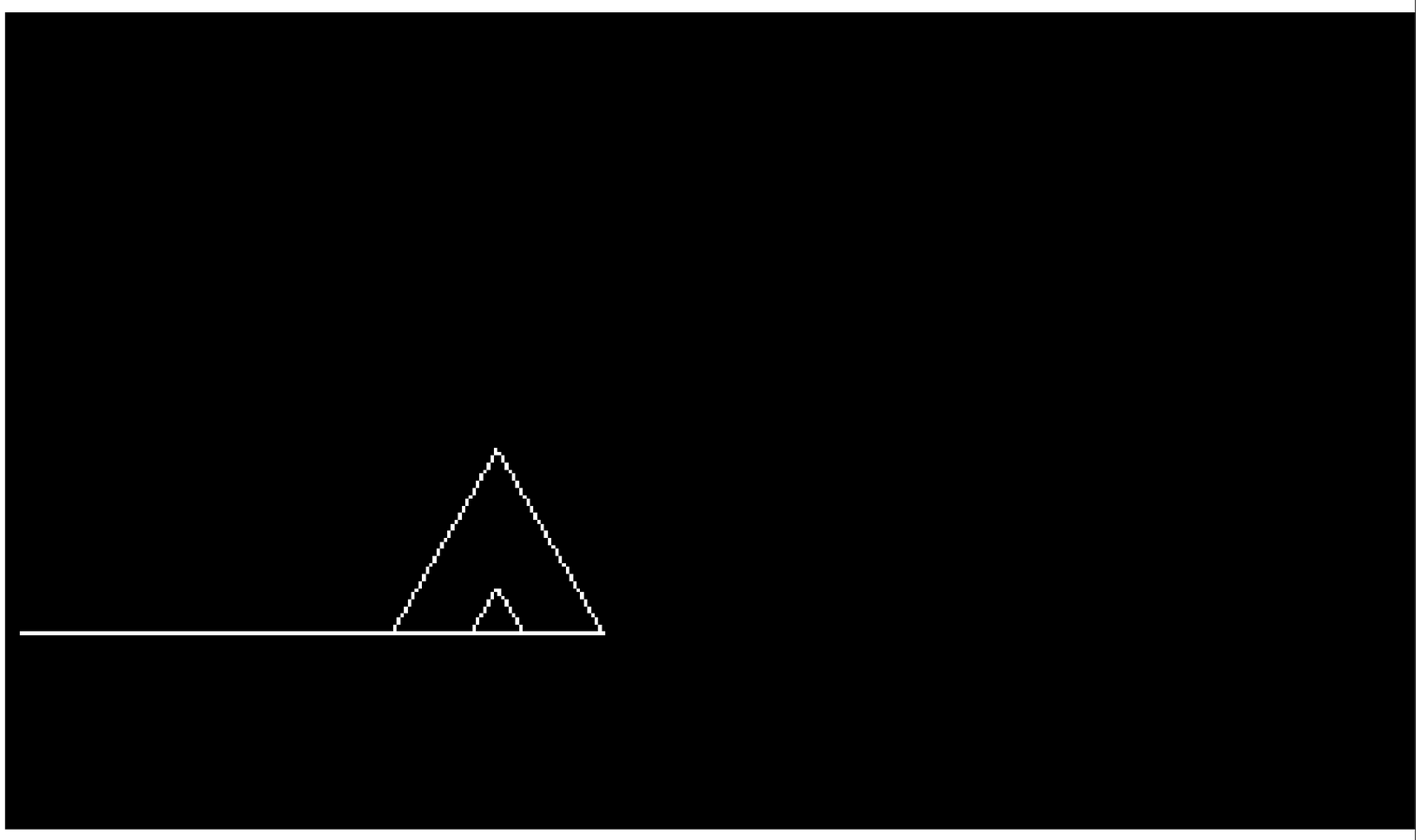


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

forward(side_length / 2)
left(90)

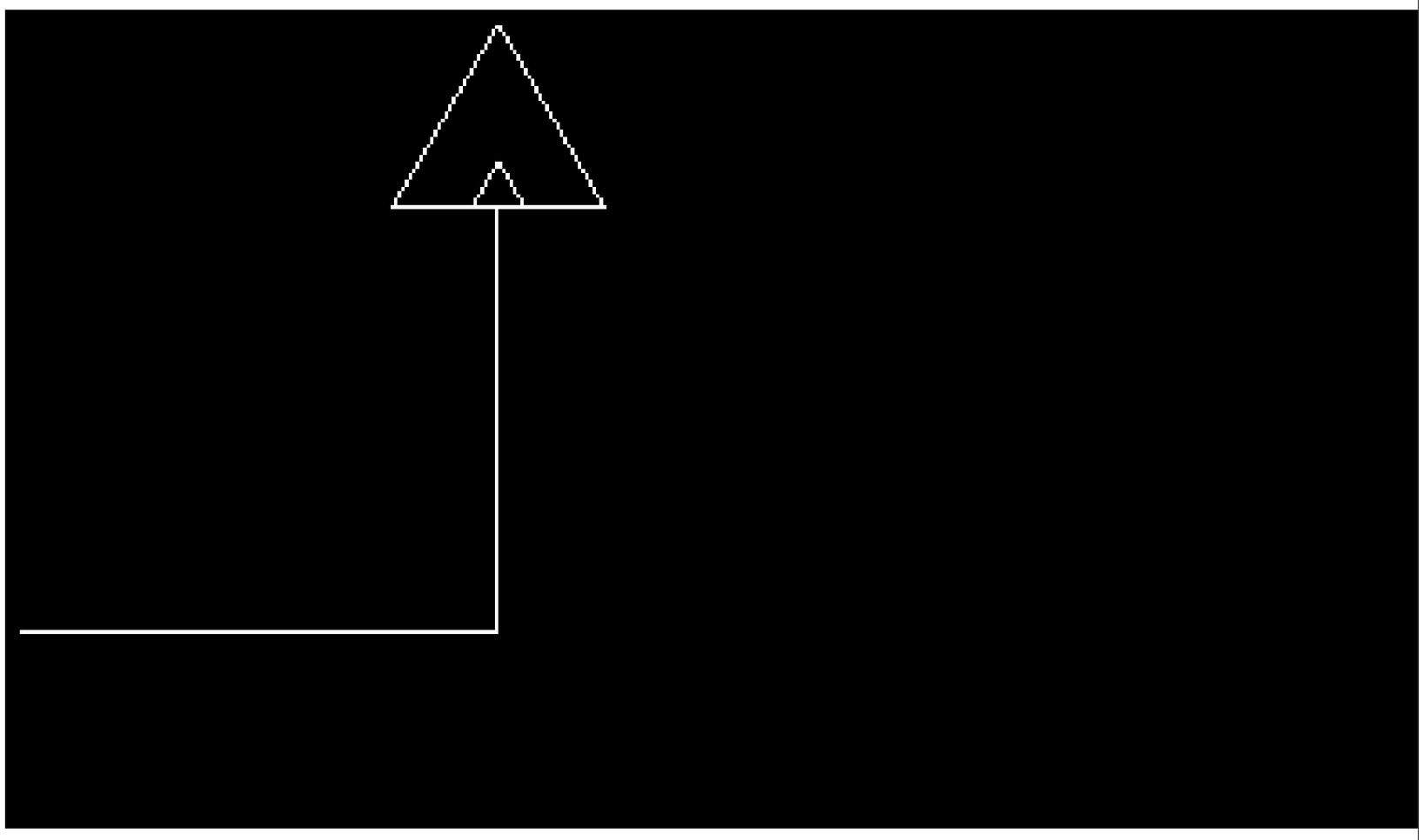


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
```

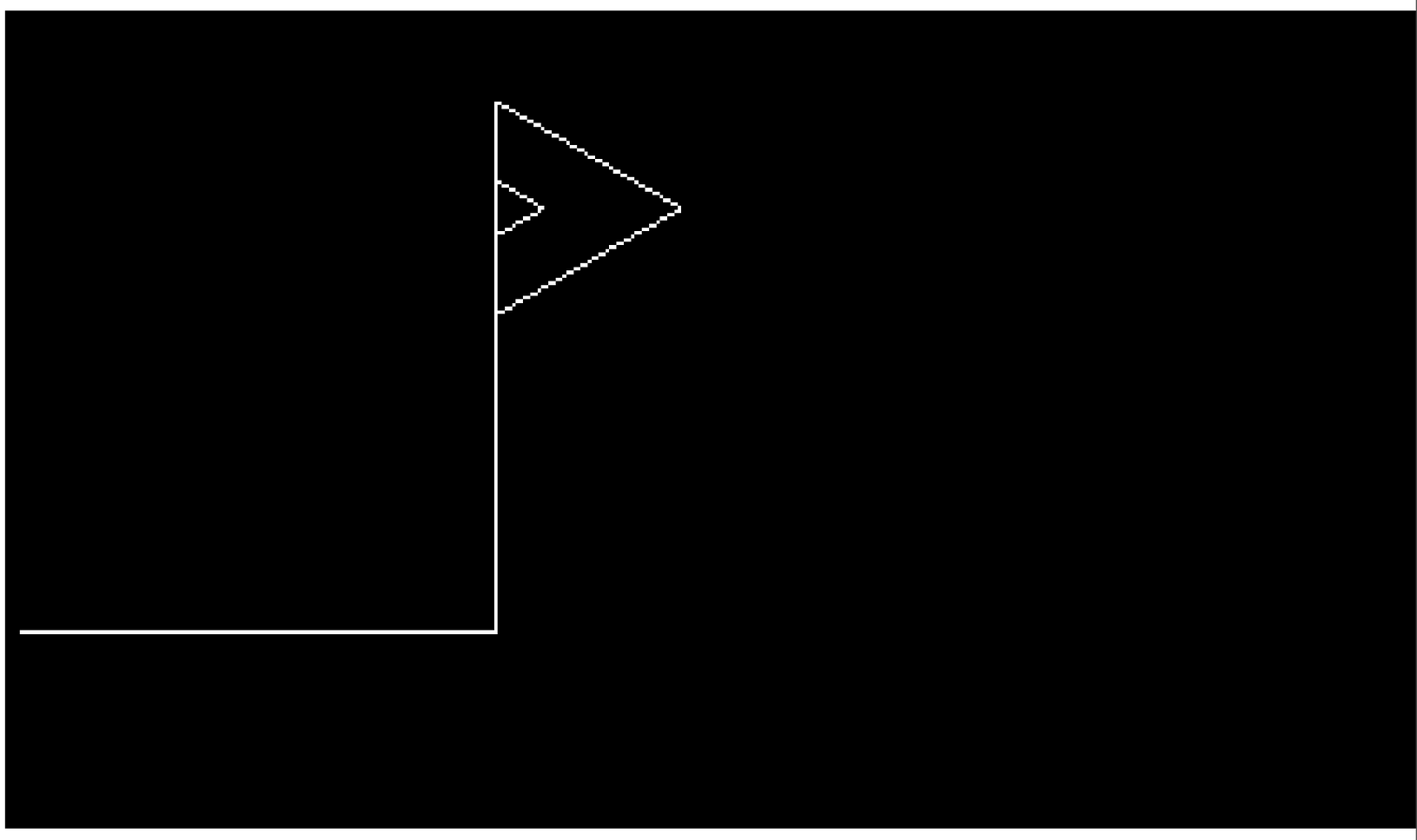


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
```

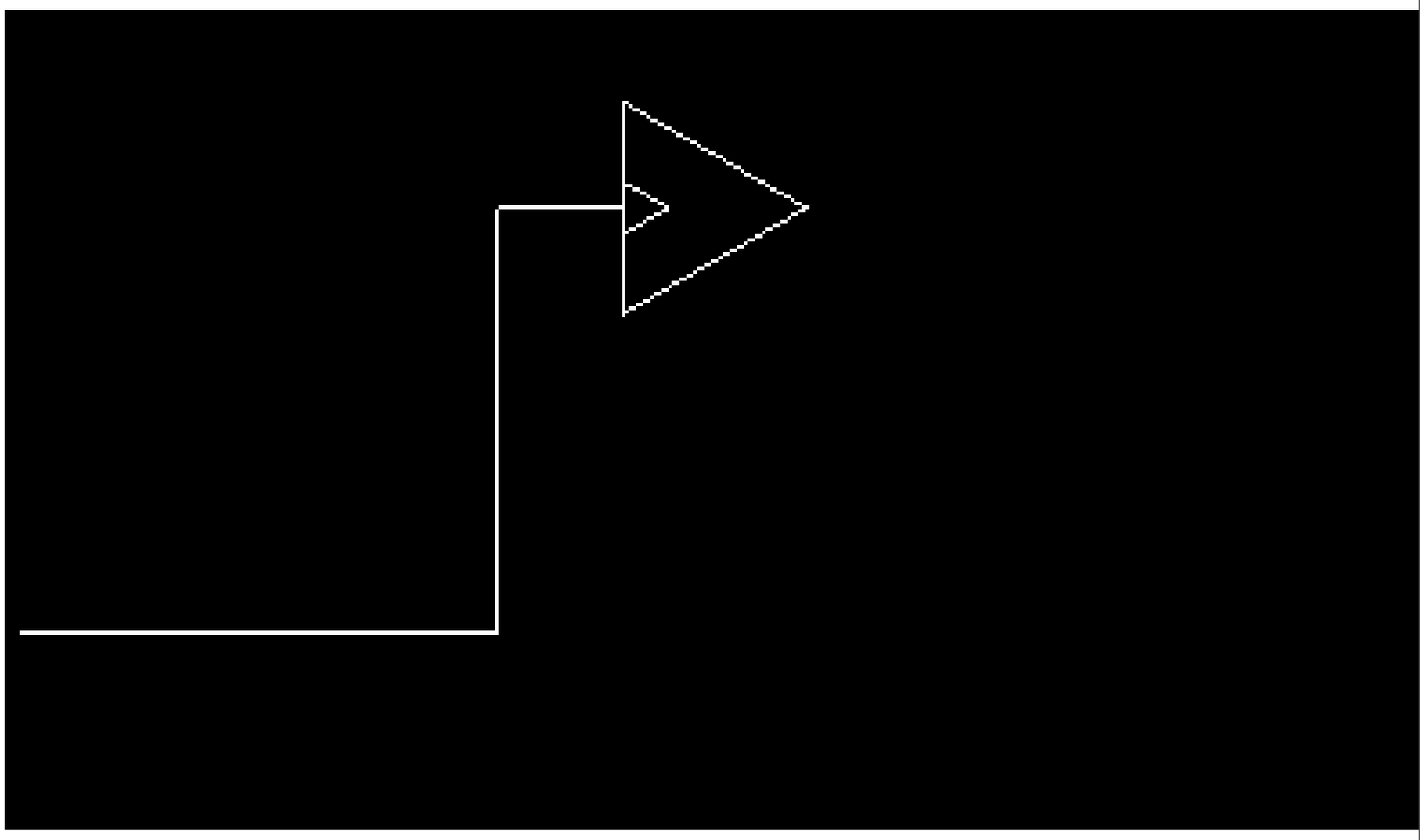


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
```

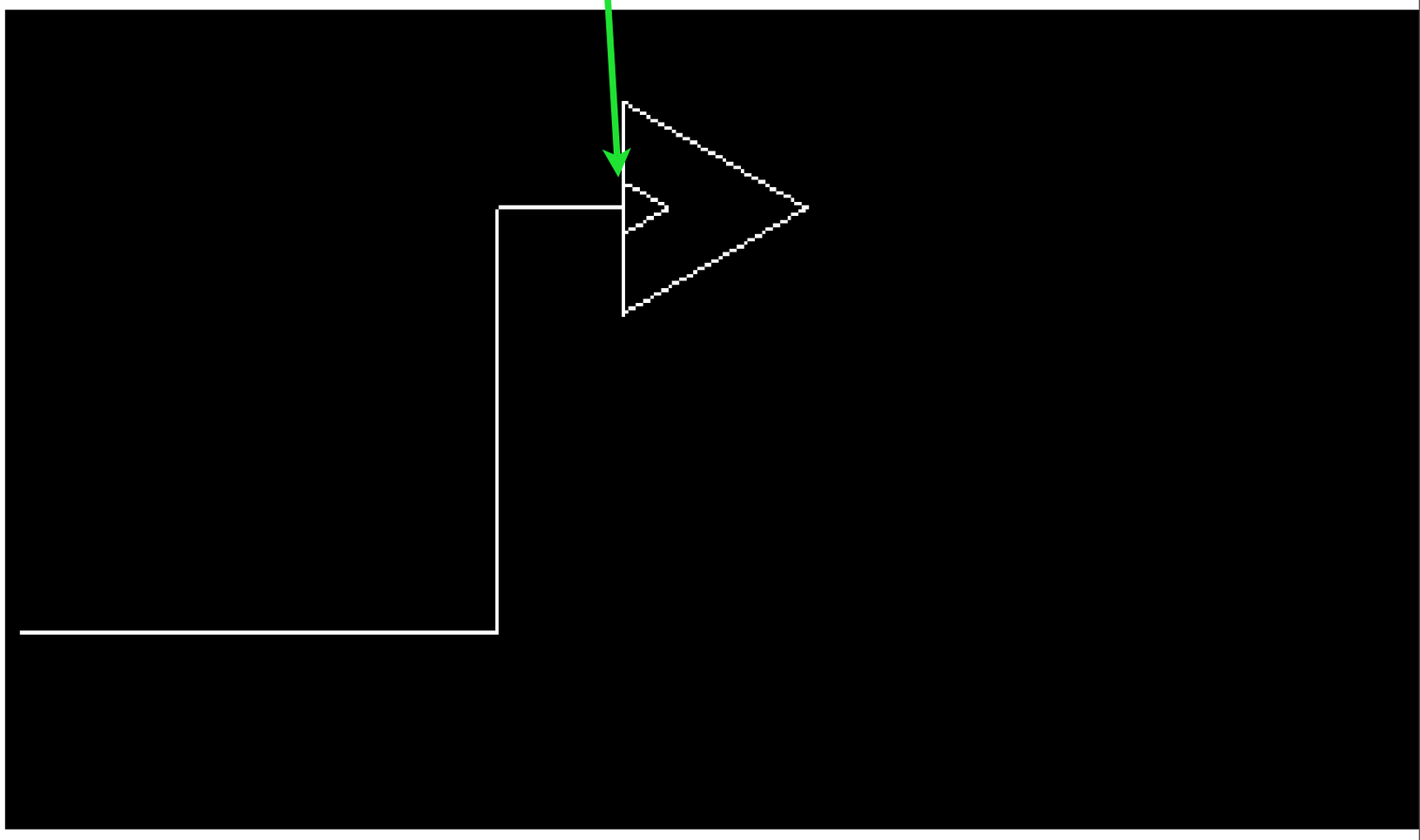


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
```

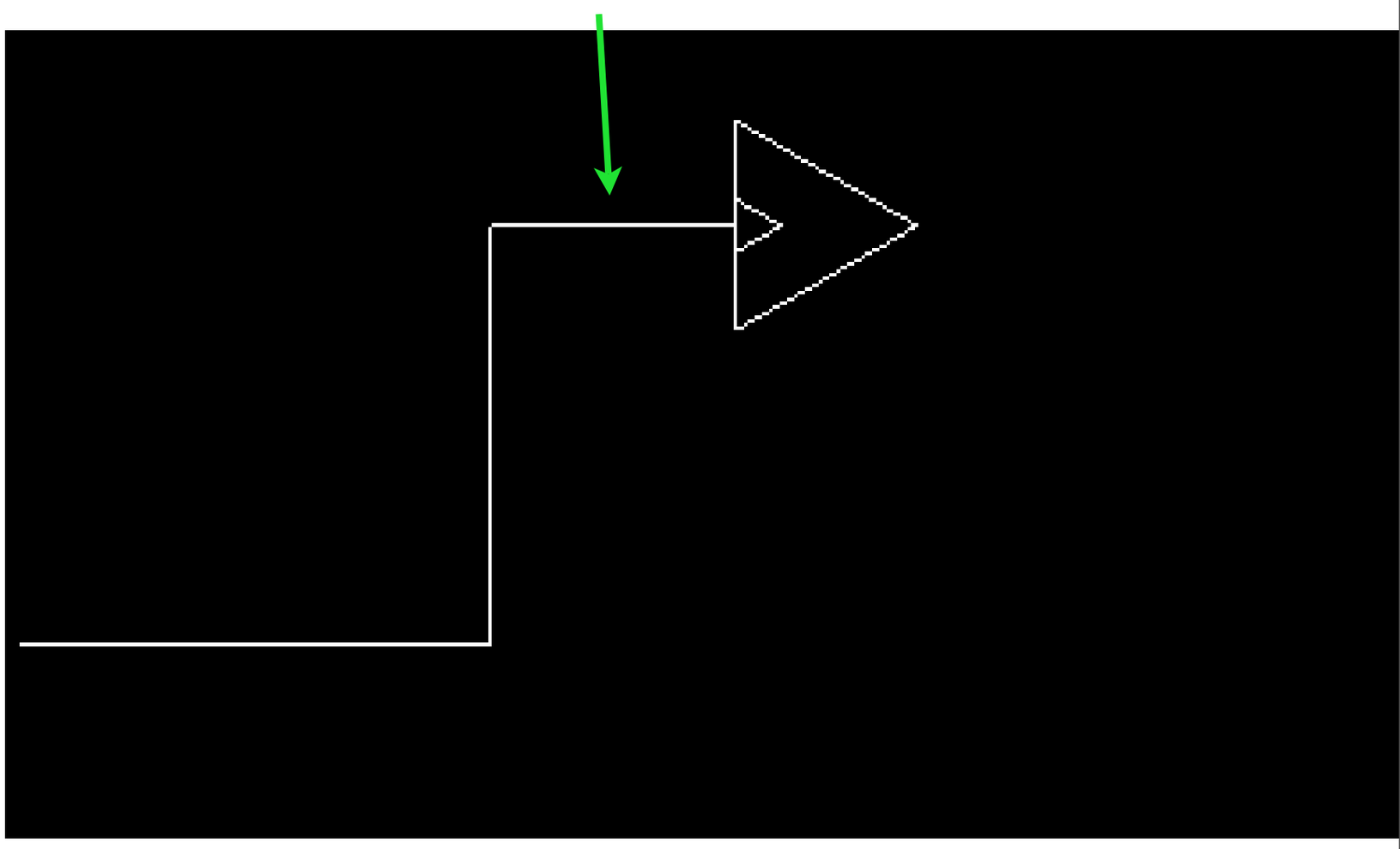


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
forward(thickness / 2)
```

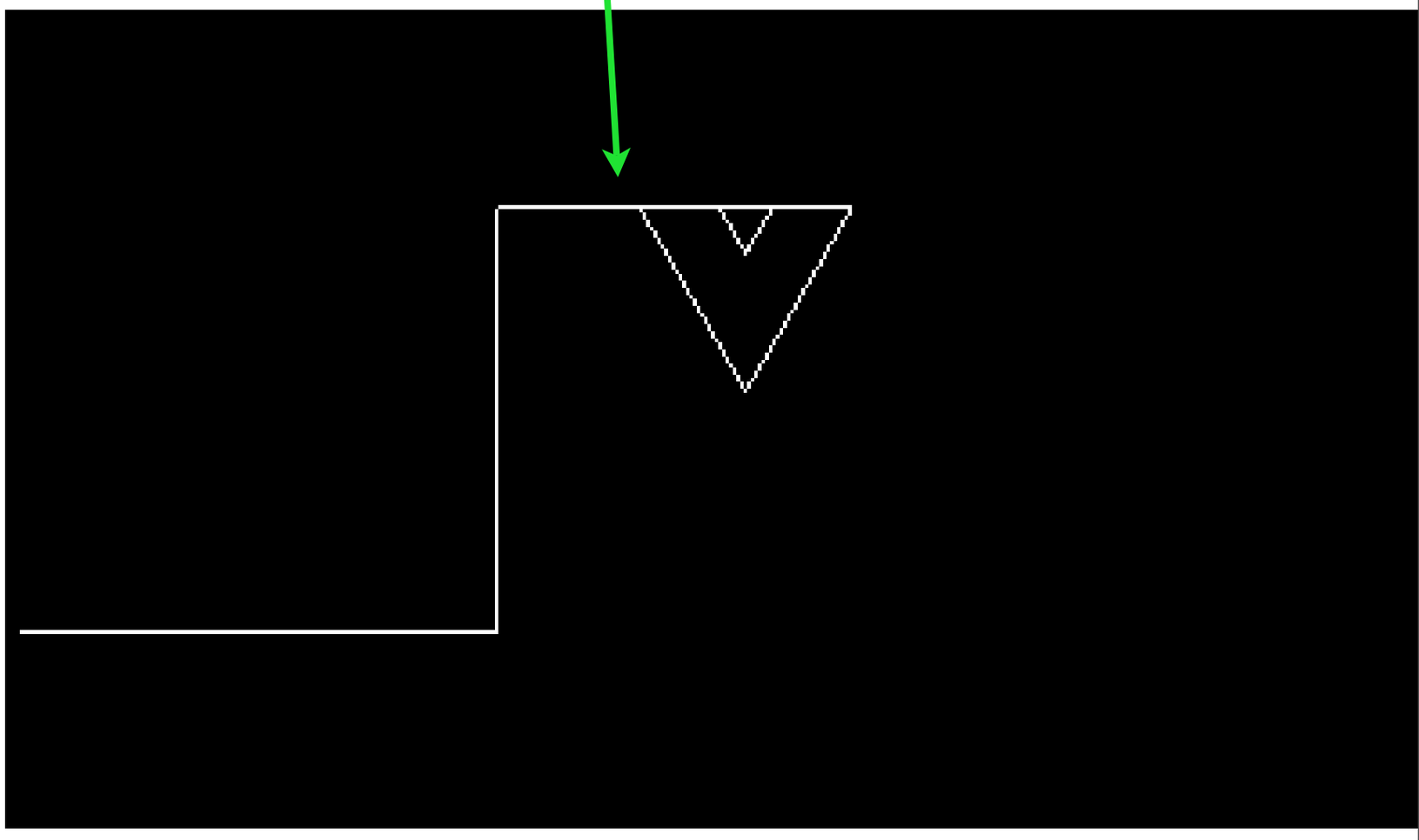


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
forward(thickness / 2)
right(90)
```

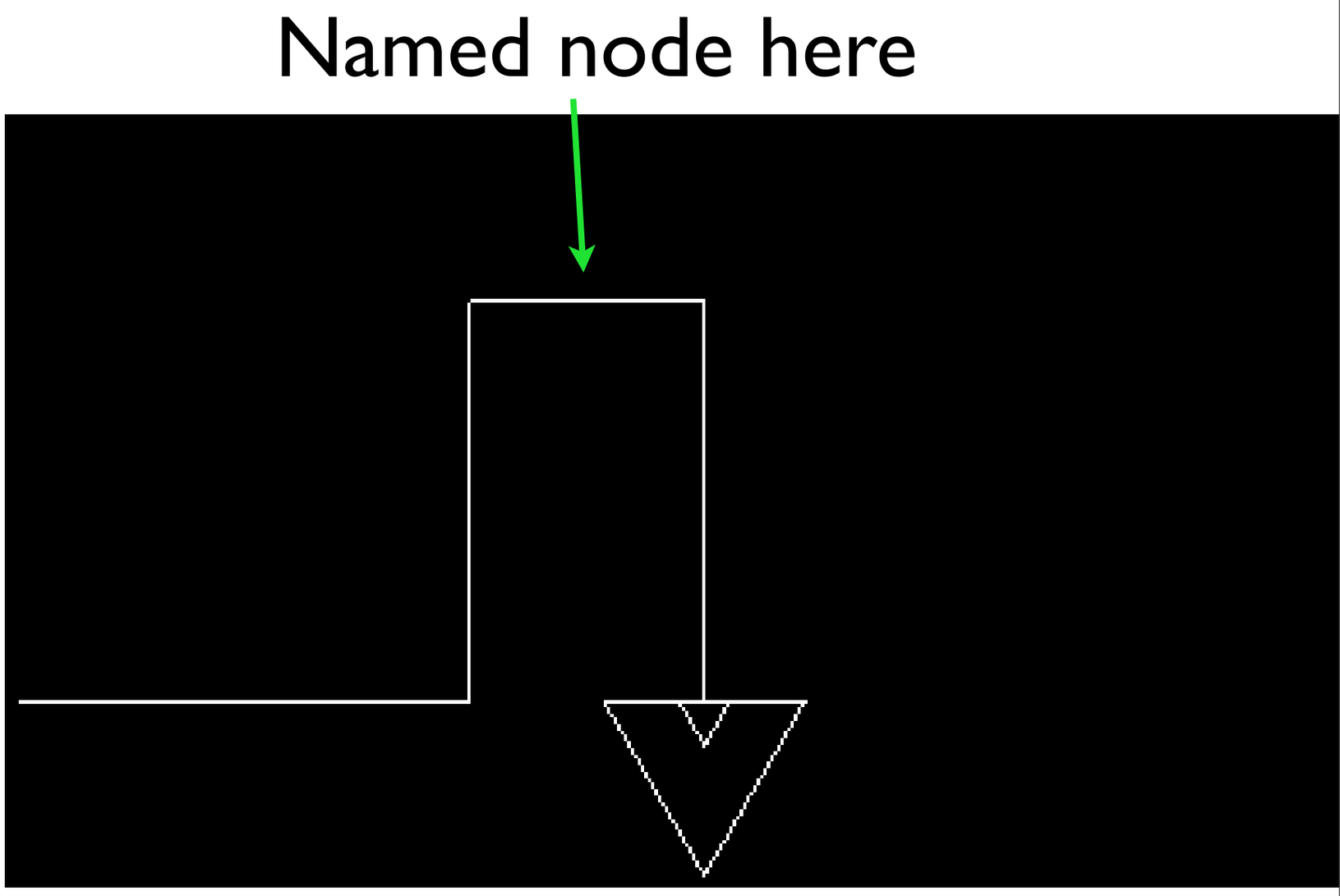


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
forward(thickness / 2)
right(90)
forward(notch_depth)
```

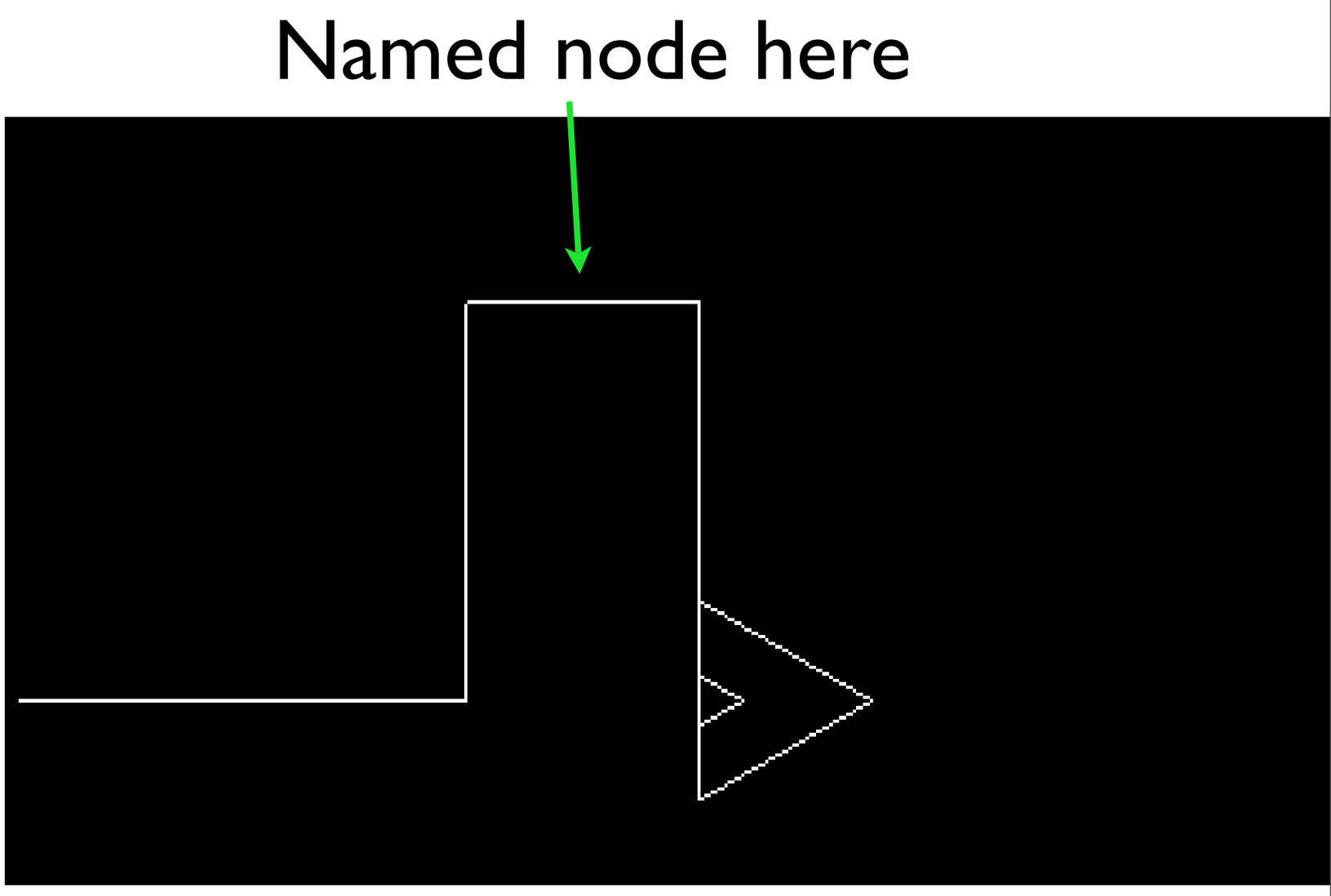


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
forward(thickness / 2)
right(90)
forward(notch_depth)
left(90)
```

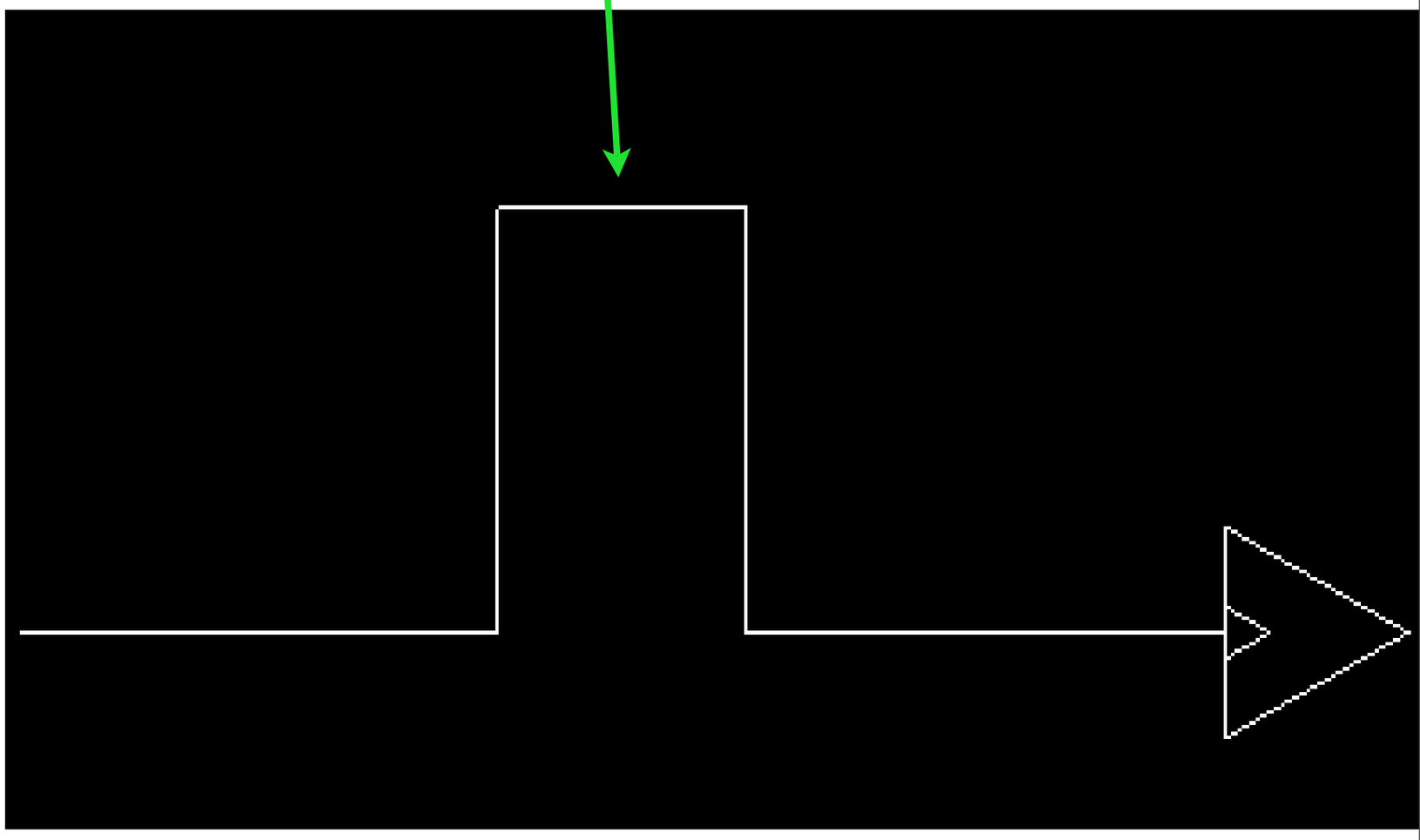


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
forward(side_length / 2)
left(90)
forward(notch_depth)
right(90)
forward(thickness / 2)
mark(name)
forward(thickness / 2)
right(90)
forward(notch_depth)
left(90)
forward(side_length / 2)
```

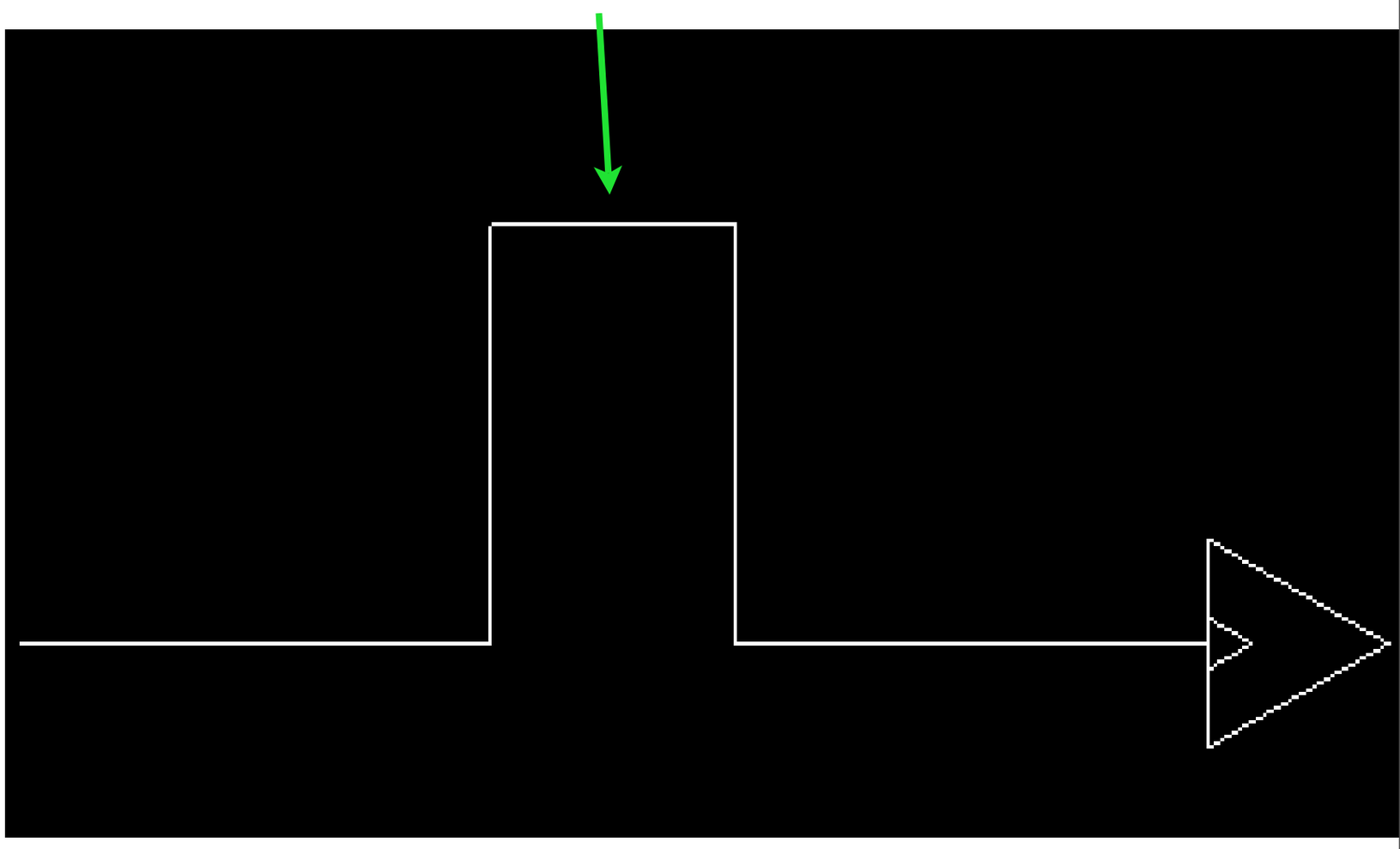


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
define side(name)
    forward(side_length / 2)
    left(90)
    forward(notch_depth)
    right(90)
    forward(thickness / 2)
    mark(name)
    forward(thickness / 2)
    right(90)
    forward(notch_depth)
    left(90)
    forward(side_length / 2)
done
```

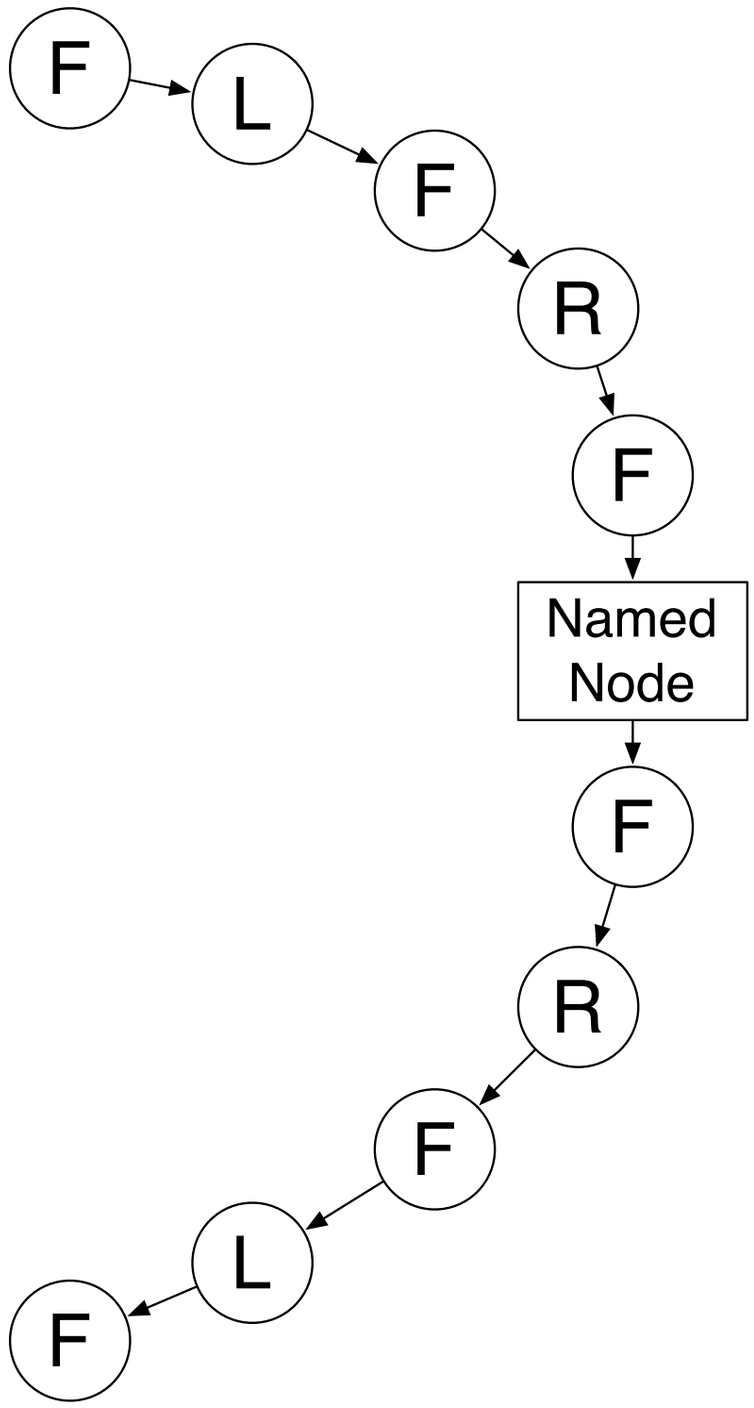


We may not know in advance how big to make each triangle, or how thick the material will be, or how deep the notches should be. We will use parameters for these values. Some of the arithmetic has been simplified here.

Each side of the triangle has a notch in it. To make this notch, we will have the turtle walk the following path. First we move forward half the length of the side, turn left, move forward again to make the notch, and turn right. So far we've only been using standard turtle operations that have been around since early versions of LOGO. But next we're going to mark a turtle location by giving it a name. This will allow us to refer to it later on. We want the named location to be at the middle bottom of the notch. So, we move forward half the thickness of the material (which is the notch's width), and give that spot a name using the 'mark' function. We can then proceed as normal. Of course we can give this list of operations a name, so we'll define a function called 'side', which takes a parameter for the name of the notch. (1:04)

Turtle Tree Example

```
define side(name)
    forward(side_length / 2)
    left(90)
    forward(notch_depth)
    right(90)
    forward(thickness / 2)
    mark(name)
    forward(thickness / 2)
    right(90)
    forward(notch_depth)
    left(90)
    forward(side_length / 2)
done
```

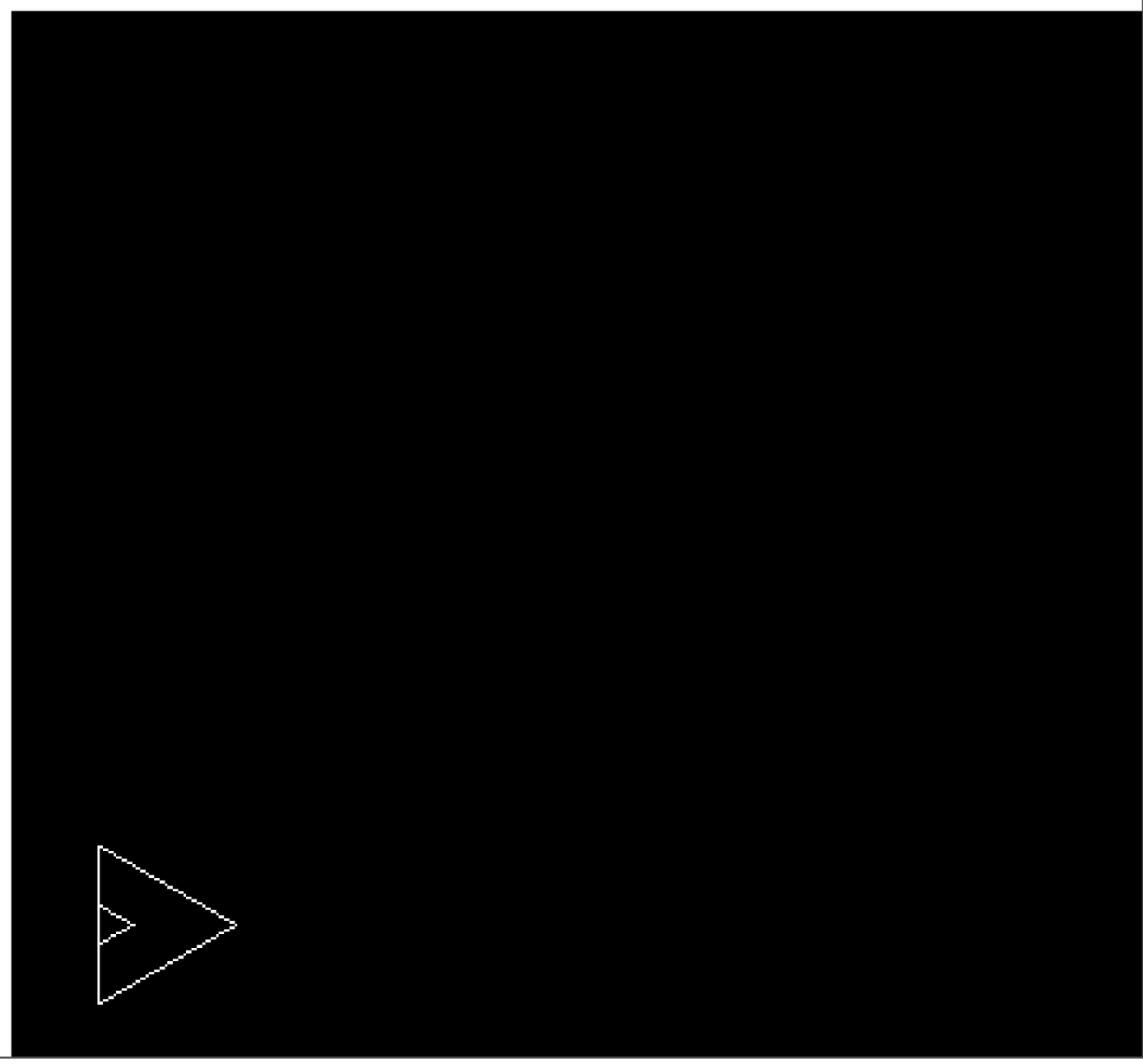


Here's a visualization of that turtle tree. Right now it isn't much of a tree, since it is only a sequential list. I'll return to that shortly. You can see the geometric operations like Forward, Left, Forward, and so on. In the middle is the named node. (0:15)

Turtle Tree Example

Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

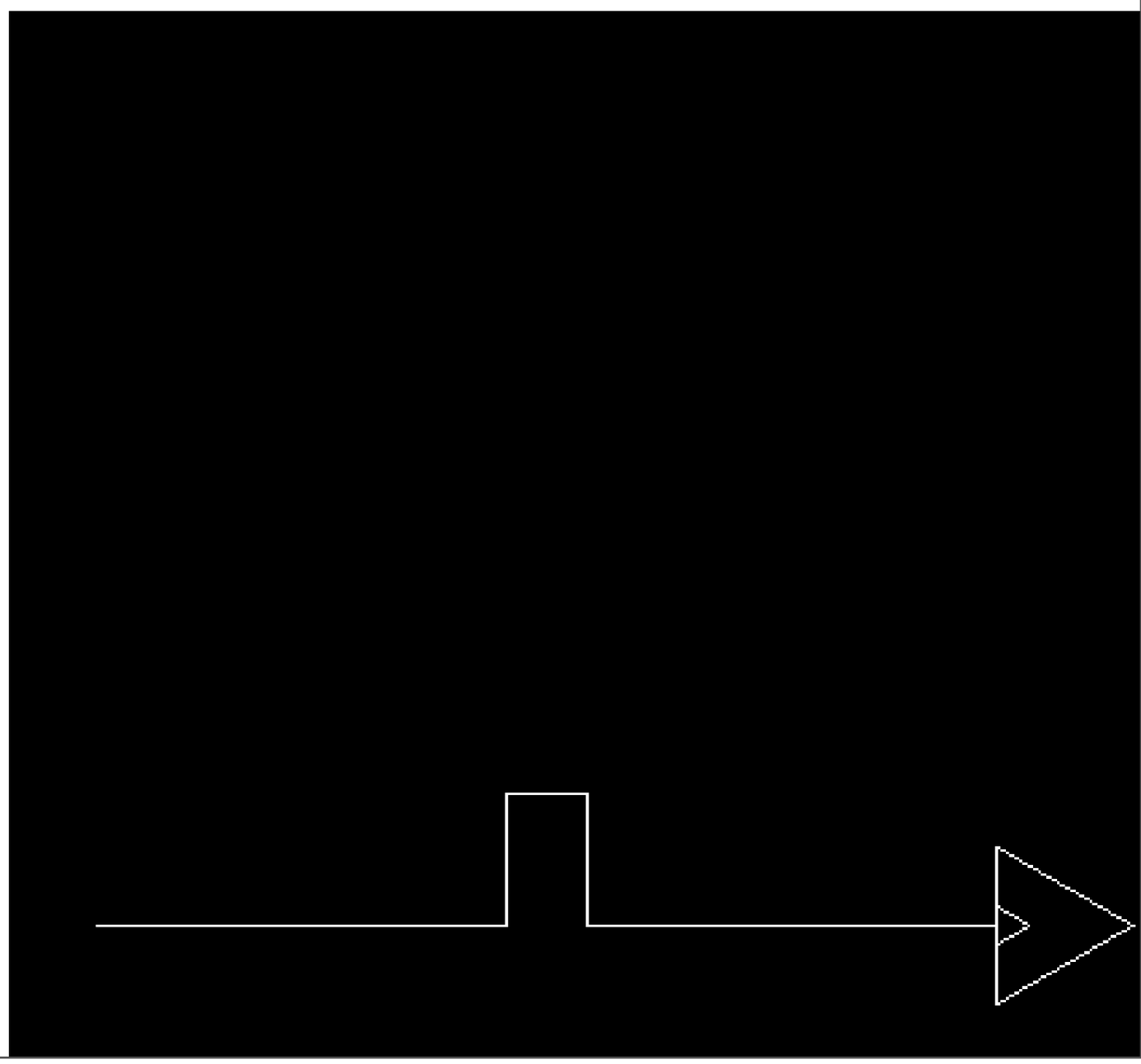
Turtle Tree Example



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

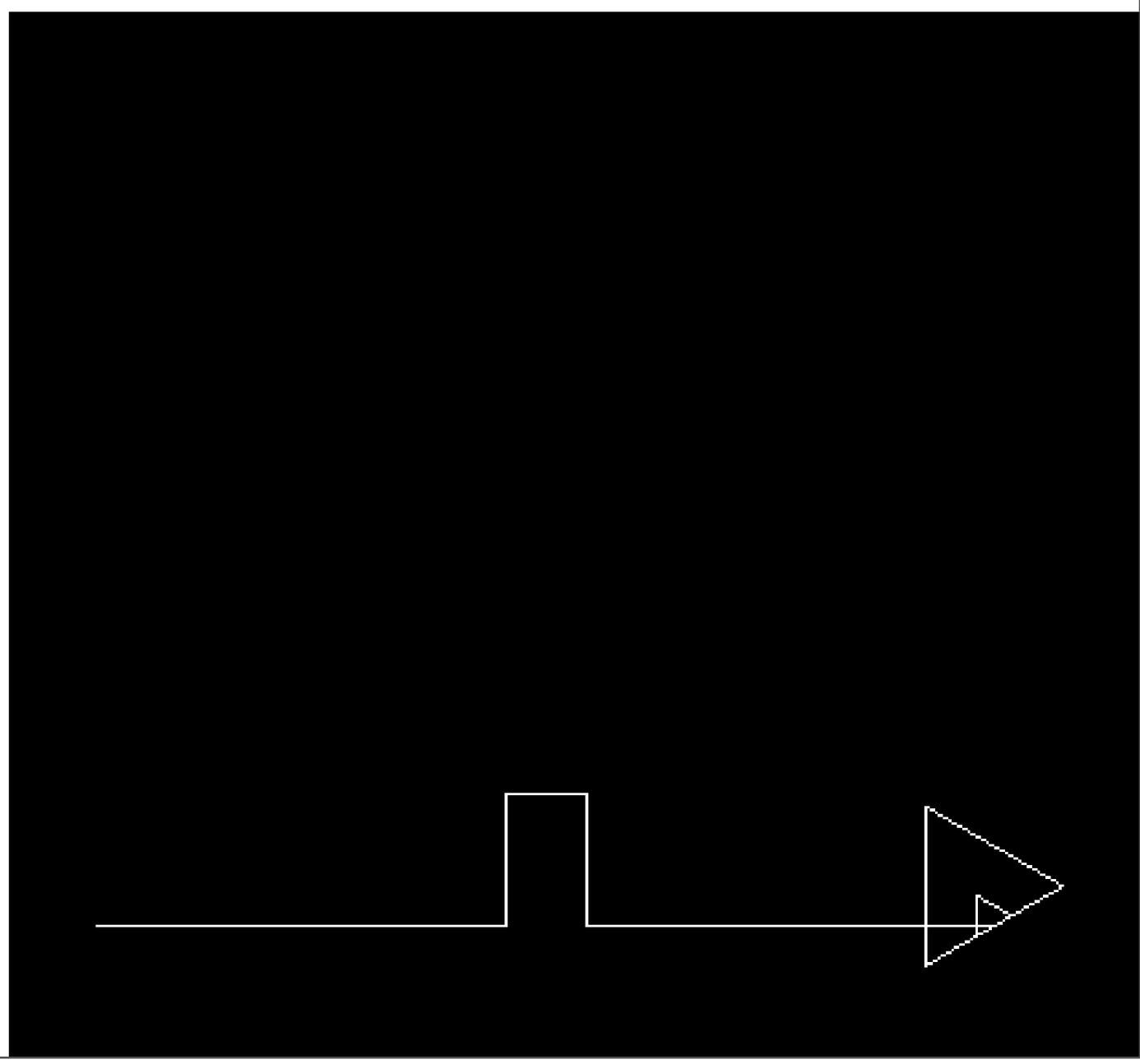
side(“a”)



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

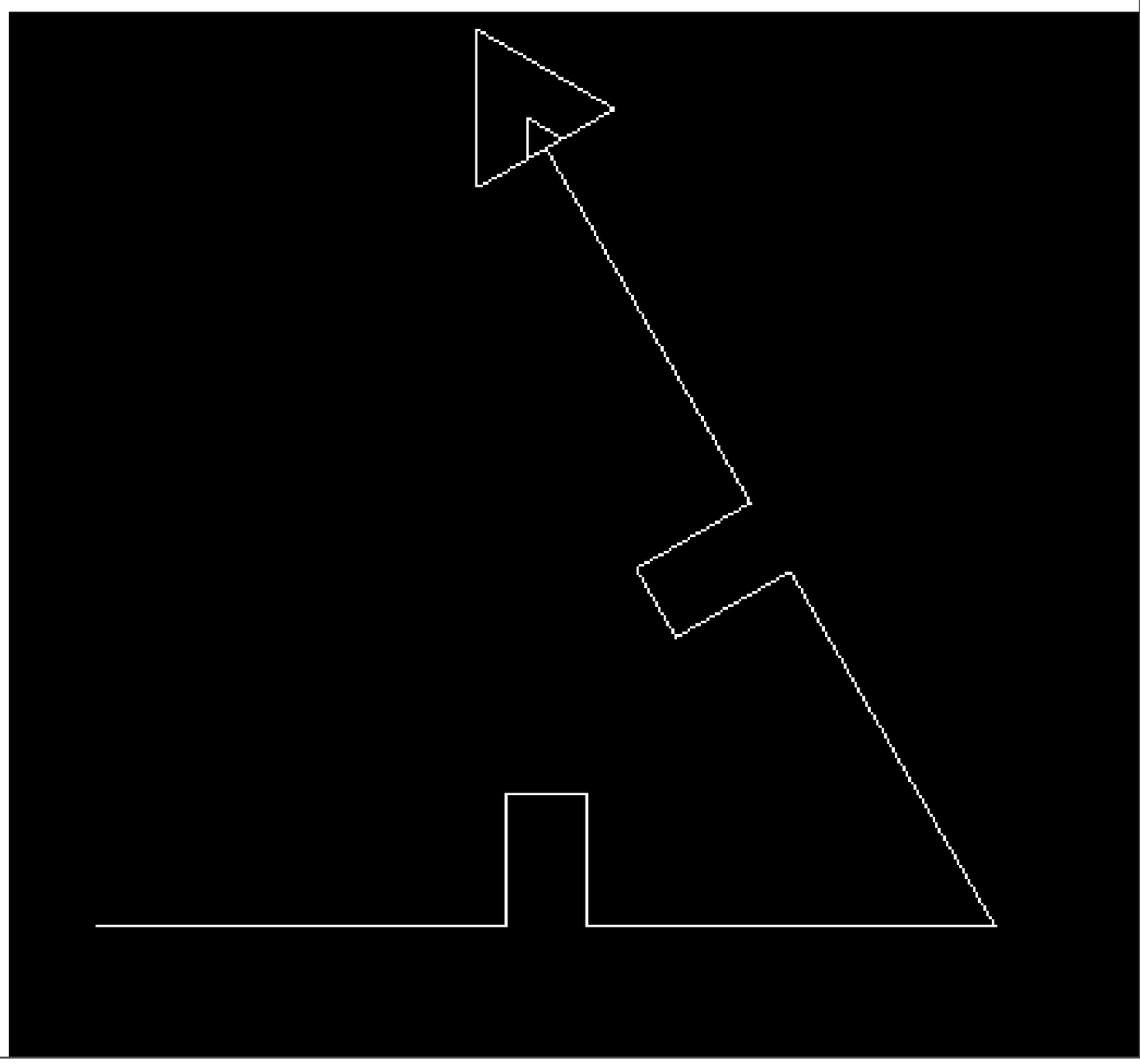
side(“a”)
left(120)



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

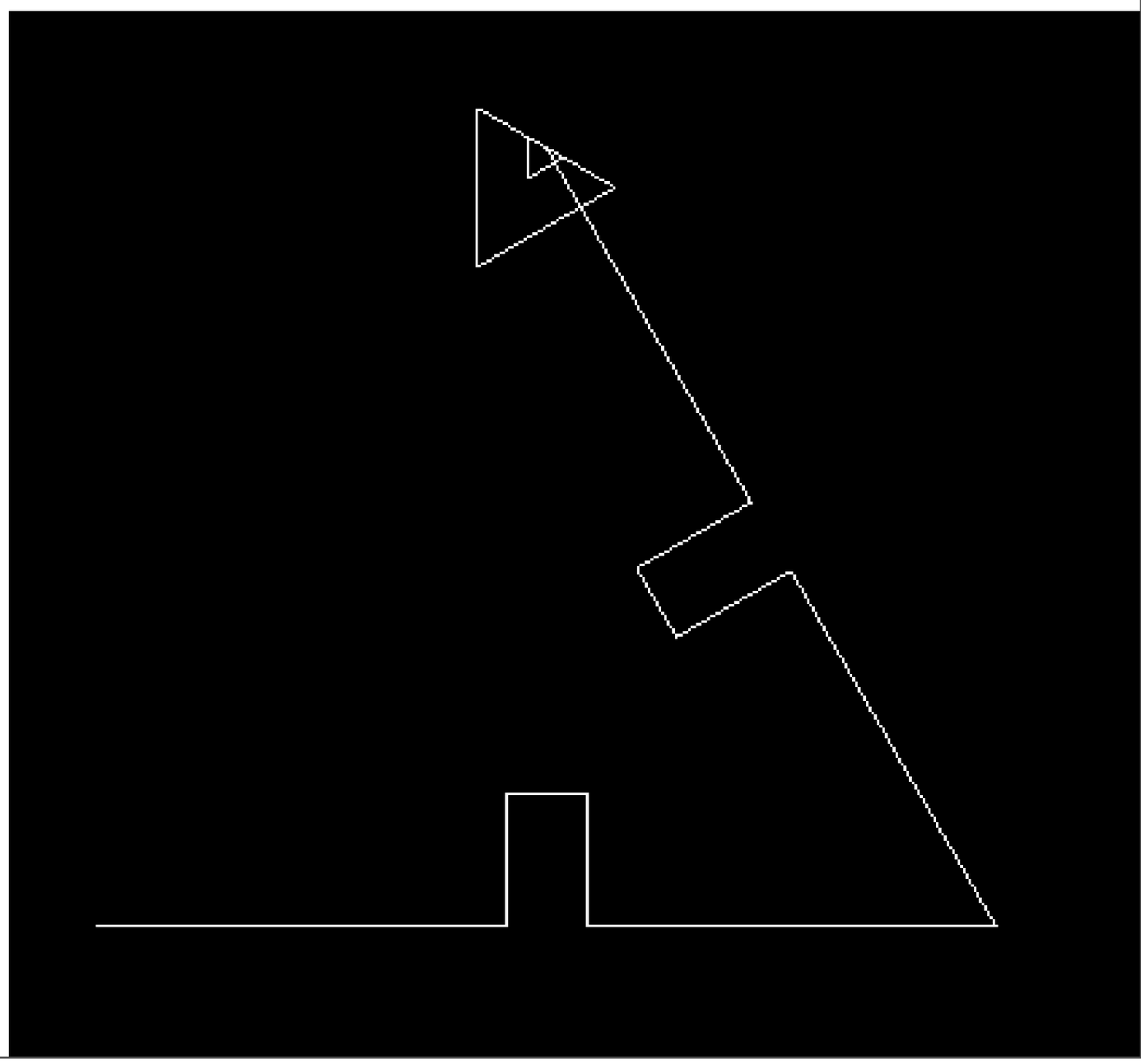
```
side(“a”)
left(120)
side(“b”)
```



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

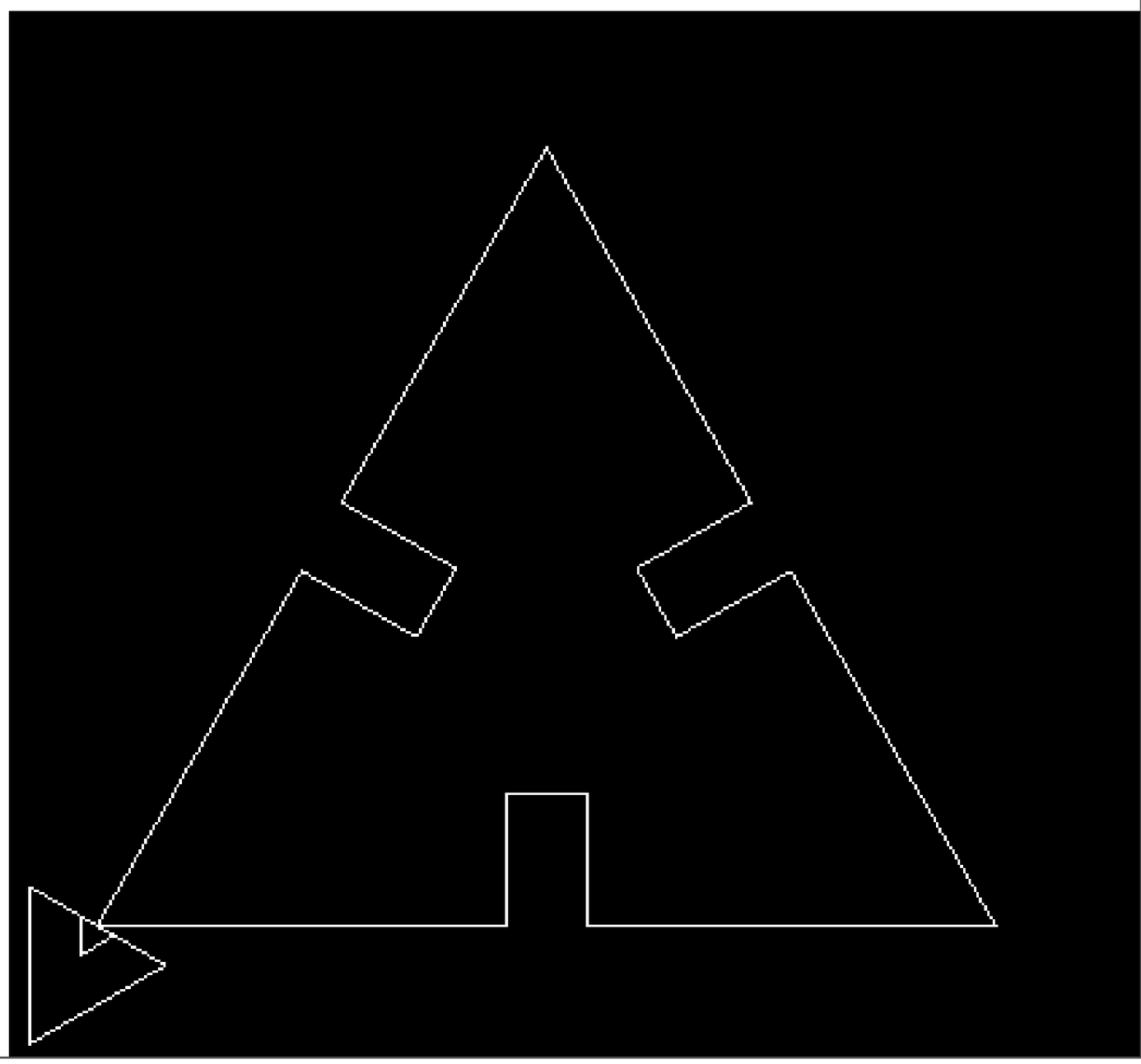
```
side(“a”)
left(120)
side(“b”)
left(120)
```



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

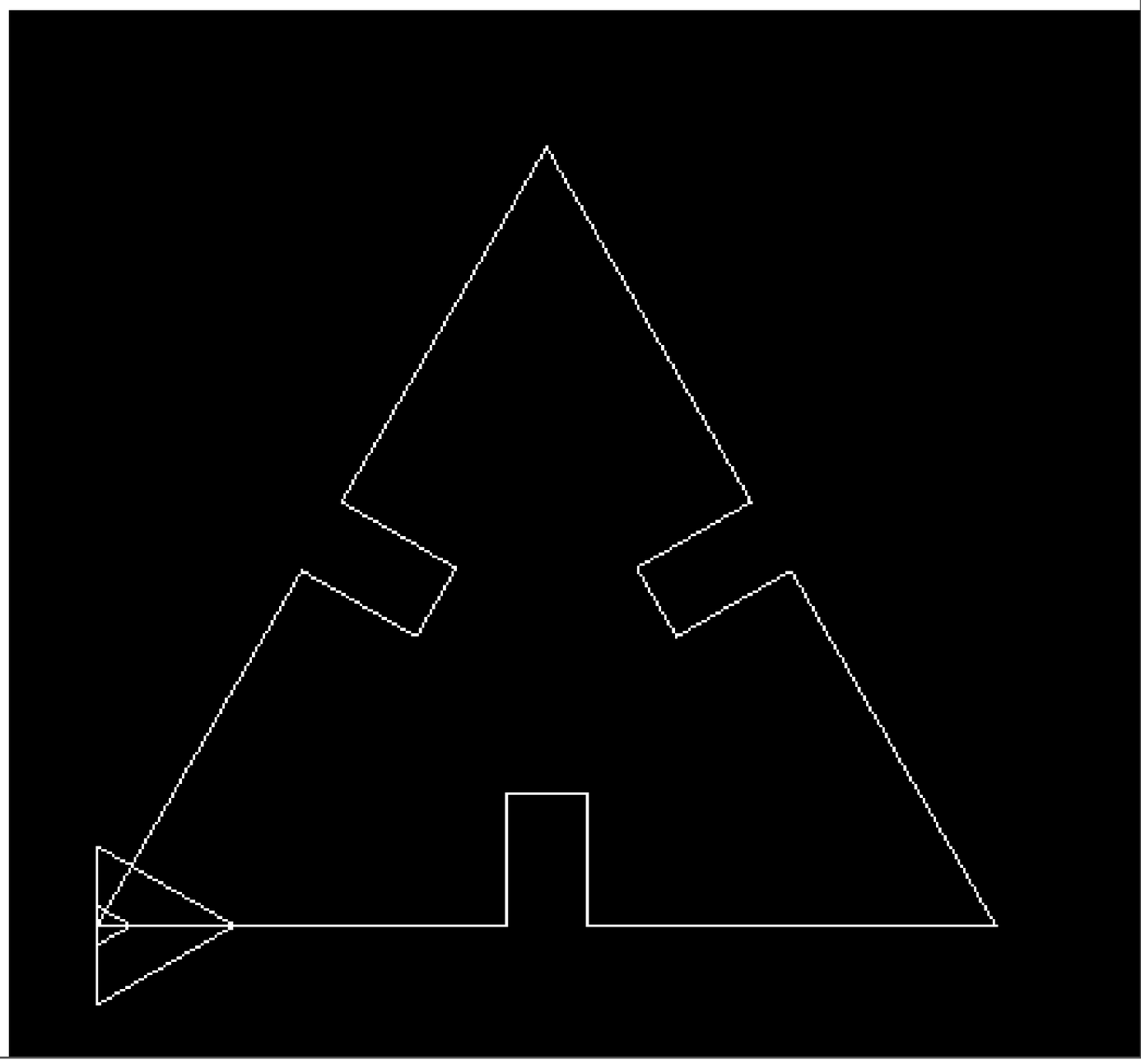
```
side(“a”)
left(120)
side(“b”)
left(120)
side(“c”)
```



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

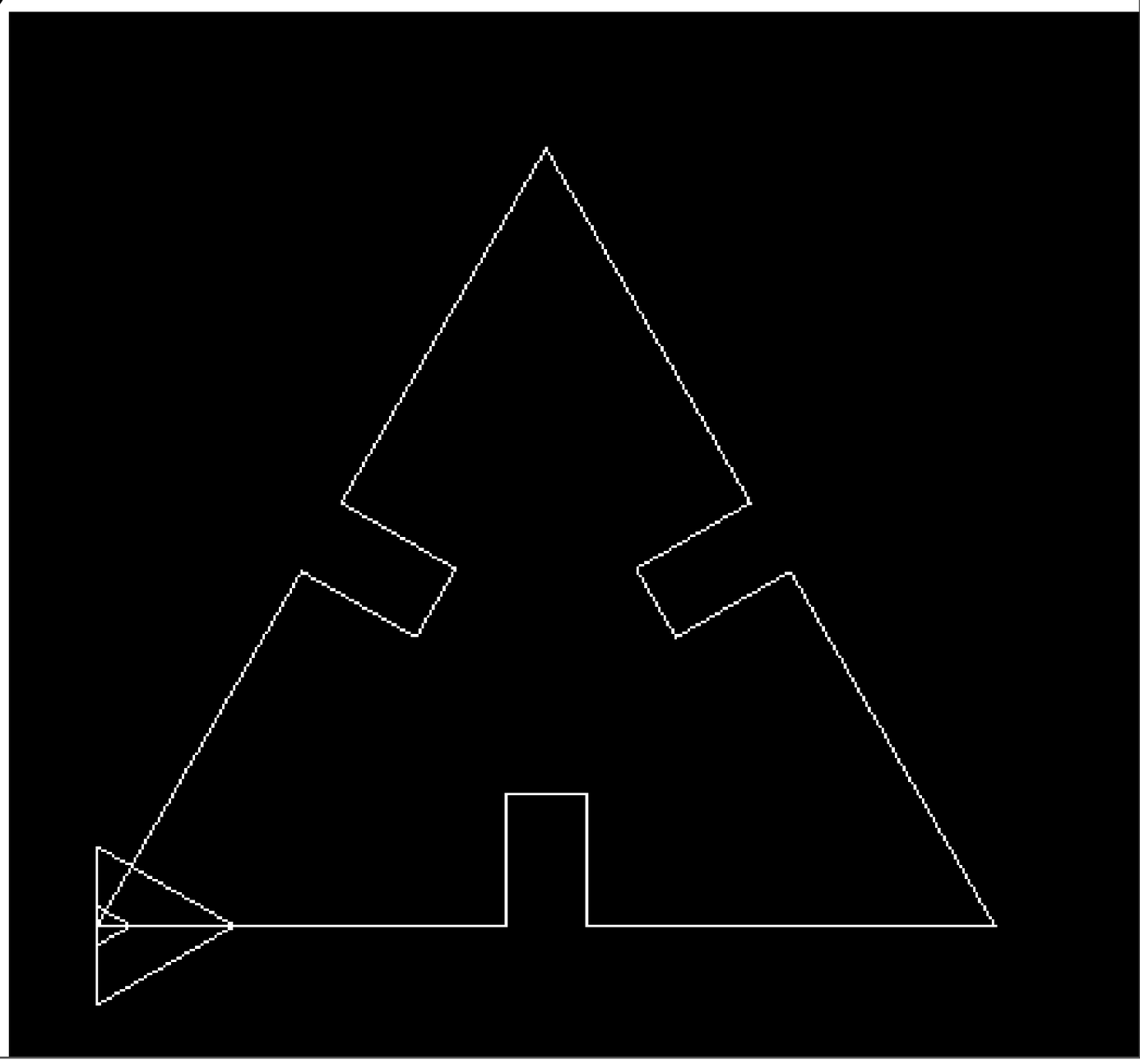
```
side(“a”)
left(120)
side(“b”)
left(120)
side(“c”)
left(120)
```



Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

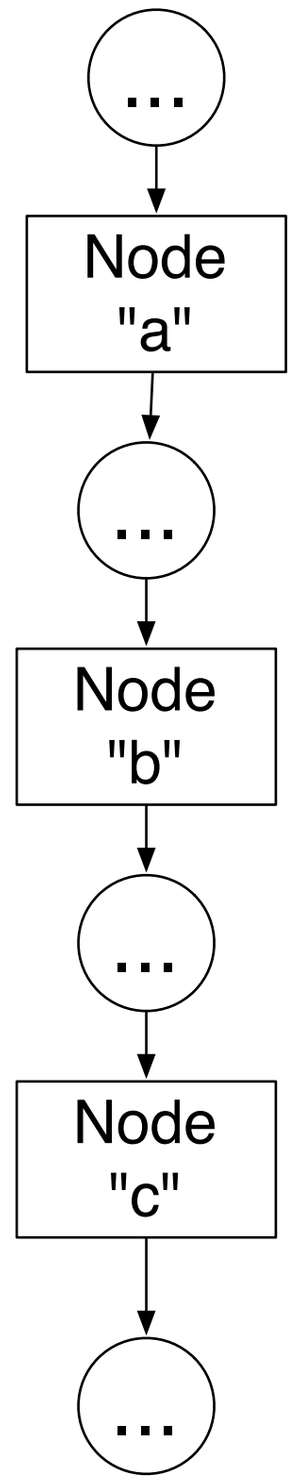
```
define notched_triangle()
    side("a")
    left(120)
    side("b")
    left(120)
    side("c")
    left(120)
done
```



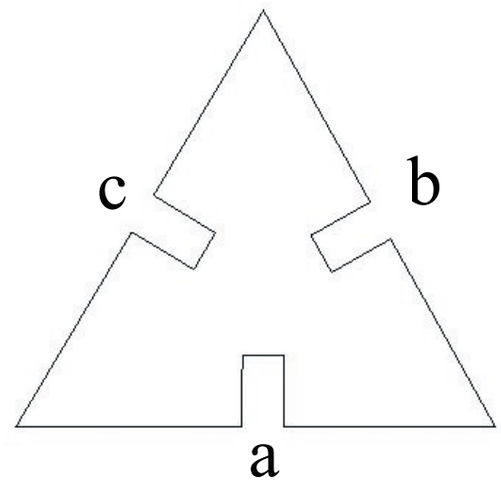
Now that we have a way to make sides with arbitrary names, lets make a triangle. This is quite easy. We simply make a side, whose notch is named “a”, turn left 120 degrees, and continue, naming the other sides “b” and “c”. Like always, we can turn that code into a function. I’ll name it “notched triangle”. (0:18)

Turtle Tree Example

```
define notched_triangle()
    side(“a”)
    left(120)
    side(“b”)
    left(120)
    side(“c”)
    left(120)
done
```

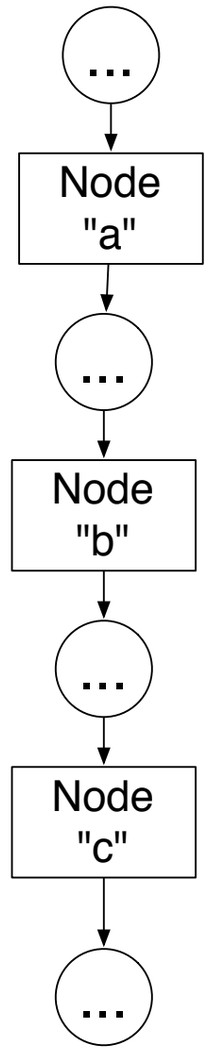


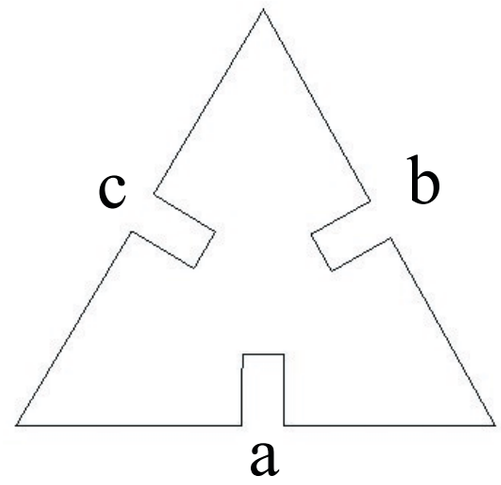
Here is the linear turtle tree for a notched triangle. I’ve collapsed the geometry nodes for clarity. (0:08)



Recall we're making a notched triangle whose named positions look like this, and a turtle tree like this. Now I'm going to turn this function call into a "shape", which is a technical term in FlatLang. All code inside the "shape" block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the "draw" function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The "from" command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes "b" and "c", and orients the turtle before drawing a new triangle shape beginning at node "a". Now you can see that the turtle tree finally has branches in it! (1:03)

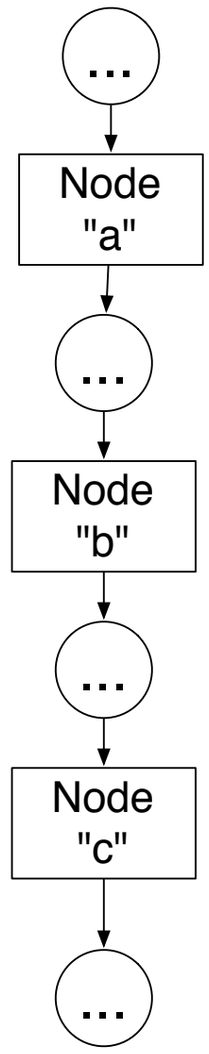


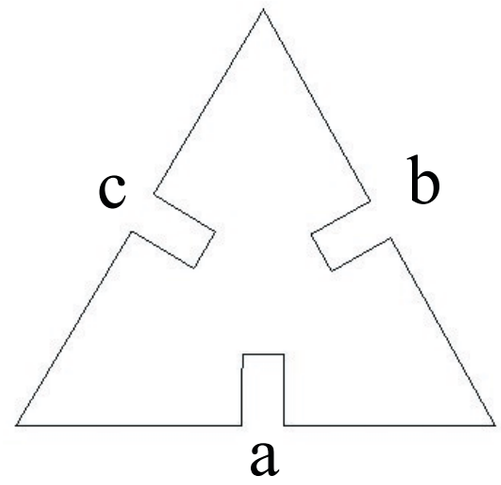


Recall we're making a notched triangle whose named positions look like this, and a turtle tree like this. Now I'm going to turn this function call into a "shape", which is a technical term in FlatLang. All code inside the "shape" block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the "draw" function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The "from" command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes "b" and "c", and orients the turtle before drawing a new triangle shape beginning at node "a". Now you can see that the turtle tree finally has branches in it! (1:03)

```
shape(“triangle”)
  notched_triangle()
done
```

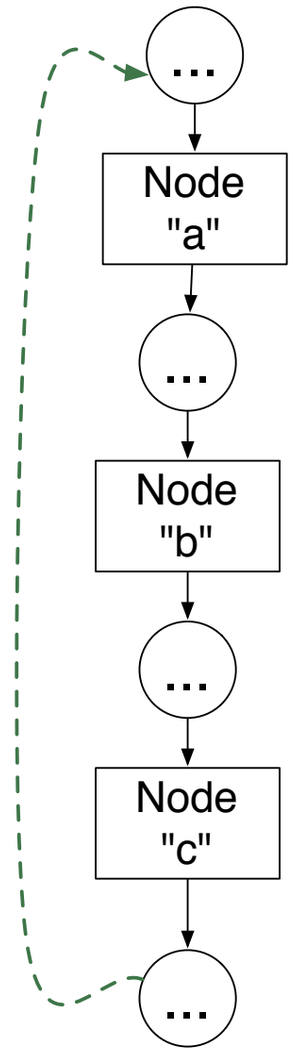


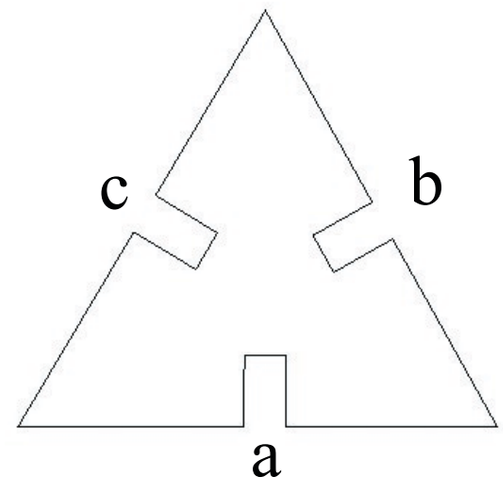


Recall we’re making a notched triangle whose named positions look like this, and a turtle tree like this. Now I’m going to turn this function call into a “shape”, which is a technical term in FlatLang. All code inside the “shape” block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the “draw” function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The “from” command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes “b” and “c”, and orients the turtle before drawing a new triangle shape beginning at node “a”. Now you can see that the turtle tree finally has branches in it! (1:03)

```
shape(“triangle”)
  notched_triangle()
done
```

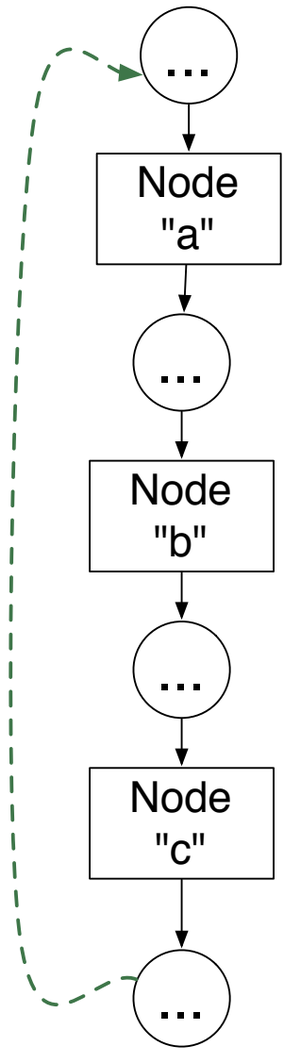


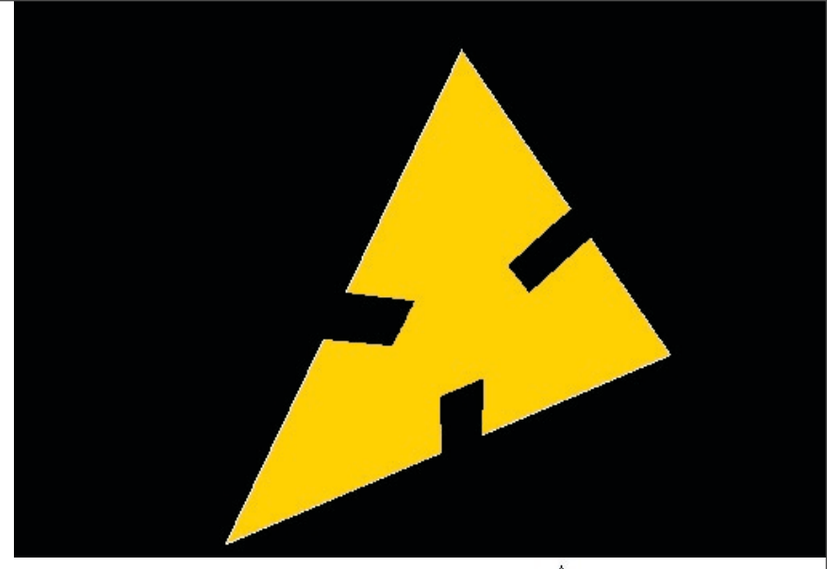


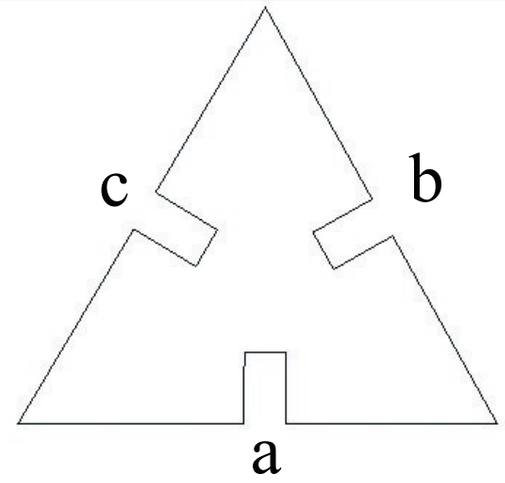
Recall we’re making a notched triangle whose named positions look like this, and a turtle tree like this. Now I’m going to turn this function call into a “shape”, which is a technical term in FlatLang. All code inside the “shape” block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the “draw” function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The “from” command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes “b” and “c”, and orients the turtle before drawing a new triangle shape beginning at node “a”. Now you can see that the turtle tree finally has branches in it! (1:03)

```
shape(“triangle”)
  notched_triangle()
done
draw(“triangle”,“a”)
```



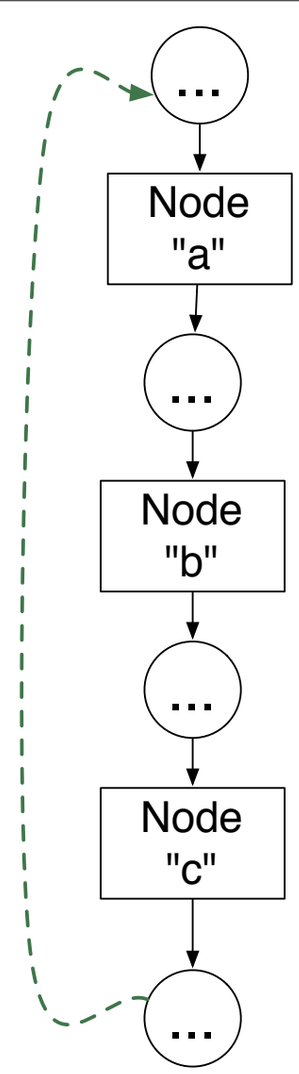


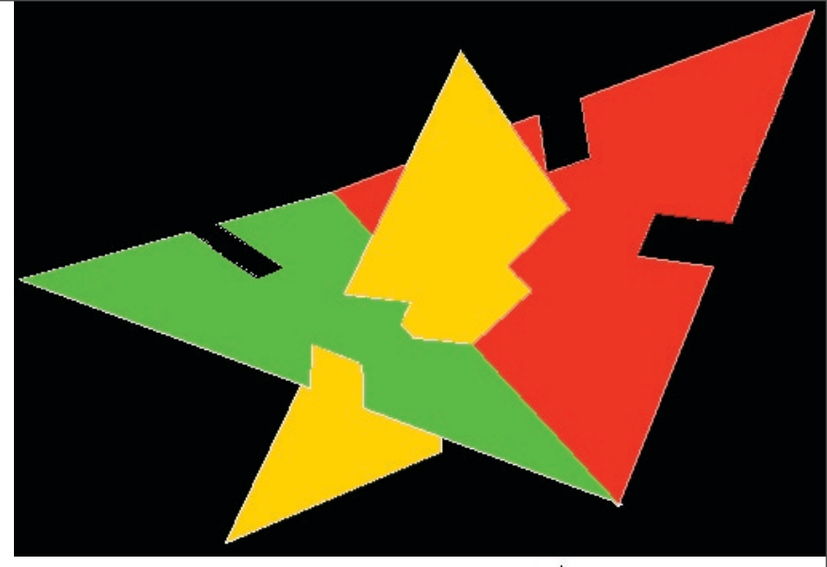


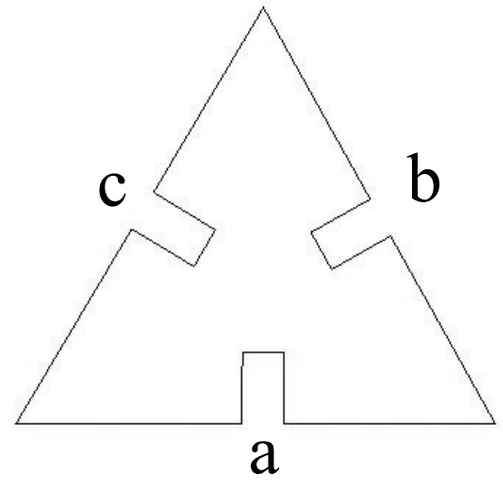
Recall we’re making a notched triangle whose named positions look like this, and a turtle tree like this. Now I’m going to turn this function call into a “shape”, which is a technical term in FlatLang. All code inside the “shape” block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the “draw” function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The “from” command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes “b” and “c”, and orients the turtle before drawing a new triangle shape beginning at node “a”. Now you can see that the turtle tree finally has branches in it! (1:03)

```
shape(“triangle”)
  notched_triangle()
done
draw(“triangle”,“a”)
from(“b”,“c”)
  pitch(90)
  left(180)
  draw(“triangle”,“a”)
done
```



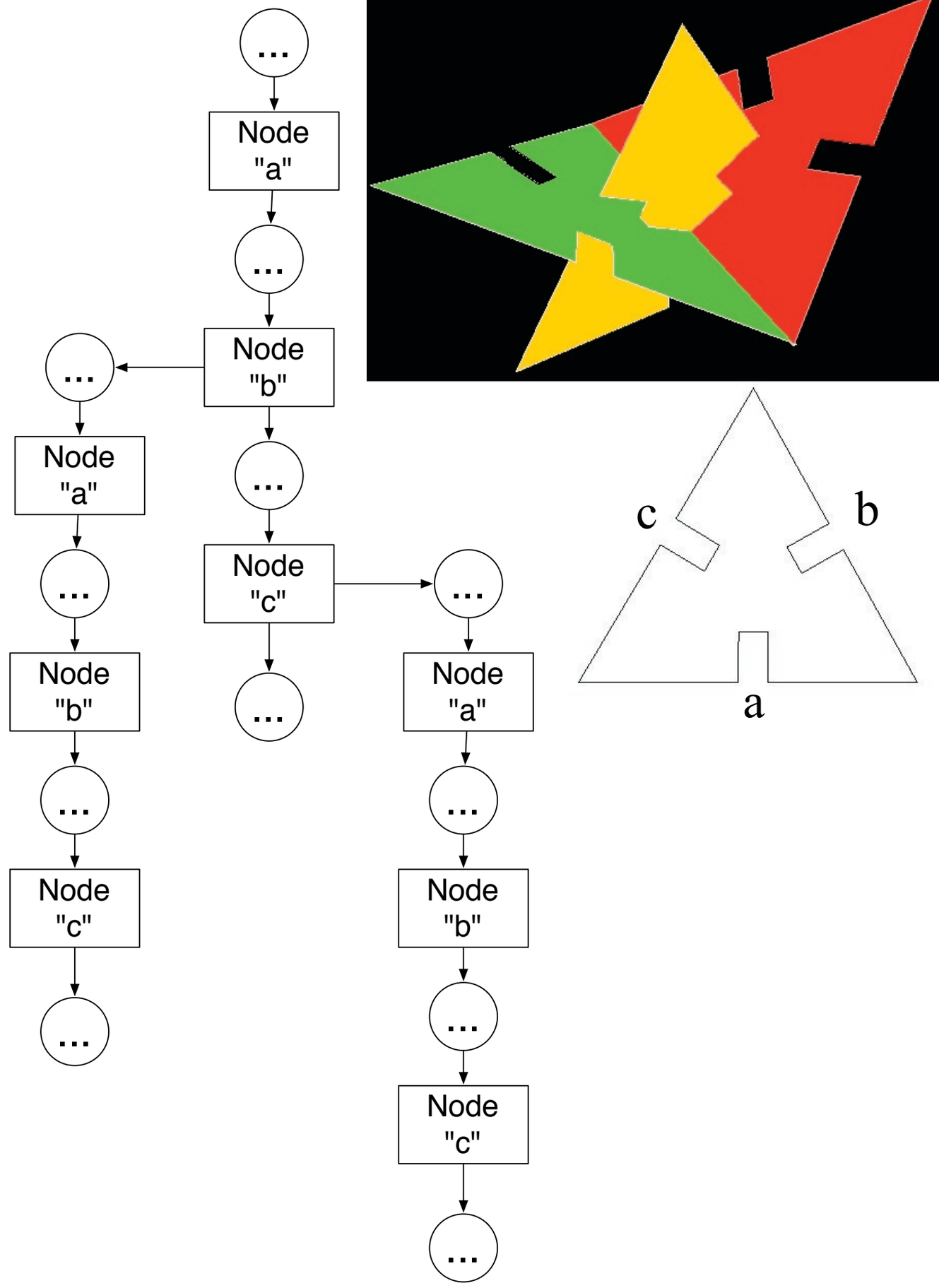




Recall we’re making a notched triangle whose named positions look like this, and a turtle tree like this. Now I’m going to turn this function call into a “shape”, which is a technical term in FlatLang. All code inside the “shape” block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the “draw” function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The “from” command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes “b” and “c”, and orients the turtle before drawing a new triangle shape beginning at node “a”. Now you can see that the turtle tree finally has branches in it! (1:03)

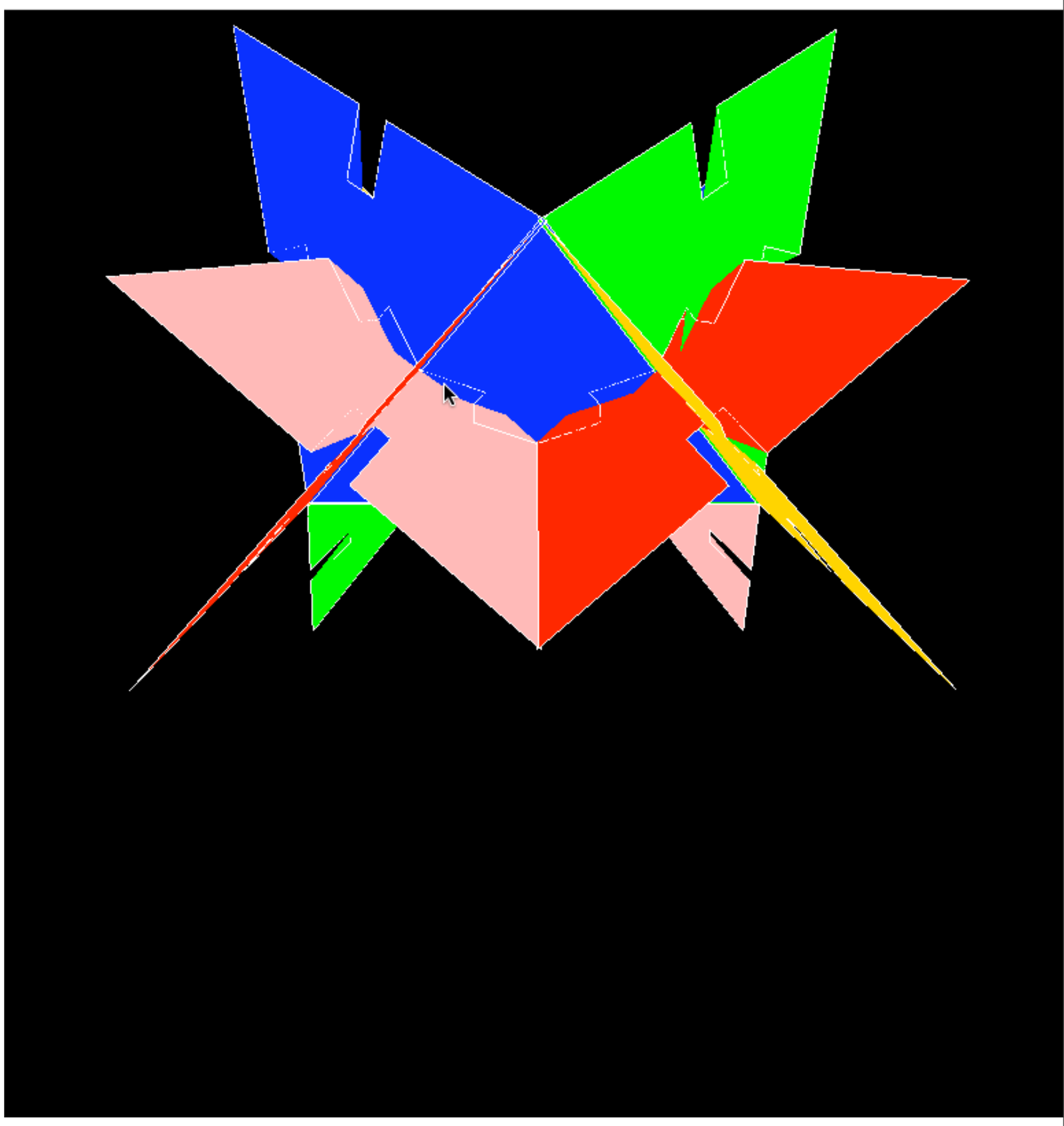
```
shape(“triangle”)
  notched_triangle()
done
draw(“triangle”,“a”)
from(“b”,“c”)
  pitch(90)
  left(180)
  draw(“triangle”,“a”)
done
```



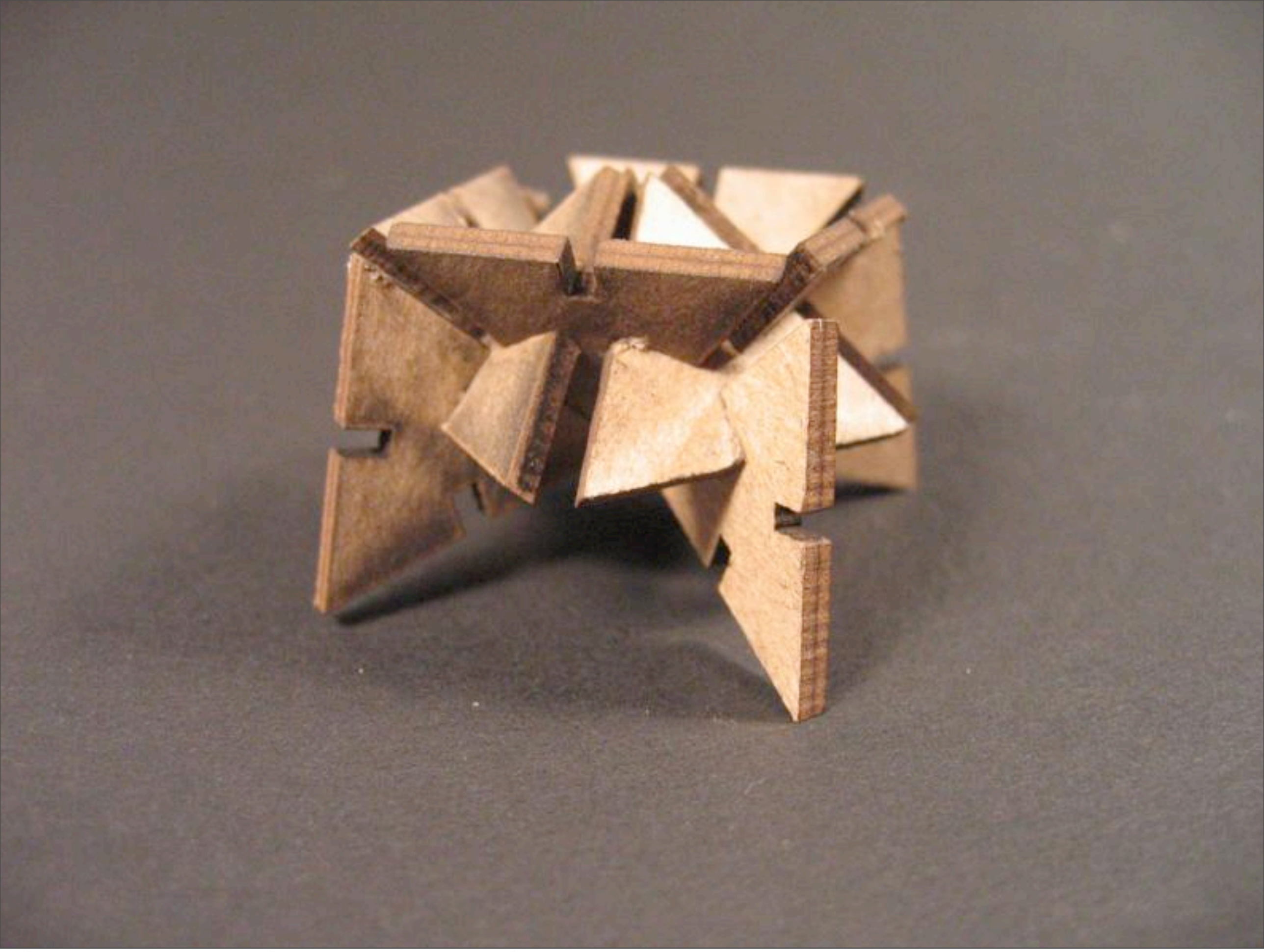
Recall we’re making a notched triangle whose named positions look like this, and a turtle tree like this. Now I’m going to turn this function call into a “shape”, which is a technical term in FlatLang. All code inside the “shape” block is executed and recorded in an off-screen buffer. It also adds an implicit link from the last turtle operation back to the first. That way we can use the “draw” function to draw named shapes beginning at any of its named locations. This is used to place shapes exactly where we want them, even if the code that made that shape began at some other location. In this case, the code begins at one of the corners, but we would like to work with notched triangles in terms of their notch locations.

The “from” command takes a list of named locations in the turtle tree and a block of code to execute at each of those locations. This code begins from nodes “b” and “c”, and orients the turtle before drawing a new triangle shape beginning at node “a”. Now you can see that the turtle tree finally has branches in it! (1:03)

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

go(3)
```



We can nest "from" blocks. Here is some recursive code similar to the previous shape example. On the right you can see the user viewing this model from different perspectives by dragging on it. The turtle tree for this has 15 branches. (0:14)



If we were to manufacture the model from the last slide we would end up with many notched triangles. It is up to the user to assemble them. This has been assembled differently. That is part of the beauty of construction kits, because you can use the pieces to make many different things. (0:16)

FlatCAD Limitations

- Making curved, organic shape
- "Where is the code that made *this*?"
- Mismatch between visual and textual representation. Some programs are unfortunately "write only".

While FlatCAD supports design of many things, it is by no means the best or even adequate for some tasks. For example, we may be able to write a program that generates a spider web, but it would be much more difficult to write a program to make a convincing model of a spider.

The development environment could also be enhanced to help the designer better see the relationship between the visual output and the code that created it. There is a mismatch between what the programmer sees on the right and what they can read on the left, and vice versa. (0:32)

State Your Intentions In The Best Way You Can

- Structured modeling tools
- Informal sketching
- Designing by/with example
- Writing code

I began this talk by mentioning several ways of designing. Each of those ways have their strengths and weaknesses. I think that writing programs can also be a powerful method of expressing design intent, and FlatCAD has helped to make that case. Truly powerful design environments would let you express your intentions in whichever mode makes the most sense, whether that is using a structured modeler, freehand sketching, giving examples or, I hope, by writing code. (0:28)

FlatCAD and FlatLang: Kits by Code

Gabe Johnson < johnsogg@cmu.edu >
Carnegie Mellon University
Computational Design Lab (codelab)

VL/HCC - September 2008

flatcad.org

Thank You:
Mark Gross, Ellen Do, Shaun Moon

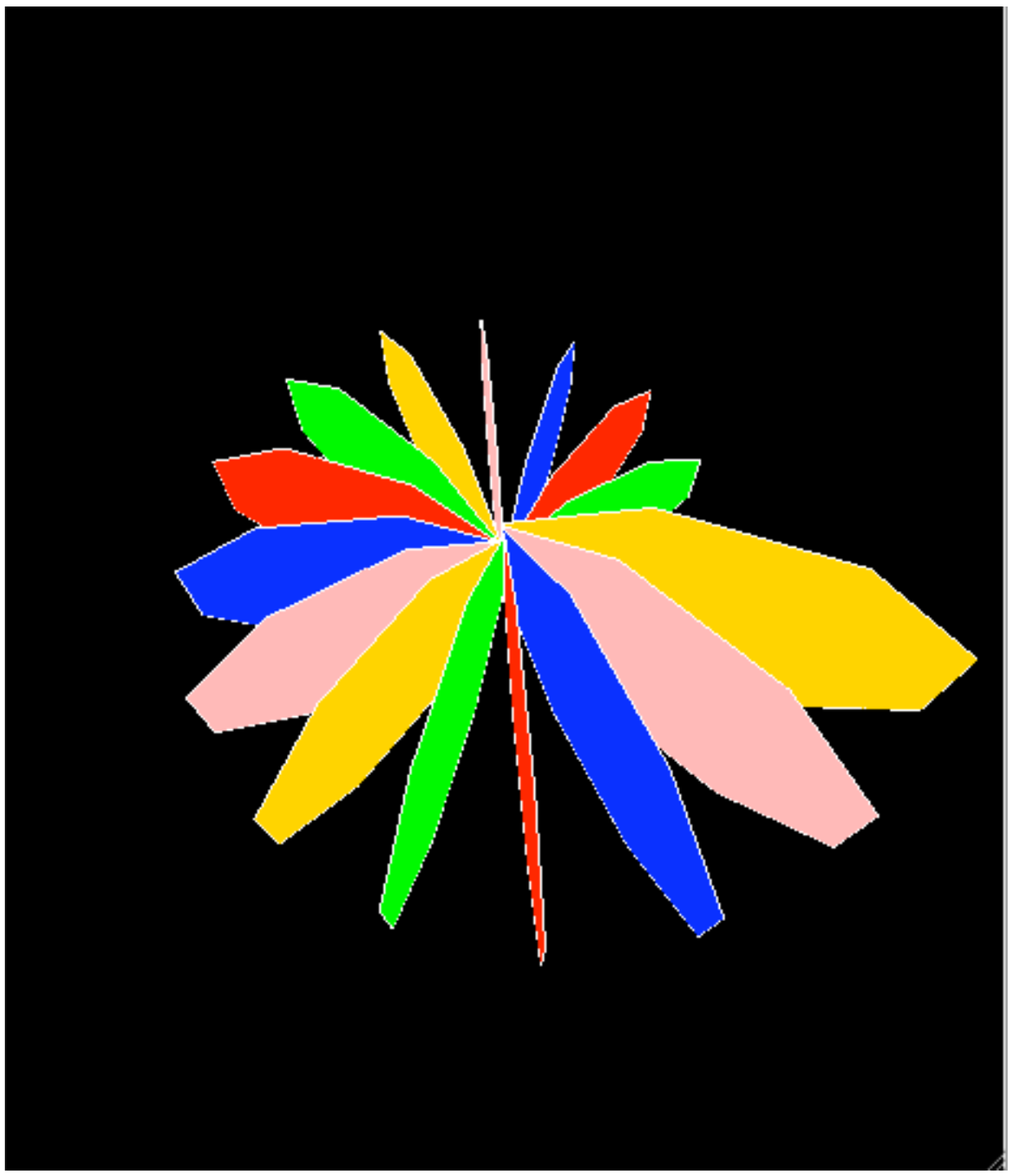
Funded by NSF grant ITR-0326054

If you're interested in trying FlatCAD for yourself, you can download binaries for Windows, OS X, or Linux. Just visit flatcad.org.

I'd like to thank my labmates at the codelab at CMU, especially my advisors Mark Gross and Ellen Do. Ellen is now at Georgia Tech. Also I want to thank Shaun Moon for creating Sewometry and giving me a lot of good feedback about FlatLang early on. Also I thank the NSF for funding the grant.

If you're interested in details about FlatCAD or FlatLang, please ask a question now or come find me later on. There's quite a bit not covered in this talk or in the paper. Thanks! (0:37)

flatcad.org - johnsogg@cmu.edu



flatcad.org - johnsogg@cmu.edu