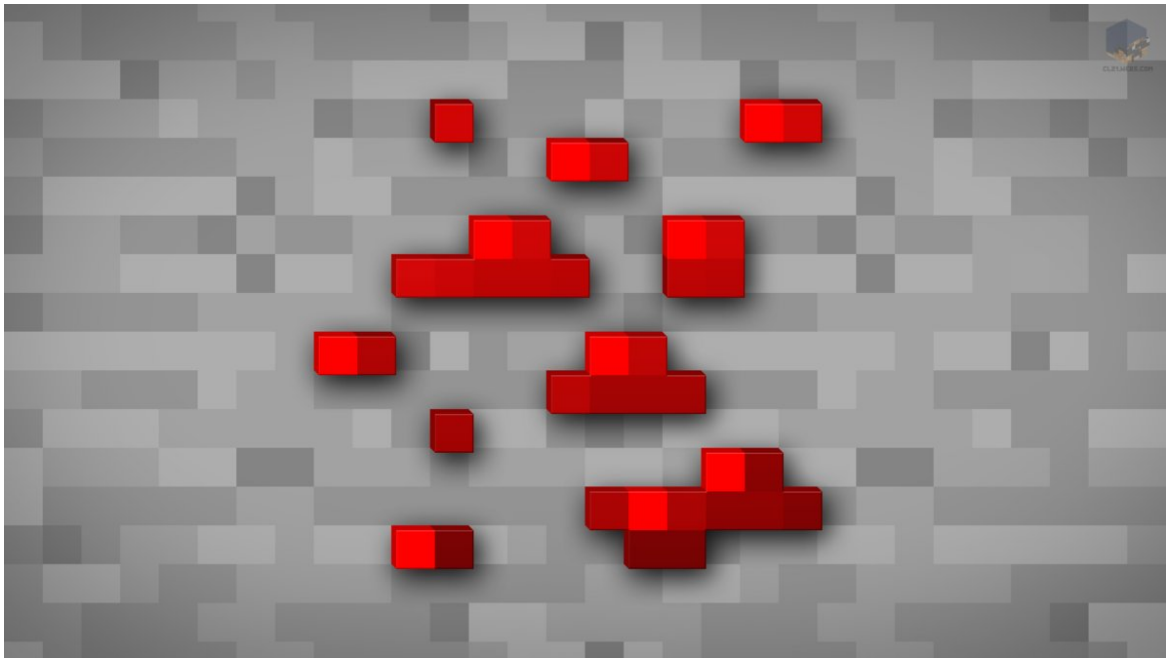


# Minecraft U Sequence 3: Advanced Redstone, ComputerCraft and Programming

Further exploring the possibilities of redstone crafting, the use of ComputerCraft robots and moving into more traditional programming within the a visual Javascript environment.

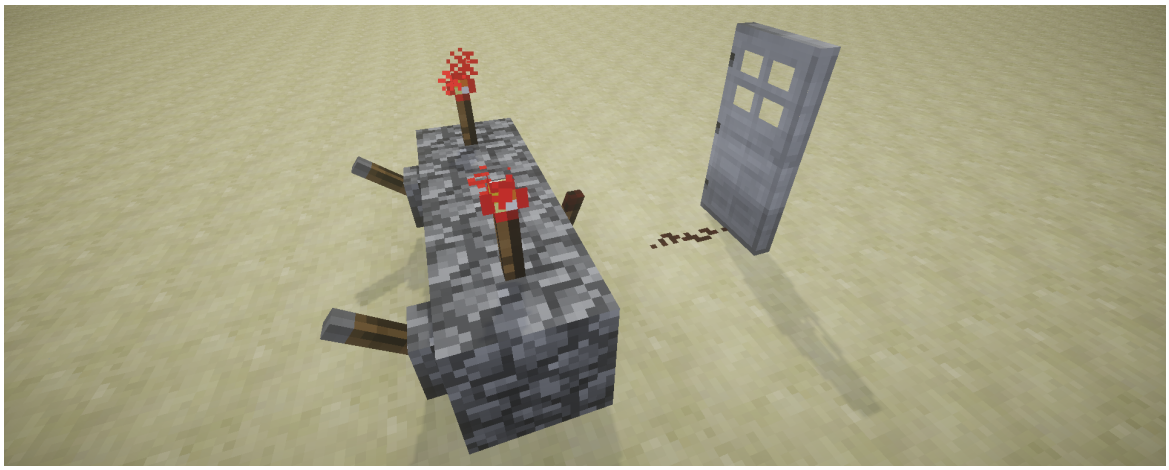
# Section 1: Logic Gates

Logic Gates are a fundamental building block for digital circuits. They perform *boolean* functions and usually have 2 inputs and 1 output

## AND

The AND gate performs the AND logic function which mathematically works like multiplication.

In Minecraft, your AND gate will take two redstone inputs which will either be ON or OFF. The output depends on the combination of inputs.



An example of an AND gate.

ON AND ON => ON
ON AND OFF => OFF
OFF AND ON => OFF
OFF AND OFF => OFF

Try these out on the above gate and see what happens. Remember that a lever is OFF if it is pointed upward.

## Truth Tables

Truth tables are an easy way for us to organize the various outputs of logic gates given different inputs.
For boolean logic, “ON” is replaced with “True” or “1” and “OFF” is replaced with “False” or “0”. For example, the AND Gate could look like this:

TRUE AND TRUE => TRUE
TRUE AND FALSE => FALSE
FALSE AND TRUE => FALSE

FALSE AND FALSE => FALSE

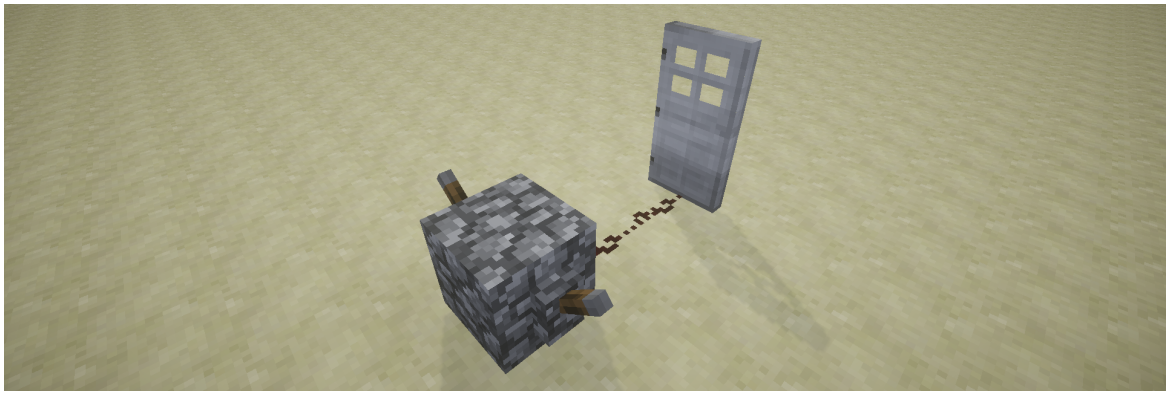
Which is the same as:
1 AND 1 => 1
1 AND 0 => 0
0 AND 1 => 0
0 AND 0 => 0

We can simplify this further to just a table:

| Input | Input | Output |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

# OR

The OR gate will only output OFF if both inputs are OFF. If either input is ON or if both are ON, the output will be on.

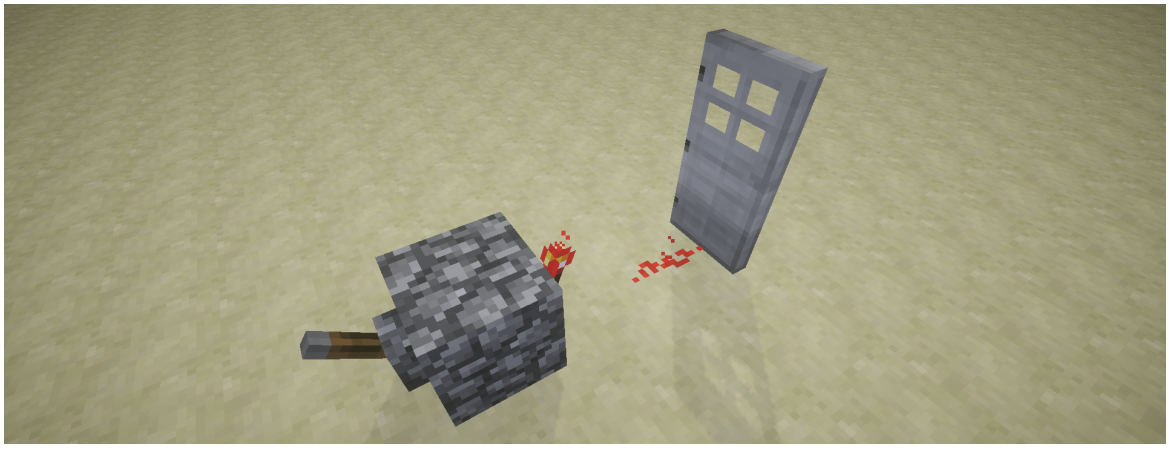


An example of an OR gate.

| Input | Input | Output |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

# NOT

A NOT gate only has one input and simply reverses that input. An ON input leads to an OFF output, and vice-versa.

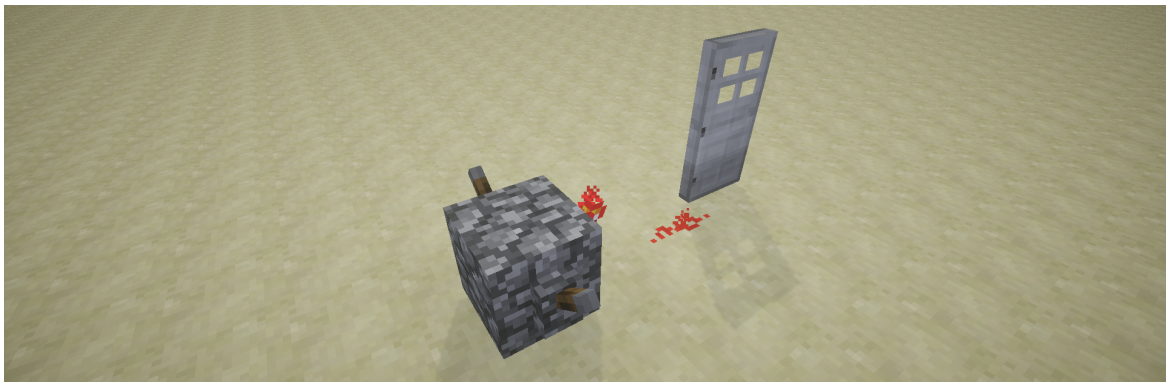


An example of a NOT gate.

| Input | Output |
|---|---|
| 1 | 1 |
| 1 | 0 |

# NOR

A NOR gate is just an OR gate with its outputs reversed. So a NOR gate will only output ON if both inputs are OFF. Otherwise the output is OFF.

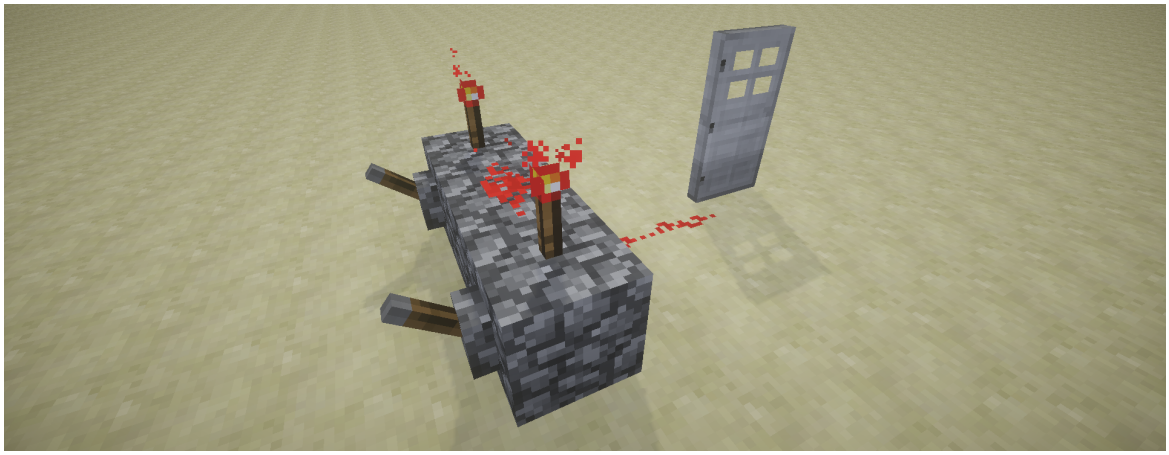


An example of a NOR gate.

| Input | Input | Output |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

## NAND

Just as a NOR gate has the opposite outputs of an OR gate, a NAND gate has the opposite outputs of an AND gate. It will output OFF only if both inputs are ON. If either input is OFF, the output will be ON.

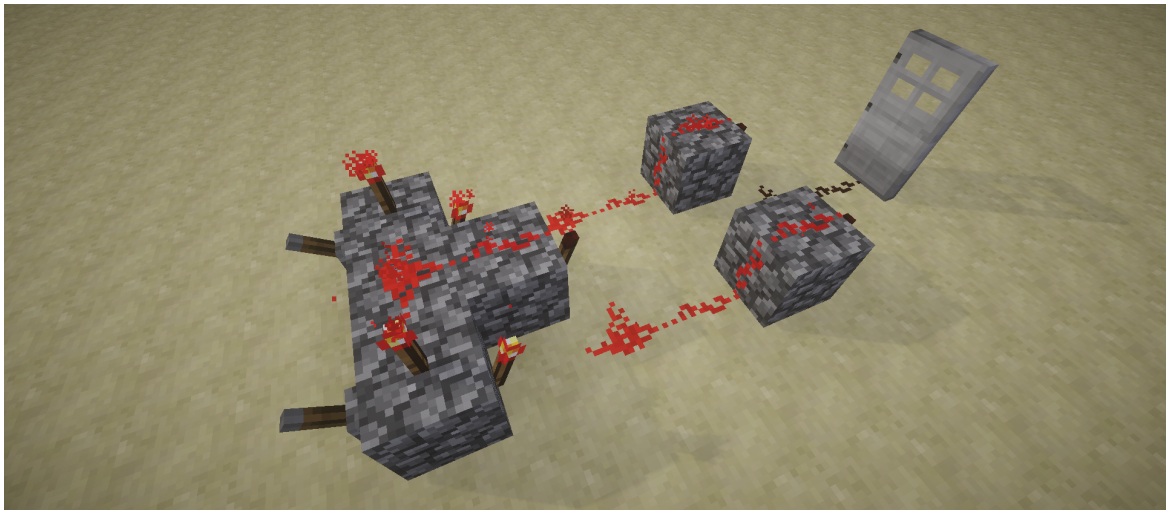


An example of a NAND gate.

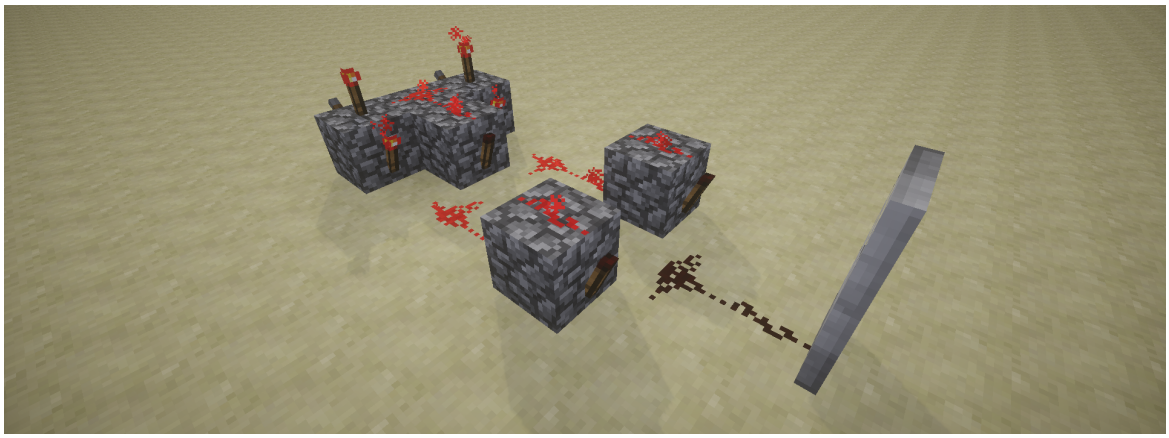
| Input | Input | Output |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 1 |

## XOR

An XOR (ex-or) gate is also called an “exclusive OR” gate. It will only output ON if either lever is ON. If both levers are either off or on, it will output OFF.



An example of an XOR gate.



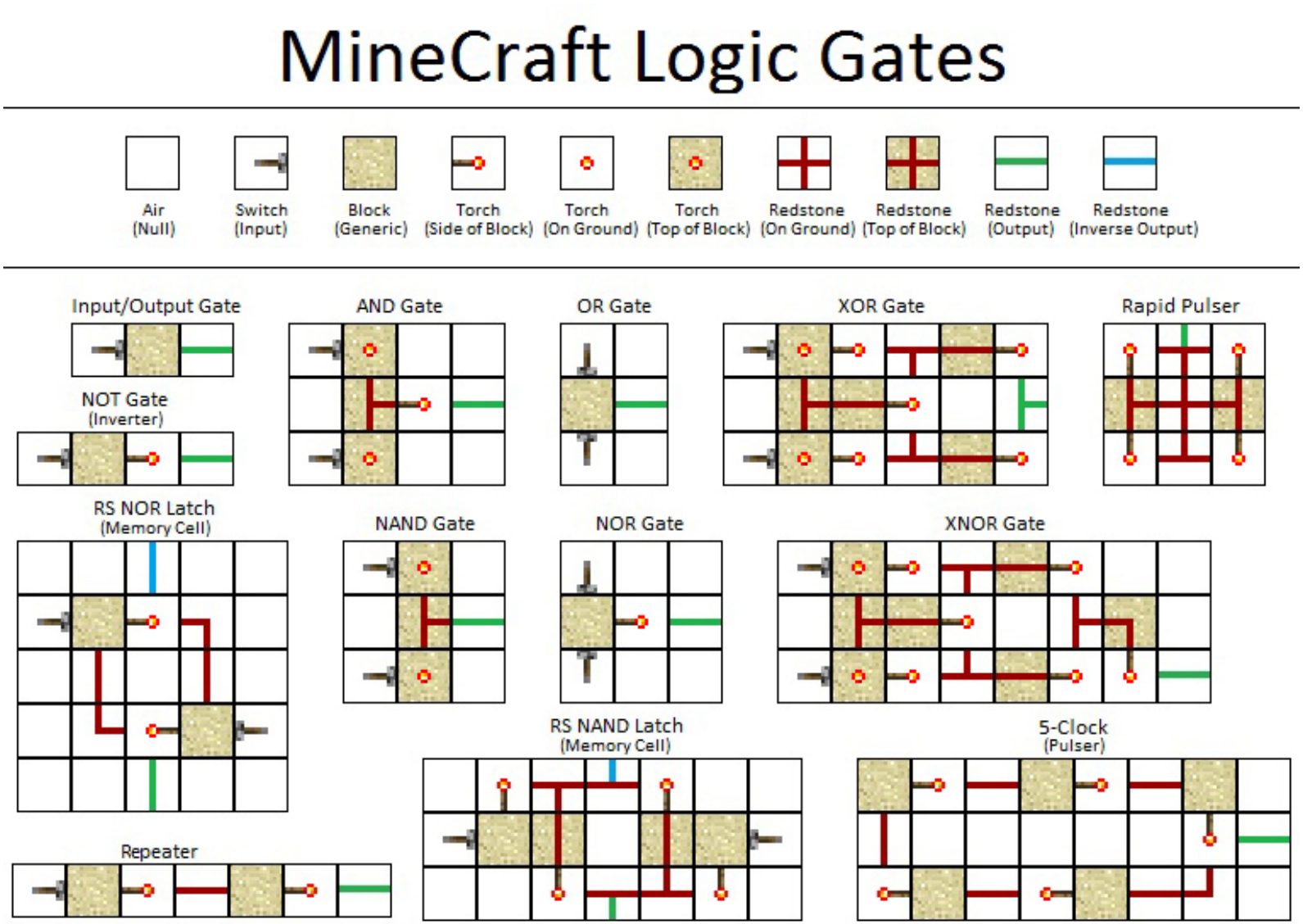
A different perspective of the same gate.

| Input | Input | Output |
|---|---|---|
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

# Reference

This diagram has most of the logic gates that we've gone over, as well as some more that you may find useful.

A diagram with redstone gate examples.

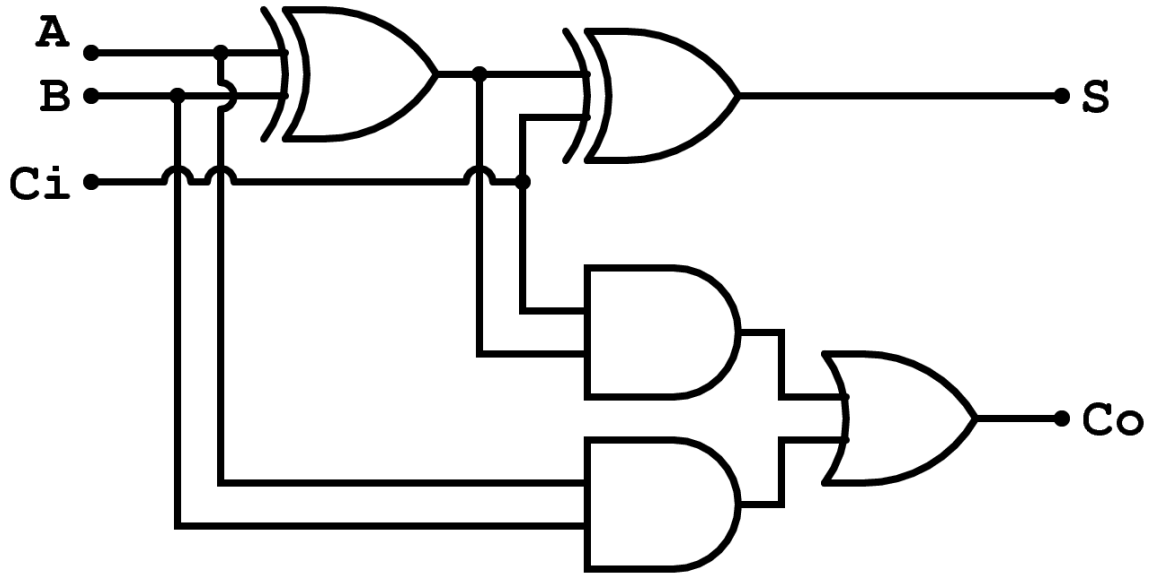


# Section 2: Adder and ComputerCraft

## Redstone Adder Circuit

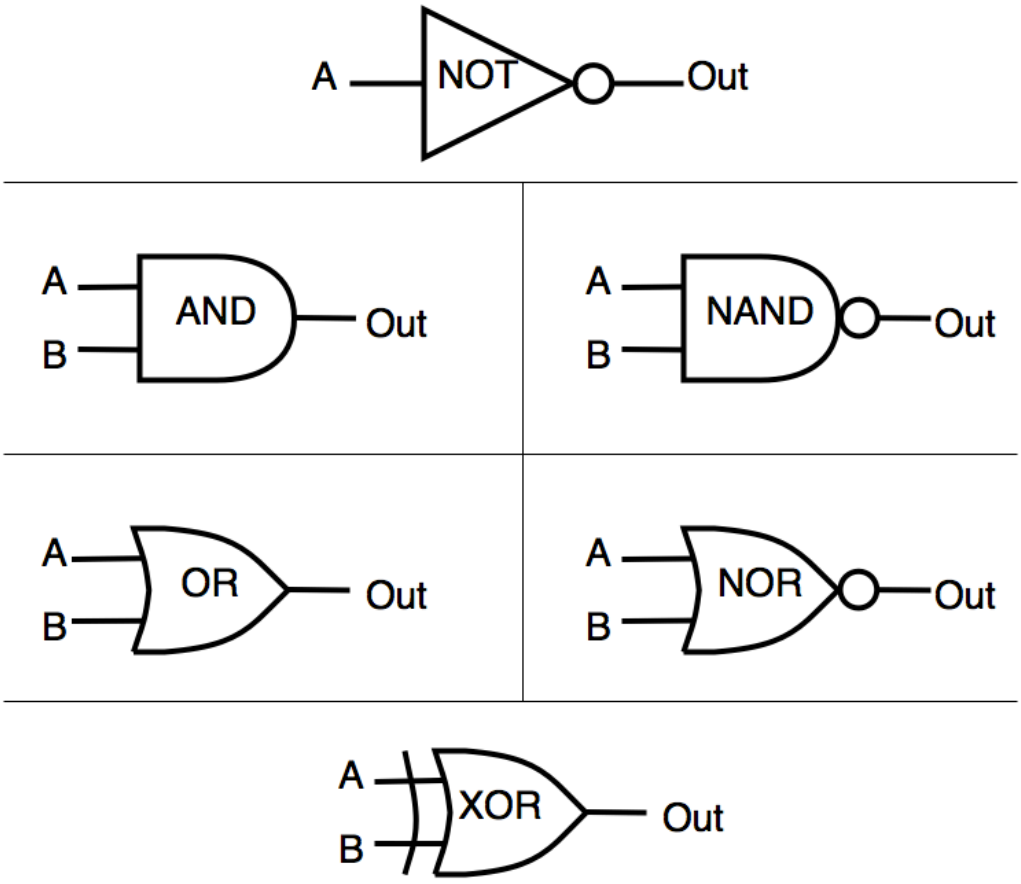
An *adder* uses logic gates to perform addition. There are three inputs to the whole circuit: A, B, and carry. The carry is the result from the previous addition (think of how you carry a 1 when you add large numbers). The two outputs are the result and the carry for the next addition.

This diagram shows the overall structure of an adder.




A diagram fo an adder.

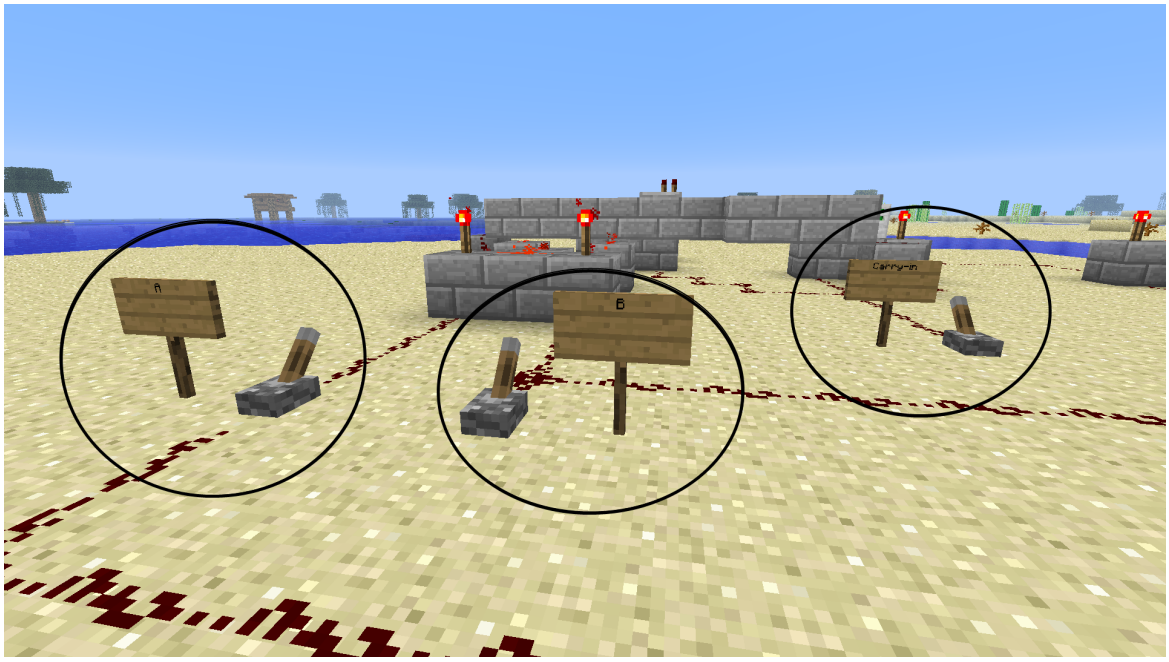
Look at the logic gates key so you can begin to understand how the adder works.




A key for the gate symbols.

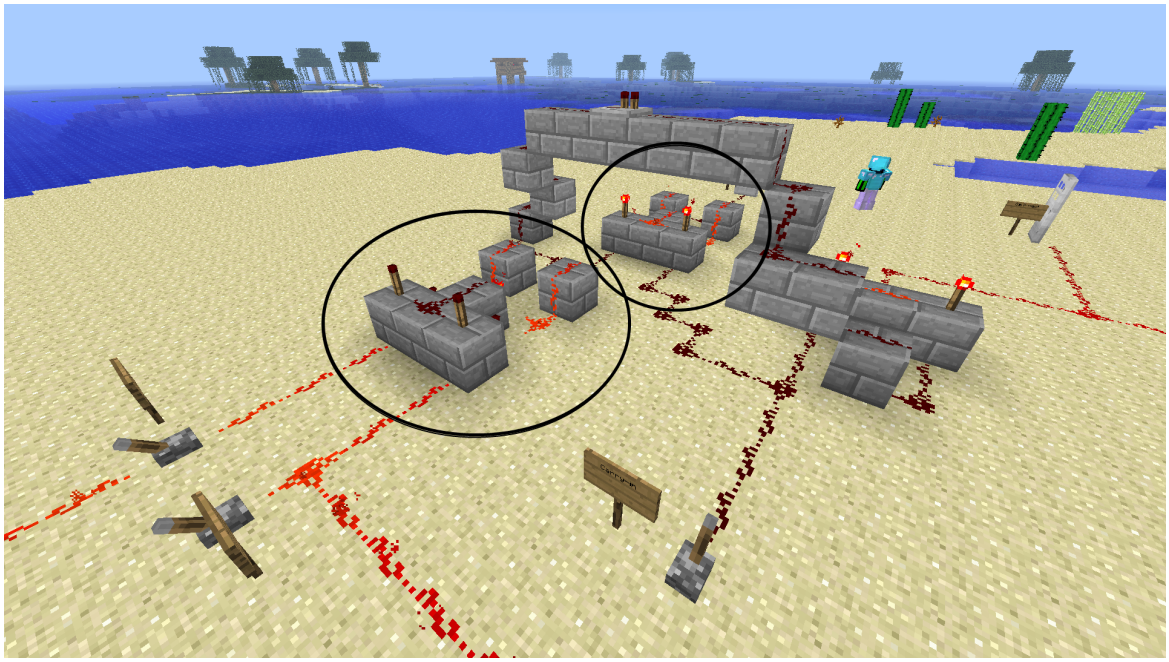
Now, we're going to build an adder using redstone! Follow along with the steps as best you can. This is a complex circuit so be sure to ask questions. The doors at the end represent the output. The door to the left represents the normal output, and the door to the right represents the carry. An open door is a 1 and a closed door is a 0.

There are three inputs, so be sure to put signs to label each one. Levers are best since we can switch them between ON and OFF easily.



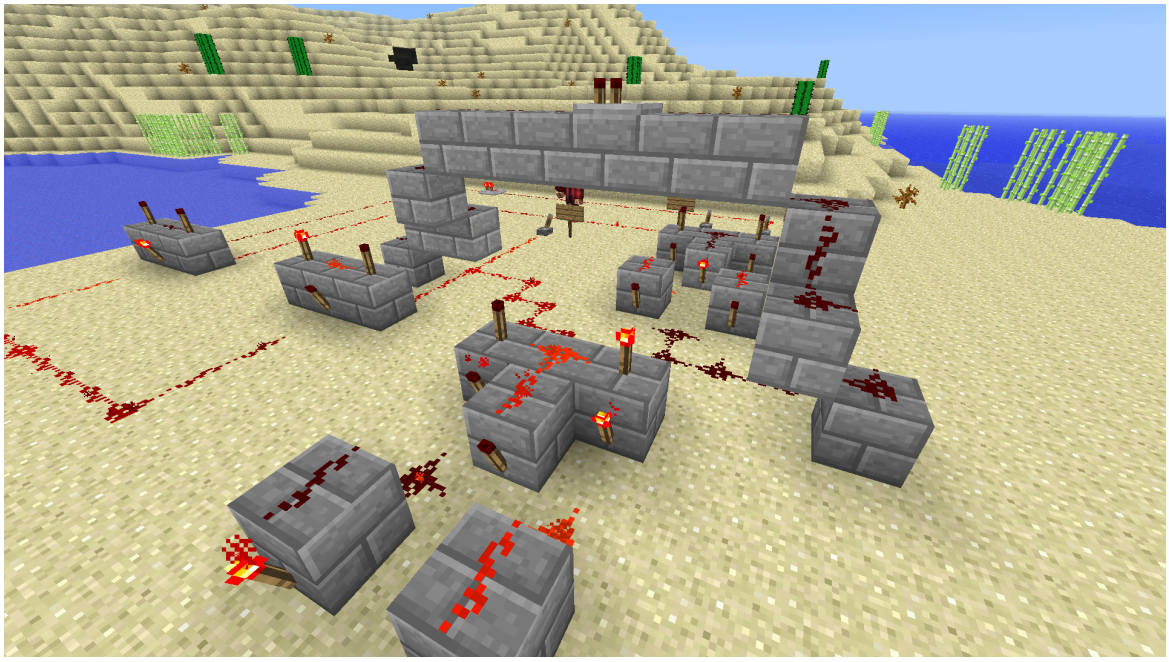

The three input levers.

Next, build two XOR gates. The first one (on the left; we'll call it XOR gate 1) takes input from A and B. The second one (XOR gate 2) takes input from the output of the first gate and the carry.



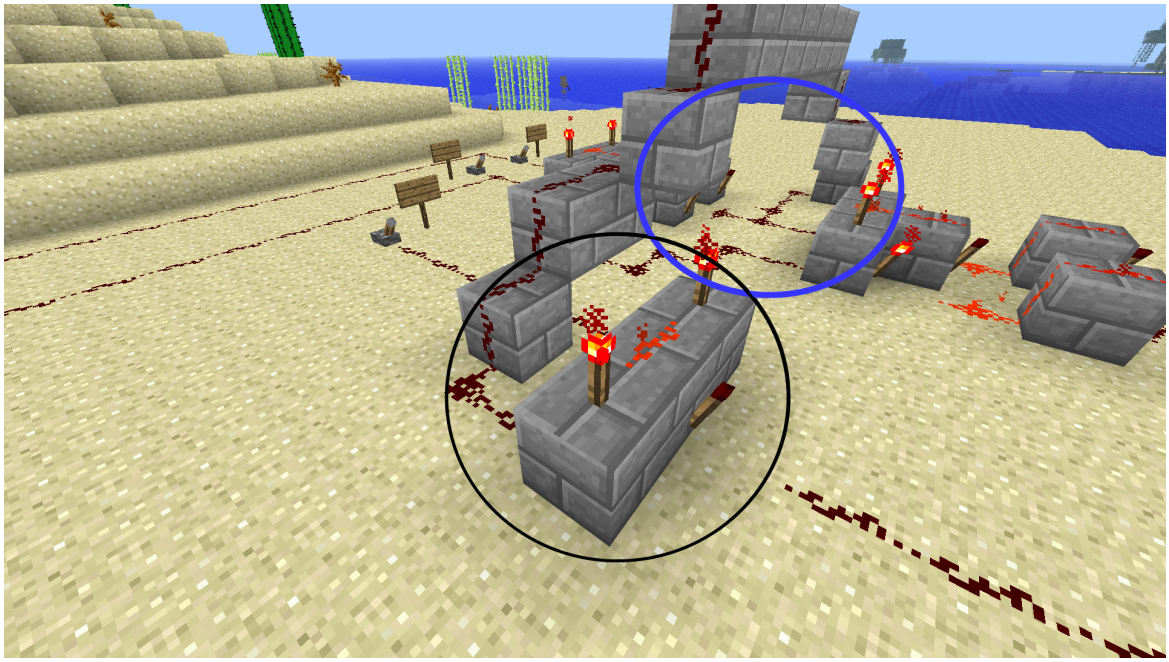
The inital XOR gates.

A second perspective of the XOR gates.



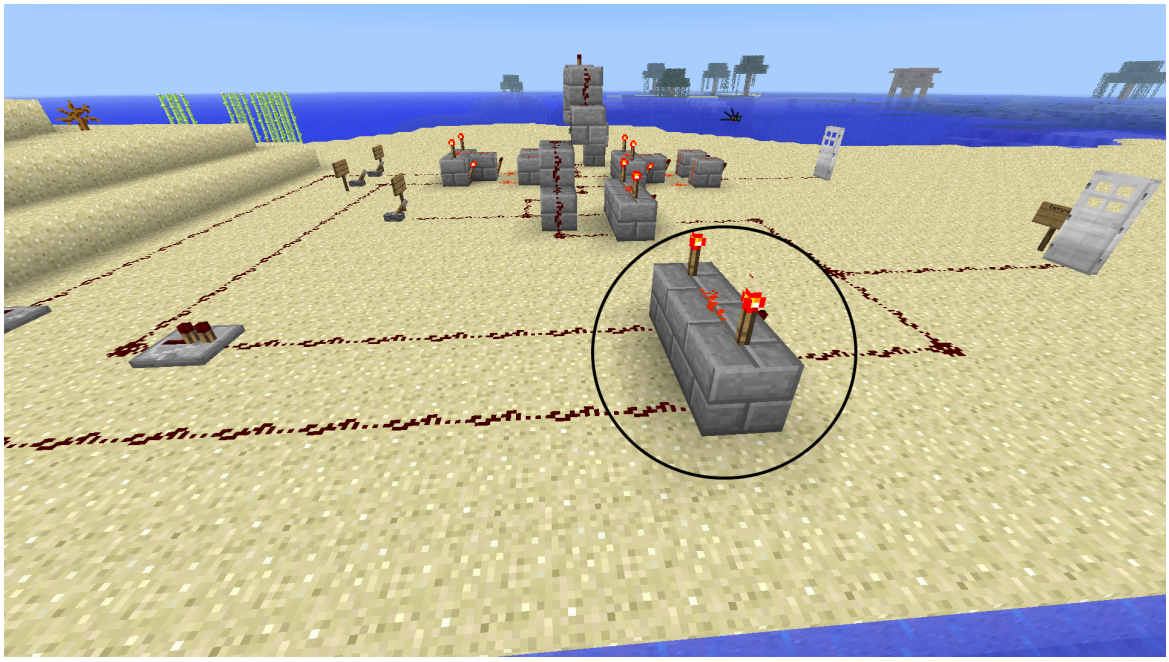
A backwards view of the XOR gates.

This AND gate takes input from the carry and the output of XOR gate 1. The blue circle shows where the gate 1 output goes up and over to this AND.



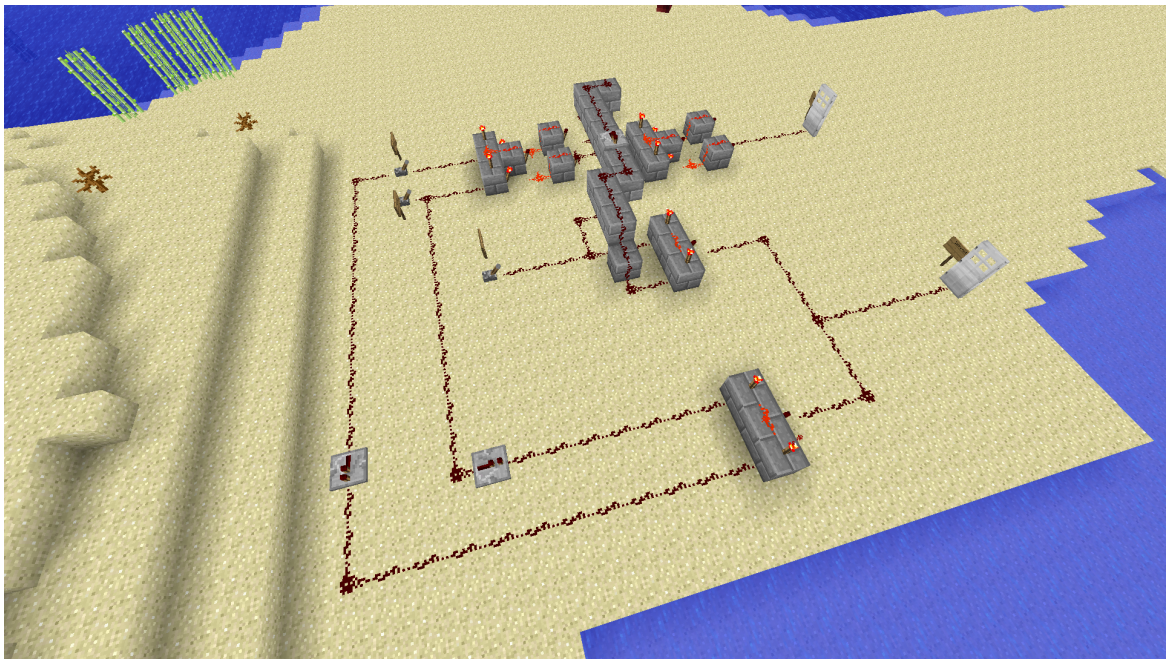
The first AND gate.

The second AND gate. This one takes input from the original A and B inputs. Notice how it and the output of the other AND gate combine and go to the door representing the carry.

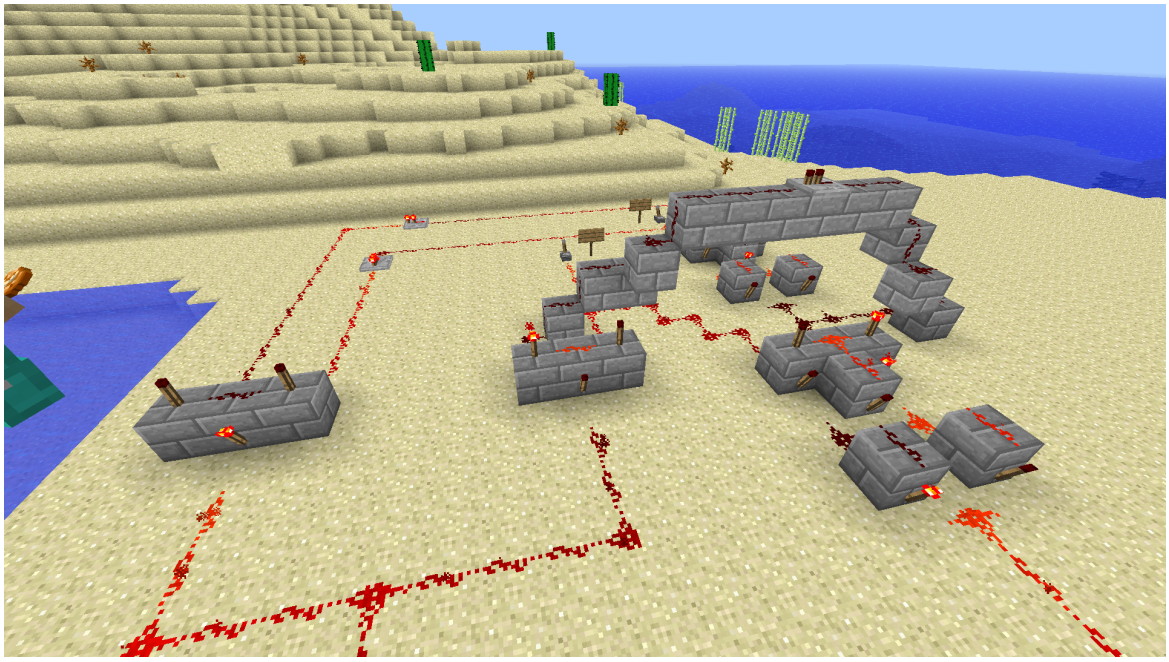


The second AND gate.

Two overviews of the entire adder from different perspectives.



An overview of the entire adder.



A second overview of the adder.

# ComputerCraft Basics

> ComputerCraft is a modification for Minecraft that's all about computer programming. It allows you to build in-game Computers and Turtles, and write programs for them using the Lua programming language. The addition of programming to Minecraft opens up a wide variety of new possibilities for automation and creativity. If you've never programmed before, it also serves as excellent way to learn a real world skill in a fun, familiar environment.

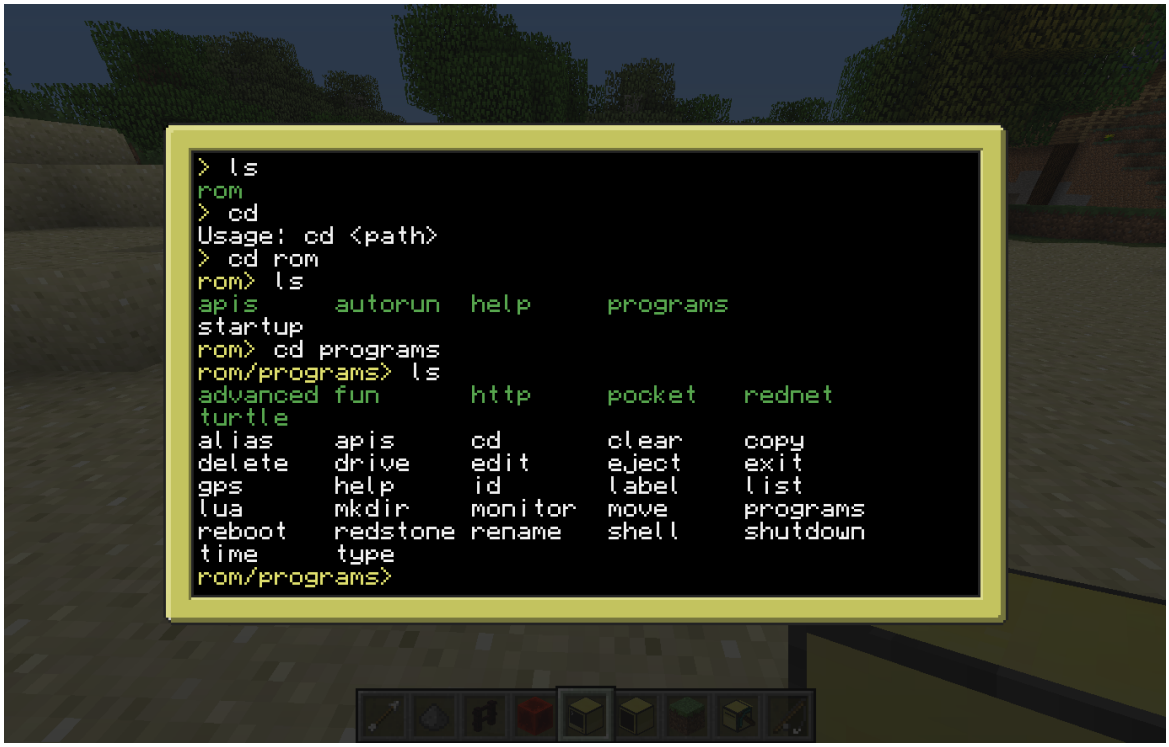
This will be your first taste of the ComputerCraft mod, and for some of you, your first time using a command line interface. With this knowledge, you can start using other command line software, and you'll get a solid foundation in simple programming skills.

## The Command Line

When you place a computer and right click on it, the first thing you see is this.


The beginning of your command line life.

This is a command line. Here, we can type word to run programs, and we can look inside folders just like in a graphical interface. Let's do that now.

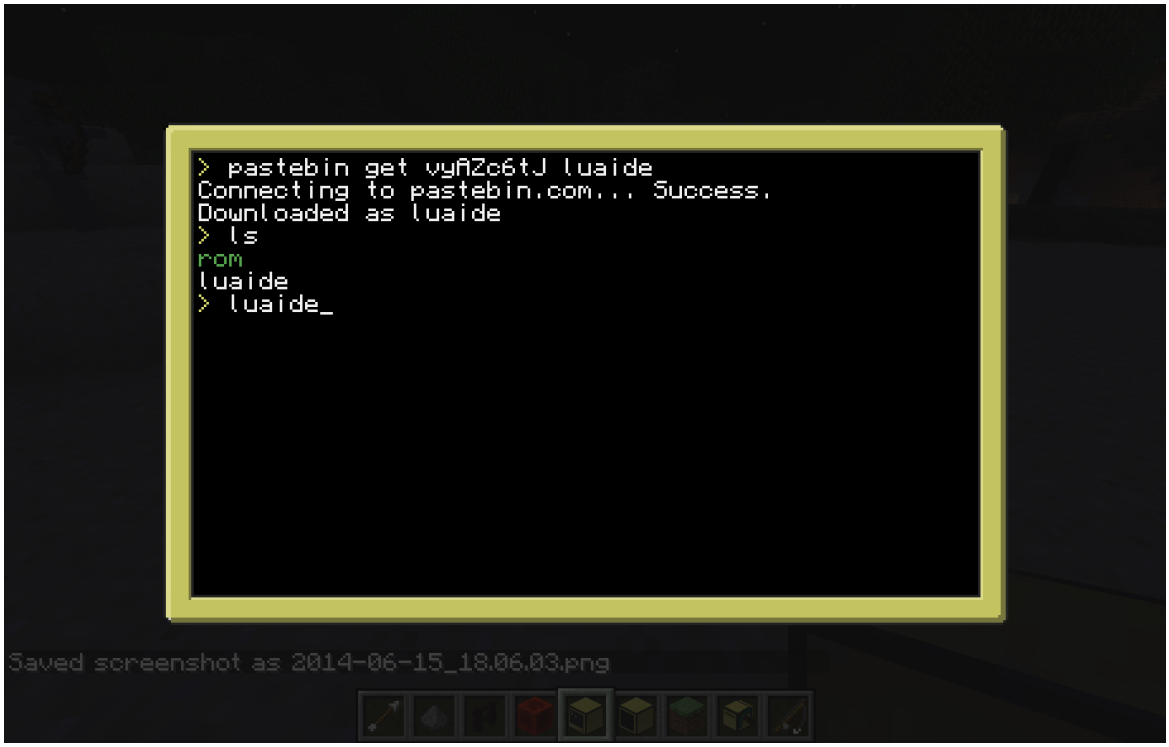



Some simple commands to run.

Up next we'll run the edit program and see what it look like.


This is the program you would usually use to edit your programs. Looks pretty spartan huh?

Let's download a better one. This program is called luaIDE, and it's a fancy code editor written just for ComputerCraft.



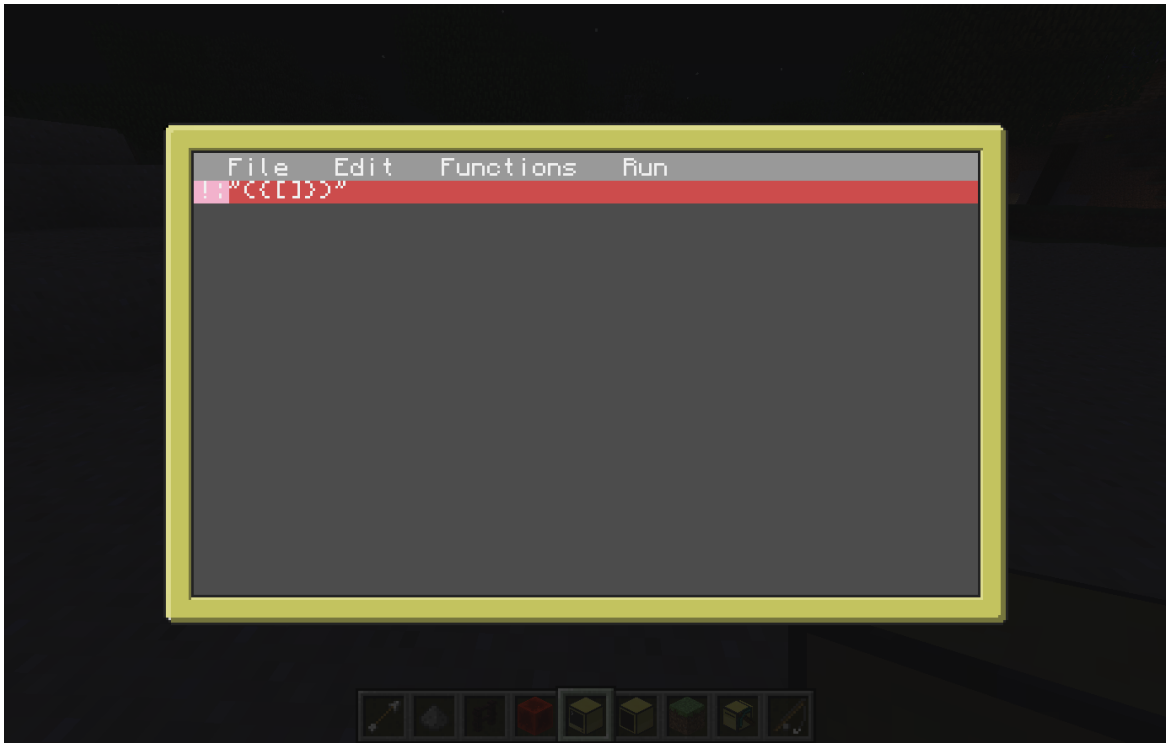


There we go, much better. If you're using an advanced computer, you can click on these buttons, otherwise just use the arrow keys to navigate.

Let's make a new file called `helloworld`



This editor will autocomplete brackets and quotes for you. Watch!



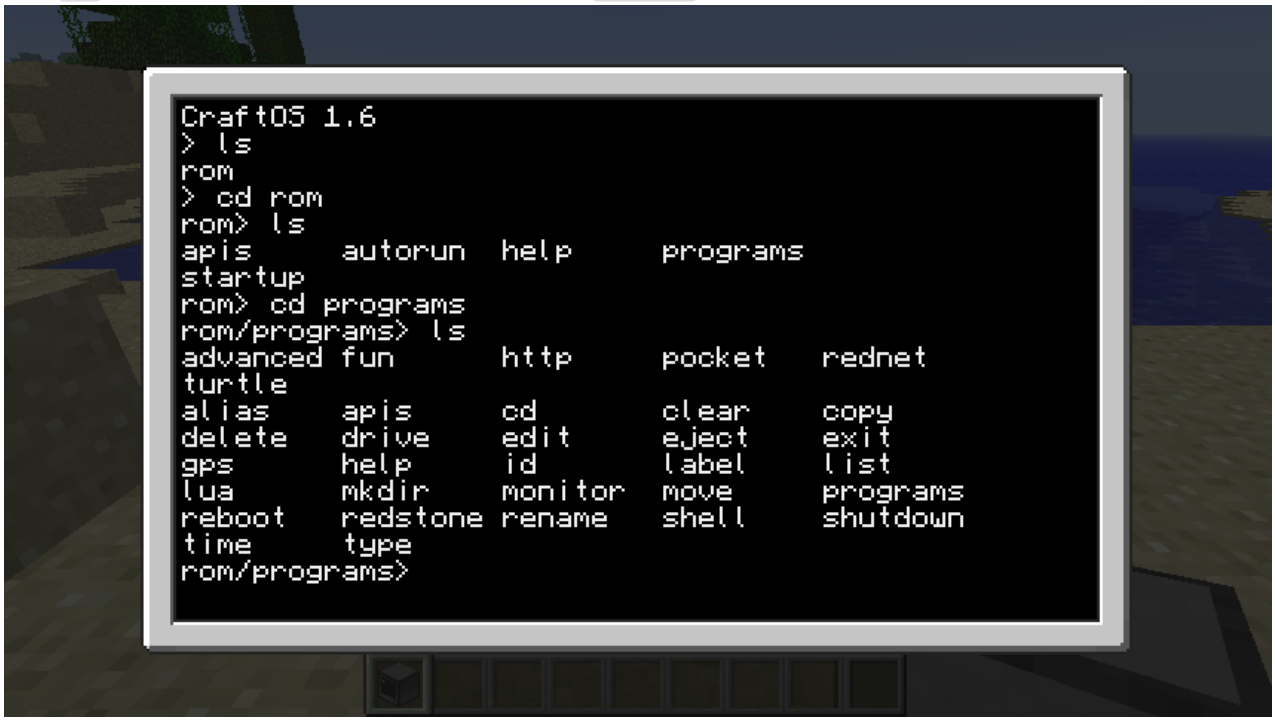

Now let's do some stuff!

## Getting familiar with ComputerCraft

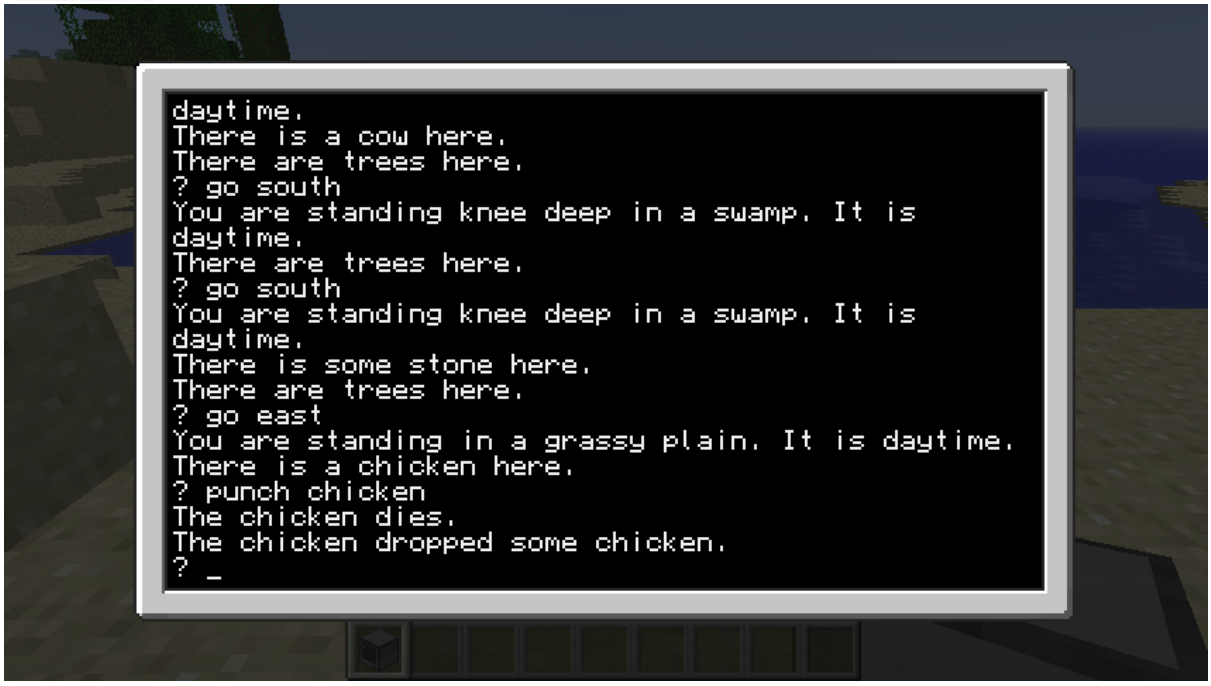
1. Open the ComputerCraft world
2. Open your inventory and search for `computer`
3. Place a computer on the ground and right-click on it

4. Navigate to programs
5. the `ls` command lists the contents of the current directory

6. the `cd` command changes directories, e.g. `cd rom`




7. Play text adventure Minecraft inside a ComputerCraft computer
8. type `adventure`
9. some of the commands available in the Adventure program:
    1. `punch`
    2. `take` or `grab`
    3. `craft` or `make`
    4. `go`
    5. `eat`
    6. `inventory`



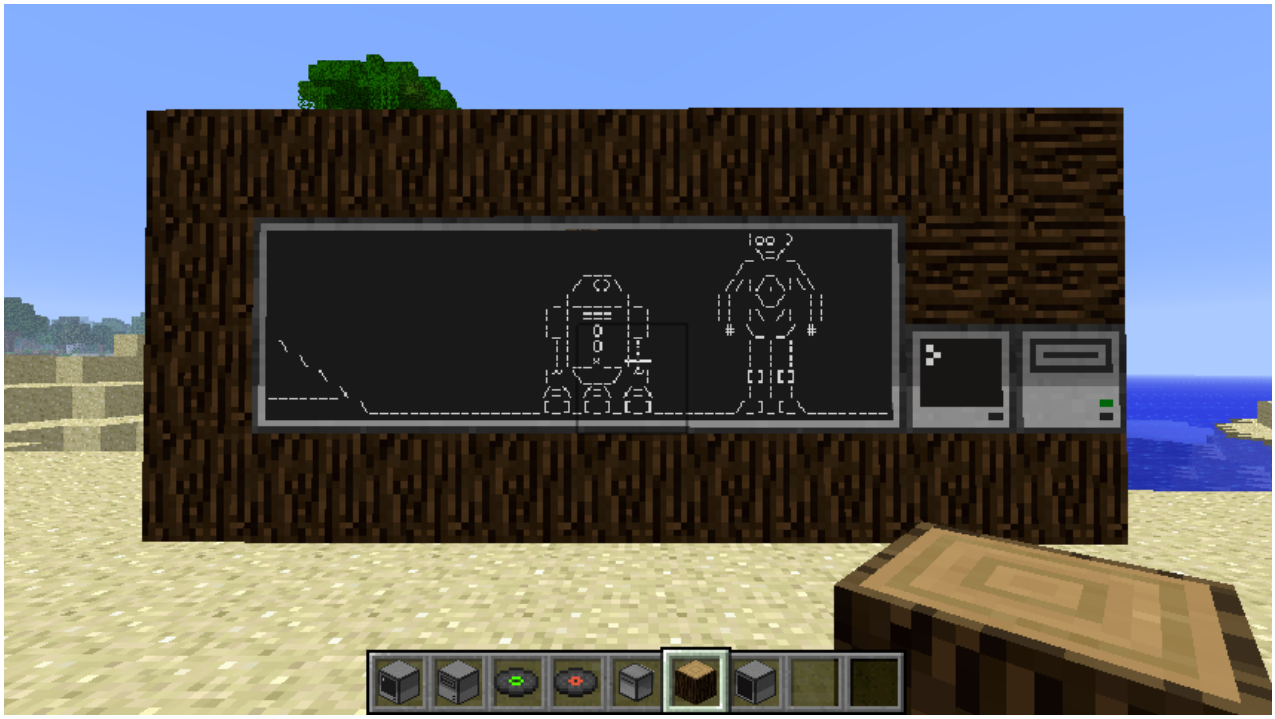

In case you haven't noticed by now, Adventure is really just text-based Minecraft. You're

playing Minecraft on a computer inside Minecraft.

10. Add disk drive
11. Open your inventory and search for `disk drive`
12. Place the disk drive next to the computer

13. Add a music disk to the disk drive
14. Play the music with the `dj` program
15. Create a monitor
16. Open your inventory and search for `monitor`
17. Place 12 monitors in a 6 wide by 2 high pattern to create a giant widescreen monitor
18. Place a computer next to the monitor
19. Place a disk drive next to the computer
20. Watch a movie on the monitor
21. Open your inventory and search for `disk`
22. Find a disk labeled `alongtimeago` and place it into the disk drive


23. Right click on the computer and run the `alongtimeago` program
    1. To run a program on the disk, specify the full path to the program like this: `disk/alongtimeago`
    2. To run the program on the monitor, specify the monitor first, so this: `monitor [top|bottom|left|right|front|back] disk/alongtimeago` The syntax `[top|bottom|left|right|front|back]` means pick which side your monitor is on and only type that direction, of those six shown. So you command would be something like `monitor left disk/alongtimeago`.
24. Sit back and enjoy the show



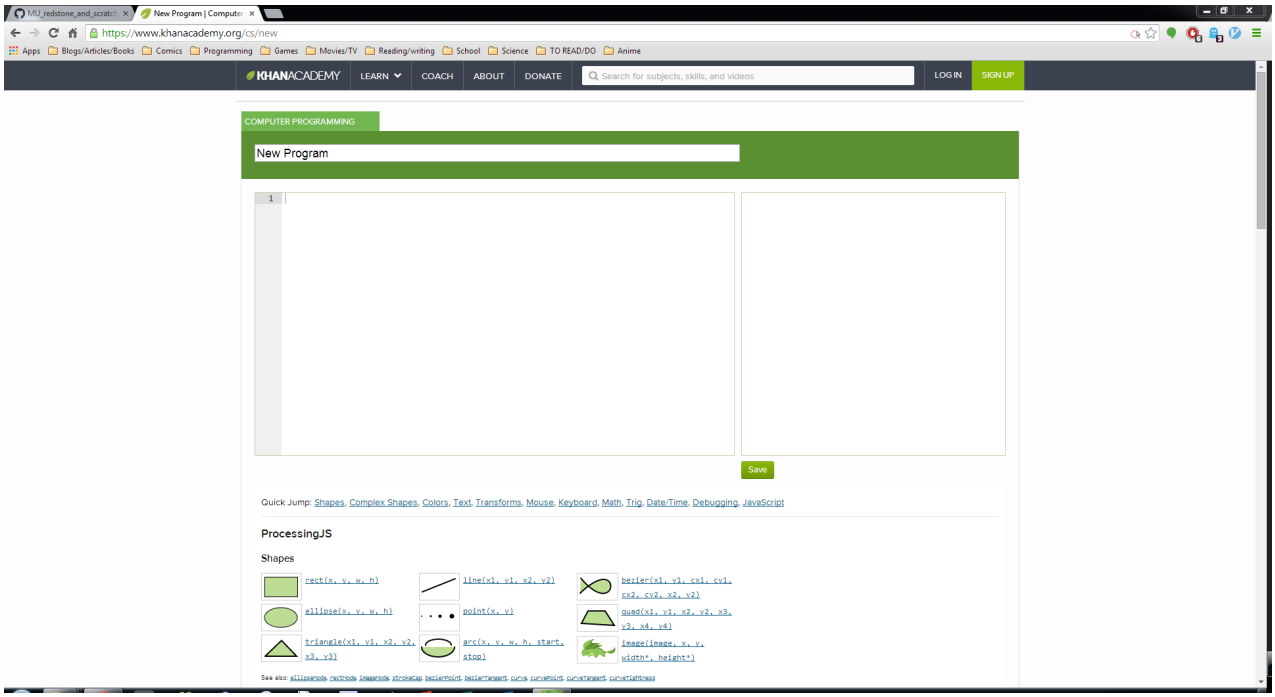
25. To quit any running program, hold down `ctrl + r`

Now explore the other programs available on ComputerCraft computers.

# Section 3: Khan Academy ProcessingJS

Now we're going to learnw some basic programming concepts using ProcessingJS and Khan Academy. If you haven't heard of Khan Academy, it is a website that has instructional videos for math, history, and many other subjects. Last year, they introduced a programming platform that's based around shapes, colors, and animation.

1. Open up khanacademy.org and click on the "Learn" tab at the top left. Then hover over the "Computing" tab and click on "Computer Programming". On the next page, click "Create Program" over on the left side of the screen.
2. This is the IDE (or Integrated Development Environment) for Khan Academy's programming tutorials. Code that is typed into the left box will be executed real-time in the right box; any changes you make will immediately be reflected on the right side.




3. If you scroll down the page, you'll see the references section. This has a listing of many of the ProcessingJS commands and their uses. Double-clicking on a command will bring up an example in a new window, so you can understand how each command works if you're ever confused.

## Drawings

To start off, we're going to make some simple printing and drawing as visuals help teach coding principles before we move onto turtles. Each line of code that we type is called a *statement* and each statement must end in a semicolon.

1. To print text to the screen, call the `text(text, x, y)` function. Be sure to put your text in double quotation marks; characters inside double quotes are called a *string*. The two numbers

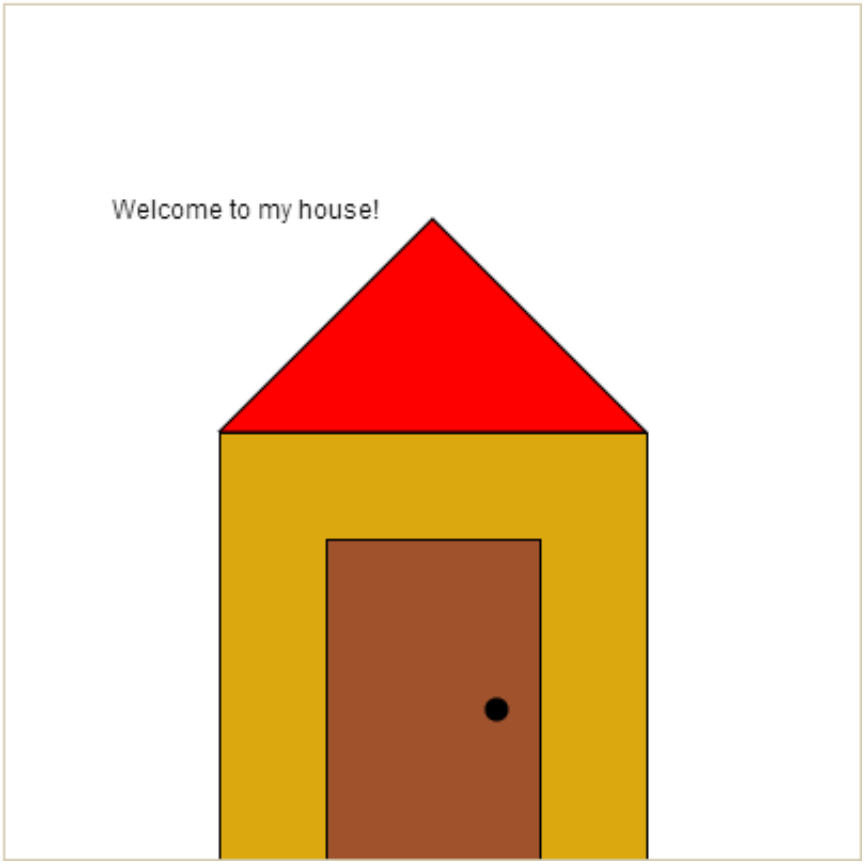
determine the `x` and `y` coordinates of the text (specifically, its bottom-left corner). Before we print text, however, we need to set the color of the text (it is white by default). The `fill(r, g, b)` command sets the color based on the red, green, and blue values given. From now on, we'll be referring to these commands as *functions*.

```
fill(0, 0, 0);
text("Hello world!", 0, 0); //Print text
text("Hi there", 100, 100);
text("This is below the other ones", 100, 200);
```

2. Any line prefaced by `//` is a *comment*. It will not be executed by the computer and exists only for the benefit of humans reading the program. Comment any part of a program that could be confusing so that other coders will know what the code does.

3. Type `rect(0, 0, 100, 100);` into the left box. `rect` is a command that draws a rectangle and the numbers (called *arguments*) are data given to the command The first two numbers are the `x` and `y` coordinates of the top left corner of the rectangle, just like the `text()` function. The third and fourth numbers are the `width` and `height` of the rectangle. You can hover over each of the numbers and drag the slider back and forth to see how the rectangle changes as each of the numbers change.

4. Try out some of the other drawing commands such as `ellipse`, `line`, and `quad`. Look at the references section to know how many arguments to give each function. In addition, look at the house below and try to re-create it using your drawing functions. Be sure to pay attention to the order of functions as that will determine which shapes are on "top" of the drawing.




# Variables

Variables are containers that can hold different values in different parts of the program. They may hold a character, a number, or a string (which is a collection of characters such as “hello”). Variables are very useful when dealing with loops and user input, as they allow you to call a function without actually specifying the exact number for the argument.

1. When using a variable, we must first *declare* the variable. Type `var x = 10;` to create a new variable `x` and set it equal to 10. Then, run `rect(100, 100, 10, 10);` and `rect(200, 100, x, x);`. Notice that the rectangles are the same size since `x` is equal to 10.

2. Now try setting `x` to different numbers using just `x = 25`. The keyword `var` is only necessary when you first declare a variable. An example program is below.

```
var x = 10;
rect(100, 100, x, x);
x = 20;
rect(100, 100, x, x);
x = x + 10;
rect(100, 100, x, x);
```

# Conditionals

Conditionals allow you to control the flow of your program. The `if` and `else` statements are the main two that we will look at. Conditionals operate on the same boolean logic that we learned on day 1.

1. Type the following program into your window. The blocks of code contained by braces `{}` run only if their corresponding `if` statement evaluates to true.

```
if (10 < 20)
{
    fill(255, 0, 0);
    rect(150, 250, 100, 150);
}

if (20 < 20)
{
    fill(0, 255, 0);
    rect(150, 250, 100, 150);
}

if (30 < 20)
{
    fill(0, 0, 255);
    rect(150, 250, 100, 150);
}
```

2. `else` statements must follow `if` statements, and run when the `if` statement evaluates to false.

```
if (8 >= 8)
{
    fill(255, 0, 0);
}
else
{
    fill(255, 0, 0);
}

rect(150, 250, 100, 150);
```

# Loops

Loops provide a way to execute the same commands multiple times. Each execution of the block (called an *iteration* when dealing with loops) depends on the result of a *boolean expression*, just like with `if` statements.

## WHILE loops

The body of `while` loops execute when the boolean of the `white` evaluates to `true`. The syntax for a while loop is below. Notice the form of the boolean expression after the keyword `while`.

```
while (x < 10)
{
    //Code body here
}
```

Execute the following program. Notice that we make a change to the value of `x`.

```
var x = 0;
while (x < 200)
{
    ellipse(x, 100, 50, 50);
    x = x + 50;
}
```

What happens if we run the same program without changing the value of `x`? Comment out the line `x = x + 50` by putting `//` in front of it. Khan Academy will give you a popup saying your program took too long to execute. Since the value of `x` never changes, the expression `x < 200` is always true and so the `while` loop will repeat forever. A loop that never ends is called an *infinite loop* and they should generally be avoided if possible.

## FOR loops

`for` loops are more structured that `while` loops. To write a `for` loop, you specify a starting point, an ending point, and an action to be performed every iteration. The following code block shows the basic syntax of a `for` loop.

```
for (var i = 1; i < 10; i = i + 1) //Notice we actually declare the variable i insid
{
    //Code goes here
}
```

`var i = 1;` is executed when the program reaches the `for` loop for the first time. `i < 10;` is the condition that is checked at the end of each run, similar to the expression for the `while` loop. The

last statement `i = i + 1` is executed at the end of iteration of the loop. It serves the same purpose as our `x = x + 50` in the `while` loop from earlier.

Now, run the following program. Change some of the variables in the loop declaration to see what they do.

```
noFill(); //This makes shapes transparent except for the edges
for (var i = 10; i < 200; i = i + 20)
{
    ellipse(200, 200, i, i);
}

for (var i = 0; i < 400; i = i + 40)
{
    rect(i, i, 20, 20);
}
```

## Bouncing Ball Program

This program is a bouncing ball program. Go ahead and execute and watch the animation. Take a look at the code and read the comments. We'll be going over this on the projector so if you're confused feel free to ask any questions.

```
var x = 150; //X position of the circle
var y = 25; //Y position of the circle

var xSpeed = 5;
var ySpeed = 5;

//Here we are creating our own function. Any function called "draw"
//is automatically executed by the program over and over, so we don't
//need to have any real loop.
var draw = function() {
    background(255, 255, 255);

    //Draw the circle
    fill(255, 0, 0);
    ellipse(x, y, 50, 50);

    //If the circle hits the left or right wall, reverse its movement
    //in the X direction
    if (x < 25 || x > 375)
    {
        xSpeed = -xSpeed;
    }

    //If the circle hits the top or bottom wall, reverse its movement
    //in the Y direction
    if (y < 25 || y > 375)
    {
        ySpeed = -ySpeed;
    }

    //Move the circle before the next iteration
    x = x + xSpeed;
    y = y + ySpeed;
};
```

# Section 4: Advanced ComputerCraft

## Tunneling

This program will require a mining turtle (a turtle with a diamond pickaxe equipped).

The “tunnel” program has the turtle dig a tunnel that is two (2) blocks high, three (3) blocks wide and *distance* blocks long (the turtle will stop mining when its inventory is full, it runs out of fuel, or it hits bedrock). It has the following format:

`tunnel <distance>`

Try typing `tunnel 5`

Note that the turtle will automatically pick up what it mines and it fills in gaps in the floor.

WARNING: It will not block water or LAVA flow.

## Excavate

The “excavate” program makes the turtle dig a hole that is *distance* x *distance* until it hits bedrock.

It uses the format: `excavate <distance>`

Try typing `excavate 5` but don’t let it dig too deep!

Notice that the Turtle digs *distance* blocks forward and then *distance* blocks to the right. Keep this in mind in case there’s something important in close proximity.

Like the tunnel program, the turtle will stop when its inventory is full, it runs out of fuel, or it hits bedrock.

Tip: Try putting a chest directly behind the turtle before you start the excavation program.

## Writing a simple turtle program (with arguments!)

### Communication between computers/turtles

As I’m sure you’ve realized, programming on a turtle can be a little tedious, especially with longer, more complex programs. For convenience, people have developed better environments for programming know as IDE’s (Integrated Development Environment). In Minecraft, we have a luaIDE on the “computer” and we can transfer programs we write on the computer to turtles using “floppy disks.”

1. Create a rough version of the program.
    1. Right click the turtle to boot it up.
    2. Type `label set mover` where *mover* is the “name” of the turtle.
        1. This will make the programs on this turtle stick around when you break it and place it somewhere else.
    3. For easier editing, run the luaide program.
    4. Create a new program called “move”
        1. Inside the file you just created, try writing a program to make the turtle move forward.
        2. If you need help you can look in a couple places:
            1. There is a complete list of every command you can give a turtle here on the ComputerCraft wiki.
            2. There is also a program called `go` that is located at `\programs\go` on your turtle.
                1. Press `Ctrl+O` to open files in LuaIDE.
            3. Looking at this program and the wiki will help you understand how the turtle commands work.
2. Basics of LUA

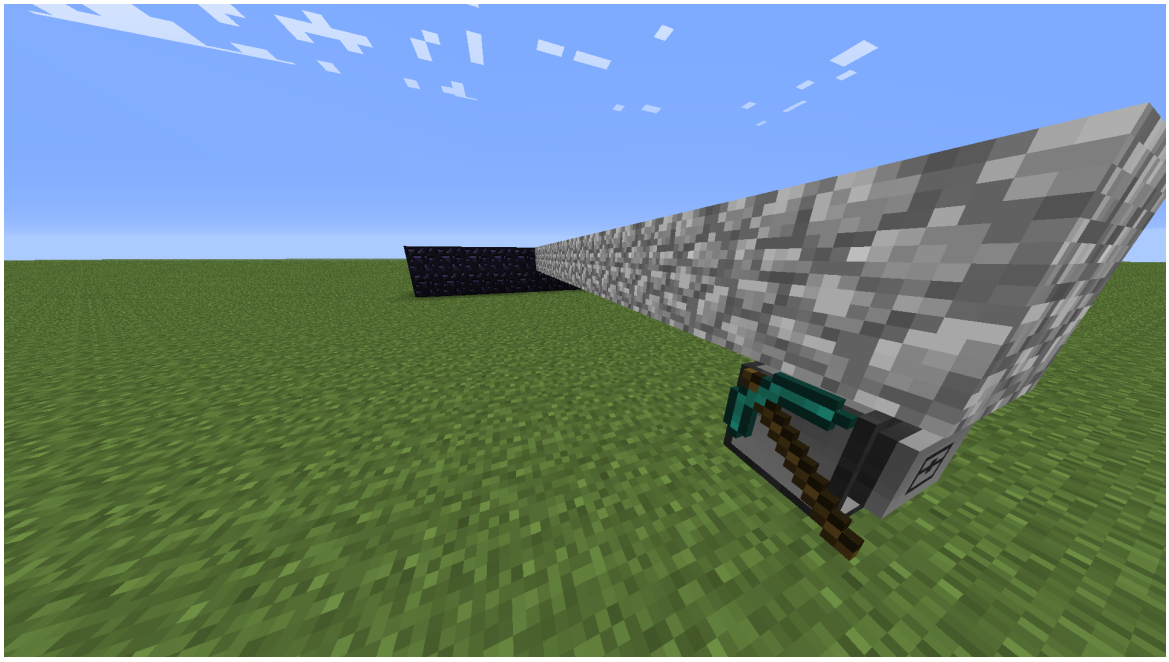
    A standard block of code in lua looks like this.

```
turtle.refuel()
while turtle.detectDown() do
  turtle.dig()
  turtle.digDown()
  turtle.down()
  turtle.dig()
  turtle.forward()
  turtle.turnLeft()
end
```

# Cobble Harvesting

Remember the cobblestone generator? What if we could make a turtle automatically harvest that cobblestone? Let’s do it!

First, underneath the row of cobblestone, put a mining turtle.



What you should see.

Since we'll be writing a program, let's name the turtle so we don't lose it.

```
label set SoMuchCobble
```

Let's start coding!

```
edit cobbling
```

This will create a new program called "cobbling" that will be stored on the turtle's internal memory. Program names **are case-sensitive.**

Now let's consider the problem. What do we want the turtle to do **exactly**? Let's break it down.

1. Detect if there is a block above it
2. If there is a block above it, mine it
3. Otherwise, wait for a block to appear above it.
4. repeat

Breaking problems down into simple steps is a large part of a programmer's job. Computers themselves are not smart…yet. So now we have a general outline for the program, but it's in English. Computers don't speak English. Let's translate it!

1. `turtle.detectUp()`
2. `if turtle.detectUp()` then `turtle.digUp()`
3. `else sleep(1)`
4. loop

Explanation:

`turtle.detectUp()` is a function that has the turtle check if there is a block directly above it.
`turtle.digUp()` is a function that makes the turtle dig or mine the block above it.
`sleep(1)` is a function that makes the turtle wait for one (1) second.

For this program, we'll be using a `while` loop.

For detailed descriptions check out the turtle API.

So our program will look like this:

```
while turtle.getItemCount(16) < 64 do
  if turtle.detectUp() then
  turtle.digUp()
  sleep(1)
  else sleep(1)
  end
end
```

It should look like this:


What you should see.

Program Walkthrough:

1. `while turtle.getItemCount(16) < 64 do` Creates what's known as a "while" loop. This has the turtle do whatever the program says as long as the condition `turtle.getItemCount(16) < 64` is "true".
2. `turtle.getItemCount(16) < 64` Checks if the number items in the last inventory slot of the turtle is less than 64. Basically it checks to make sure the turtle isn't out of inventory space and returns a "true" or "false".
3. `if turtle.detectUp() then` has the turtle check if there is a block above it and if it's "true" it

continues with the `then` statement. If there is not a block above it, the check will return “false” and the turtle will skip to the `else` part of the program.

4. `turtle.digUp() sleep(1)` has the turtle break the block above it and then wait for one (1) second for the cobble generator to create another block.
5. `else sleep(1)` Only happens if step 3 returned “false” in which case the turtle waits one (1) second for the cobble generator to create another block.
6. The two end statements simply close the loops.

# Room building

Let’s write a program to build a simple square room: `luaide room`

Step one is to check your arguments to make sure they are what you need them to be.

```
local tArgs = {...}

if #tArgs ~= 3 then
  print ("Usage: room <l> <w> <h>")
end
for i=1,3 do
  if tonumber(tArgs[i]) < 1 then
    print("Usage: room <l> <w> <h>")
  end
end
```

This block of code takes agruments and check to make sure there are 3 of them, and that all are numbers greater than 1. If they aren’t, it prints a line that tells the user how to run the program.

Now, we take the three arguments and assign them to the correct variables.

```
length = tonumber(tArgs[1])
width = tonumber(tArgs[2])
height = tonumber(tArgs[3])
```

And we refuel the turtle.

```
turtle.select(1)
turtle.refuel()

if turtle.getFuelLevel() < 50 then
  print("Fuel level too low.")
  return false
end
```

Here, we select the first slot, and load the turtle with fuel from that slot.

This program is going to be in a few different sections, or functions.

```
function buildRow(rowLen)
  for i = 1, rowLen do
    findItems()
    turtle.placeDown()
    turtle.forward()
  end
end
```

This first function builds one single row of blocks, using a for loop to stop it when it reaches the desired length.

```
function buildLayer()
  turtle.up()
  buildRow(length)
  turtle.turnRight()
  buildRow(width)
  turtle.turnRight()
  buildRow(length)
  turtle.turnRight()
  buildRow(width)
  turtle.turnRight()
end
```

This function calls the `buildRow()` function using the variables we set earlier, to make a whole layer of the house. Note that it turns right every time. This means that you must start the house at the bottom left corner.

```
function findItems()
  local slot = 2
  while slot < 16 do
    if turtle.getItemCount(slot) < 1  then
      slot = slot + 1
    else
      turtle.select(slot)
      return true
    end
  end
  return false
end
```

This is our final function. It searches for items starting in slot 2, the slot right after the fuel slot. Note that it doesn't check to make sure they are placable items, and will not stop you from trying to build a house made of carrots.

Now that we have useful functions to call, we can write the part of the program that does things!

```
for i=1,height do
  buildLayer()
end
```

All this does is call buildLayer() for the number of layers you specified with `height`

And we're done! Feed your turtle coal and a building material and run the program with `room <length> <width> <height>`

# Mining

Another useful program! You can point this one at a wall and it will dig a two block tall and one block wide tunnel. It will also place torches every 8 blocks.

We'll also build this one with several different functions. The first will check if the turtle is on solid ground, and if not, it will palce a block below itself.

```
function placeFloorBlock()
  if not turtle.detectDown() then
    turtle.select(3)
    if turtle.placeDown() then
      return true
    end
    print("Placing Floor Failed")
    return false
  end
end
```

This next function is a basic check for fuel, and if it sees that the turtle is low on fuel, it will try to refuel from inventory slot 1.

```
function fuel()
  if turtle.getFuelLevel() < 10 then
    turtle.select(1)
    if turtle.refuel(1) then
      return true
    end
    print("Refuelling Failed")
    return false
  end
end
```

You can see that this function returns true if the refueling was successful, and false if not.

Our next function digs our tunnel. The basic idea of this while and if loop is that if the turtle is not moving forward, it will dig, and if it can't dig, it attacks. It also detects blocks above it and digs those as well.

```
function DigAndMove()
  while not turtle.forward() do
    if not turtle.dig() then
      turtle.attack()
    end
  end
  while turtle.detectUp() do
    turtle.digUp()
    sleep(0.5)
  end
  placeFloorBlock()
end
```

This function turns the turtle around. Should be obvious.

```
function turnAround()
  turtle.turnLeft()
  turtle.turnLeft()
end
```

This is a function that turns the turtle to the right, digs out a block, checks make sure that the block didn't get filled in, and then places a torch in the space. If it fails to place a torch, it returns false, otherwise it returns true.

```
function placeTorch()
  turtle.turnRight()
  turtle.dig()
  if not turtle.detect() then
    turtle.select(2)
    if turtle.place() then
      turtle.turnLeft()
      return true
    end
  end
  print("Place Torch Failed")
  turtle.turnLeft()
  return false
end
```

Again, this is a piece of code that accepts and checks arguments to make sure they work with the program.

```
local tArgs = {...}
if #tArgs ~= 1 or tonumber(tArgs[1]) == nil or math.floor(tonumber(tArgs[1])) ~= ton
  print("Usage: tunnel+ <length>")
  return
end
```

Here we set two variables, one for the length you give as an argument, and one to use in the `moveBack` section of code.

```
local length = tonumber(tArgs[1])
local moveBack = 0
```

Now we put it all together.

- First we fuel the turtle by calling `fuel()`.
- Then we call `DigAndMove()` to move the turtle forward one tunnel section.
- We increment the `blocksMovedForward` variable to keep track of where we are.
- Next we check if the block we are on is a multiple of 8.
- If it is, we call the `placeTorch()` function.
- At the end, we turn the turtle around and retrace our steps back to the start of the tunnel.

```
local blocksMovedForward = 0
while blocksMovedForward < length do
  fuel()
  DigAndMove()
  blocksMovedForward = blocksMovedForward + 1

  -- Add torch every 8 blocks
  if (blocksMovedForward % 8) == 0 then
    placeTorch()
  end

  if blocksMovedForward == length then
    turnAround()
    while moveBack < length do
      turtle.forward()
      sleep(1)
      print("Taking step: ")
      print(moveBack)
      moveBack = moveBack + 1
    end
  end
end
```

# Appendix 1: MCU Servers

Access to the camp server is contingent on behavior guidelines just like those in real life. Anything you wouldn't do to someone else in real life, you should not do on the server. PVP is turned off. Do not ask for it to be turned on.

Violation of any of these rules will result in bans on a graduated scale described below.

1. No griefers. A griefer is a player in a multiplayer video game who deliberately irritates and harasses other players within the game.
2. No stealing. You wouldn't just walk into someone else's house and take things out of their closet. Similarly, you may not remove things from other player's chests without consequences. The only exceptions to this are chests marked with a sign as community chests, and food in the case of emergency.
3. No bad language.
4. No raging or rage quitting. Even if not intended as raging, all caps chatting may be interpreted as such.
5. Fill creeper holes and repair any other damage done by creeper blasts.
6. Replant community crops, and leave as much behind in community chests as you take out.

Punishment for breaking any of the server rules are as follows:

1. Verbal warning through chat.
2. If immediately reachable, warning to a parent.
3. Kick.
4. Ban for 24 hours.
5. Ban for a week.
6. Ban for three weeks.
7. Permanent ban. Reversible only through appeal.