

Extracted from:

# 3D Game Programming for Kids

Create Interactive Worlds with JavaScript

This PDF file contains pages extracted from *3D Game Programming for Kids*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2013 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

# 3D Game Programming for Kids

# Create Interactive Worlds With JavaScript



Chris Strom

*Edited by Fahmida Y. Rashid*

# 3D Game Programming for Kids

Create Interactive Worlds with JavaScript

Chris Strom

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at <http://pragprog.com>.

The team that produced this book includes:

Fahmida Rashid (editor)  
Potomac Indexing, LLC (indexer)  
Candace Cunningham (copyeditor)  
David J Kelly (typesetter)  
Janet Furlow (producer)  
Juliet Benda (rights)  
Ellie Callahan (support)

Copyright © 2013 The Pragmatic Programmers, LLC.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.  
ISBN-13: 978-1-937785-44-4  
Encoded using the finest acid-free high-entropy binary digits.  
Book version: P1.0—September, 2013

*For Greta, so that she knows she can do  
anything.*

*When you're done with this chapter, you will*

- *Understand a superpowerful tool (functions) for programmers*
- *Know two reasons to use functions*
- *Recognize some common JavaScript errors and know how to fix them*

## CHAPTER 5

# Functions: Use and Use Again

We've come across functions more than once. Most recently we saw them in [Chapter 4, Project: Moving Avatars, on page ?](#), where we used them to make a forest. If you were paying close attention, you may have noticed that we also used a function to build the keyboard event listener in the same chapter.

Although we have used functions already, we haven't talked much about them. You may already have a sense that they are pretty powerful, so let's take a closer look now.

We're not going to talk about every aspect of functions—they can get quite complicated. We'll talk about them just enough to be able to understand the functions that we use throughout the book.

## 5.1 Getting Started

Create a new project in the ICE Code Editor. Use the Empty project template and call it Functions.

After the opening `<script>` tag, delete the line that says “Your code goes here,” and enter the following JavaScript.

```
var log = document.createElement('div');
log.style.height = '75px';
log.style.width = '450px';
log.style.overflow = 'auto';
log.style.border = '1px solid #666';
log.style.backgroundColor = '#ccc';
log.style.padding = '8px';
log.style.position = 'absolute';
log.style.bottom = '10px';
log.style.right = '20px';
document.body.appendChild(log);
```

```

var message = document.createElement('div');
message.textContent = 'Hello, JavaScript functions!';
log.appendChild(message);

message = document.createElement('div');
message.textContent = 'My name is Chris.';
log.appendChild(message);

message = document.createElement('div');
message.textContent = 'I like popcorn.';
log.appendChild(message);

```

The first chunk of that code creates a place within the browser to log messages. The last three blocks of code write three different messages to that log. If you have everything typed in correctly, you should see the three messages printed at the bottom right of the page.

<pre> 19 message = document.createElement('div'); 20 message.textContent = 'My name is Chris.'; 21 log.appendChild(message); 22 23 message = document.createElement('div'); 24 message.textContent = 'I like popcorn.'; 25 log.appendChild(message); 26 &lt;/script&gt; </pre>	<p>Hello, JavaScript functions! My name is Chris. I like popcorn.</p>
--	---

Back in [Chapter 3, Project: Making an Avatar, on page ?](#), we used a function to avoid having to repeat the same process for creating a tree four times. So you can probably guess the first thing that we'll change here. Let's change the way we log those three messages.

Start by deleting everything from the first `var message` line all the way through the last `log.appendChild` line. Where that code was, add the following.

```

logMessage('Hello, JavaScript functions!', log);
logMessage('My name is Chris.', log);
logMessage('I like popcorn.', log);

function logMessage(message, log) {
  var holder = document.createElement('div');
  holder.textContent = message;
  log.appendChild(holder);
}

```

When we write that code, a surprising thing happens—it gets easier to read. Even nonprogrammers could read those first three lines and figure out that they send a message to the log. This is a *huge* win for programmers like us.

If we decide later that we want to add the time before each message, now it's much easier to figure out where to make that change.

---

### Readable Code Is Easier to Change Later

---



One of the skills that separates great programmers from good programmers is the ability to change working code. And great programmers know that it's easier to make changes when the code is easy to read.

---

Obviously we need to change something inside the function. Before, it would have taken us some time to figure out that those three code blocks were writing log messages, and how to change them.

This also brings up a very important rule.

---

### Keep Your Code DRY—Don't Repeat Yourself

---



This book was published by the same people behind a famous book called *The Pragmatic Programmer*. If you keep programming, you'll read that book one day. It contains a fantastic tip that programmers should keep their code DRY—that they follow the rule known as Don't Repeat Yourself (DRY for short).

---

When we first wrote our code, we repeated three things:

1. Creating a holder for the message
2. Adding a text message to the holder
3. Adding the message holder to the log

It was easy to see that we were repeating ourselves since the code in each of the three chunks was identical except for the message. This is another opportunity for us to be lazy. If we add more than three messages, we only have to type one more line, not three.

And of course, if we have to change something about the log message, we only have to change one function, not three different blocks of code.

We're not quite finished using functions here. If you look at all of the code, you'll notice that it takes a long time to get to the important stuff. (See [Figure 7, A Lot of Junk Before the Function, on page 10.](#))

The important work—writing the messages—doesn't start until line 15. Before we write messages to the log we need a log, but all of that other stuff is just noise.

To fix that, let's move the noise into a function below the `logMessage()` lines. Add a new function named `makeLog()` in between the three lines that call `logMessage()` and where we defined the `logMessage()` function. The “noise” of





Figure 7—A Lot of Junk Before the Function

creating the log holder that goes in `makeLog()` starts with the line that says `var log = document.createElement('div');` and ends with the line `document.body.appendChild(holder)`. Move those lines and everything in between into `makeLogM()`:

```

function makeLog() {
  var holder = document.createElement('div');
  holder.style.height = '75px';
  holder.style.width = '450px';
  holder.style.overflow = 'auto';
  holder.style.border = '1px solid #666';
  holder.style.backgroundColor = '#ccc';
  holder.style.padding = '8px';
  holder.style.position = 'absolute';
  holder.style.bottom = '10px';
  holder.style.right = '20px';
  document.body.appendChild(holder);

  return holder;
}

```

Note that we have changed `log` to `holder`. Also, don't forget the last line, which returns `holder` so that we can do something else with it.

We can create our log with this function. Our first four lines after the opening `<script>` tag become the following:

```

var log = makeLog();
logMessage('Hello, JavaScript functions!', log);
logMessage('My name is Chris.', log);
logMessage('I like popcorn.', log);

```

That is some very easy-to-read code!

It's more difficult to write code like that than you would think. Really good programmers know not to use functions until there's a good reason for them. In other words, great programmers do exactly what we've done here: write working code, then look for ways to make it better.

---

#### Always Start with Ugly Code

---

You are a very smart person. You have to be to have made it this far. So you must be thinking, “Oh, I can just write readable code to begin with.”



Believe me when I say that you can't. Programmers know this so well that we have a special name for trying it: *premature generalization*. That's a fancy way to say it's a mistake to guess how functions are going to be used before you write *ugly* code. Programmers have fancy names for mistakes that we make a lot.

---

## 5.2 Understanding Simple Functions

So far we have looked at reasons why we want to use functions. Now let's see how functions work.

Remove the three `logMessage()` lines from the code. Write the following after the `var log = makeLog` line.

```
logMessage(hello('President Obama'), log);
logMessage(hello('Mom'), log);
logMessage(hello('Your Name'), log);

function hello(name) {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

The result of this `hello()` function would be to first return the phrase “Hello, President Obama! You look very pretty today :)”. Logging these phrases should look something like this:

```
Hello, President Obama! You look very pretty today :)
Hello, Mom! You look very pretty today :)
Hello, Purple Fruit Monster! You look very pretty
```

There is a lot going on in the `hello` function to make that work, so let's break down the function into smaller pieces.

```

1 function hello(name) {
2   return 'Hello, ' + name + '! You look very pretty today :>';
}

```

The pieces of a function are as follows:

- 1 The word `function`, which tells JavaScript that we're making a function.

The name of the function—hello in this case.

Function *arguments*. In this case, we're accepting one argument (`name`) that we'll use inside the function body. When we call the function with an argument—`hello(Fred)`—we're telling the function that any time it uses the `name` argument, it is the same as using `Fred`.

The body of the function starts with an open curly brace, `{`, and ends with a closing curly brace, `}`. You may never have used curly braces when writing English. You'll use them a lot when writing JavaScript.

- 2 The word `return` tells JavaScript what we want the result of the function to be. It can be anything: numbers, letters, words, dates, and even more-interesting things.

JavaScript lines, even those inside functions, should end with a semicolon.

---

#### Letters, Words, and Sentences Are Strings

---



Things inside quotes, like `'Hello'`, are called *strings*. Even in other programming languages, letters, words, and sentences are usually called strings.

Always be sure to close your quotes. If you forget, you'll get very weird errors that are hard to fix.

---

Next, let's try to break it intentionally so that we get an idea of what to do when things go wrong.

## 5.3 When Things Go Wrong

Let's put our hacker hats on and try to break some functions.

Although it's easy to do something wrong with JavaScript functions, it's not always easy to figure out what you did wrong. The most common mistakes that programmers make generate weird errors. Let's take a look so that you'll be better prepared.

---

### Hack, Don't Crack

---



Don't worry! We won't really break anything. Breaking something would be *cracking*, not *hacking*. Hacking is a good thing. You'll often hear nonprogrammers using the word *hack* wrongly. Since you're a programmer now, you need to know what the word means and how to use it correctly. Hacking means that we are playing around with code, an application, or a website. We play with it to learn, not to cause damage. And sometimes we try to break our own code—but only to better understand it.

Hack always. *Never* crack.

---

### Unexpected Errors

The most common thing to do is forget a curly brace:

```
// Missing a curly brace - this won't work!
function hello(name)
    return 'Hello, ' + name + '! You look very pretty today :>';
}
```

This is a compile-time error in JavaScript—one of the errors that JavaScript can detect when it's trying to read, compile, and run—that we encountered in [Section 2.5, \*Debugging in the Console\*, on page ?](#). Since it's a compile-time error, the ICE Code Editor will tell us about the problem.

```
9 // Missing a curly brace - this won't work!
10 function hello(name) |
11     return 'Hello, ' + name + '! You look very pretty today :>';
12
13
```

What happens if we put the curly brace back, but remove the curly brace after the return statement?

```
// Missing a curly brace - this won't work!
function hello(name) {
    return 'Hello, ' + name + '! You look very pretty today :>';
}
```

There are no errors in our hello function, but there is an error at the very bottom of our code.

```
8
9 Unmatched '{'. // Missing a curly brace - this won't work!
10 function hello(name) {
11     return 'Hello, ' + name + '! You look very pretty today :>';
12
13
```

This can be a tough error to fix. Often programmers will type many lines and possibly several functions before they realize that they have done something wrong. Then it takes time to figure out where you meant to add a curly brace.

## Challenge

Try to figure out the following broken code on your own. Where do the errors show up? *Hint:* as in [Section 2.5, Debugging in the Console, on page ?](#), some of these may be run-time errors.

Forgot the parentheses around the argument:

```
function hello name {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Forgot the function's argument:

```
function hello() {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Wrong variable name inside the function:

```
function hello(name) {
  return 'Hello, ' + person + '! You look very pretty today :)';
}
```

Function called with the wrong name:

```
logMessage(helo('President Obama'), log);

function hello(name) {
  return 'Hello, ' + name + '! You look very pretty today :)';
}
```

Wow! There sure are a lot of ways to break functions. And believe me when I tell you that you'll break functions in these and many other ways as you get to be a great programmer.

---

### Great Programmers Break Things All the Time

---



Because they break things so much, they are *really* good at fixing things. This is another skill that makes great programmers great.

---

Don't ever be upset at yourself if you break code. Broken code is a chance to learn. And don't forget to use the JavaScript console like you learned in [Playing with the Console and Finding What's Broken](#) to help troubleshoot!

---

## 5.4 Weird Tricks with Functions

Functions are so special in JavaScript that you can do all sorts of crazy things to them. Whole books have been written on “functional” JavaScript, but let’s take a look at one trick that we will use later.

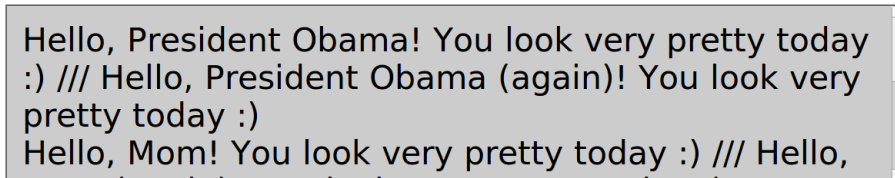
### Recursion

Change the hello like this:

```
function hello(name) {
  var ret = 'Hello, ' + name + '!' + 'You look very pretty today :)';
  if (!name.match(/again/)) {
    ret = ret + ' /// ' + hello(name + ' (again)');
  }
  return ret;
}
```

- ❶ Look closely here. Inside the body of the function hello, we’re calling the function hello!

This will log the hello messages twice.



```
Hello, President Obama! You look very pretty today
:) /// Hello, President Obama (again)! You look very
pretty today :)
Hello, Mom! You look very pretty today :) /// Hello,
```

A function that calls itself like this is actually not crazy. It is so common that it has a special name: a *recursive function*.

Be careful with recursive functions! If there is nothing that stops the recursive function from calling itself over and over, you’ll lock your browser and have to go into edit-only mode to fix it, as described in [Section 2.6, Recovering When ICE Is Broken, on page ?](#).

In this case, we stop the recursion by calling the hello function again only if the name variable doesn’t match the again. If name doesn’t match again, then we call hello() with name + '(again)' so that the next call will include again.

Recursion can be a tough concept, but you have a great example in the name of your code editor:

- What does the I in ICE Code Editor stand for?
- It stands for ICE Code Editor.
- What does the I in ICE Code Editor stand for?

- It stands for ICE Code Editor.
- What does the I in ICE Code Editor stand for?
- ...

You could go on asking that question forever, but eventually you'll get sick of it. Computers don't get sick of asking, so you have to tell them when to stop.

## 5.5 The Code So Far

In case you would like to double-check the code in this chapter, it's included in [Section A1.5, Code: Functions: Use and Use Again, on page ?](#).

## 5.6 What's Next

Functions are very powerful tools for JavaScript programmers. As we saw, the two main reasons to use a function are for reuse and for making code easier to read. We created a `logMessage()` function so that we could use its functionality over and over again. We created the `makeLog()` function to move a whole bunch of messy code out of the way of more important code. We even took a peek at some of the crazy things that we can do with functions, like recursion.

And we're still just scratching the surface!

As you'll see shortly, we'll use functions a lot in the upcoming chapters. Let's get started in the next chapter as we teach our avatar how to move its hands and feet!