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

The Pragmatic Programmers

3D Game Programming for Kids

Create Interactive Worlds With JavaScript

Chris Strom

Edited by Fahmida Y. Rashid

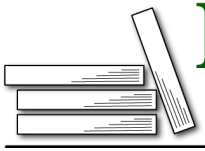
3D Game Programming for Kids

Create Interactive Worlds with JavaScript

Chris Strom

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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.

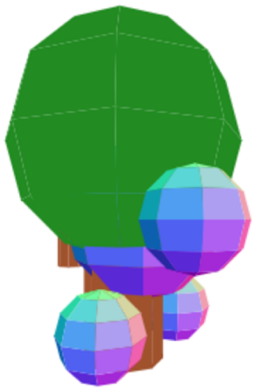
When you're done with this chapter, you will

- *Be able to stop game elements from moving through each other*
- *Understand collisions, which are important in gaming*
- *Have game boundaries for your avatar*

CHAPTER 10

Project: Collisions

We have a pretty slick game avatar. It moves, it walks, it even turns. But you may have noticed something odd about our avatar. It can walk through trees.



In this chapter we'll use tools that are built into our Three.js 3D JavaScript library to prevent the avatar-in-a-tree effect. (As we'll see in other chapters, there are other ways to do the same thing.)

10.1 Getting Started

If it's not already open in the ICE Code Editor, open the project from *Project: Turning Our Avatar* that we named My Avatar: Turning.

Make a copy of our avatar project. From the menu in the ICE Code Editor, select Make a Copy and enter My Avatar: Collisions as the new project name.

10.2 Rays and Intersections

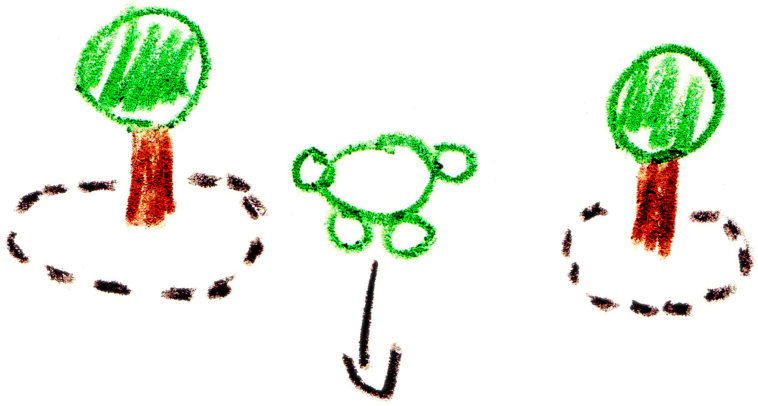
The way we prevent our avatar from walking through trees is actually quite simple. Imagine an arrow pointing down from our avatar.

In geometry, we call an arrow point a *ray*. A ray is what you get when you start in one place and point in a direction. In this case, the place is where our avatar is and the direction is down. Sometimes giving names to such simple ideas seems silly, but it's important for programmers to know these names.

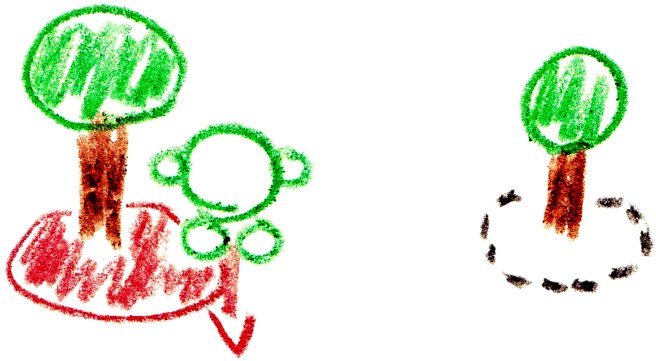
Programmers Like to Give Fancy Names to Simple Ideas

Knowing the names for simple concepts makes it easier to talk to other people doing the same work. Programmers call these names *patterns*.

Now that we have our ray pointing down, imagine circles on the ground around our trees.



Here is the crazy-simple way that we prevent our avatar from running into a tree: we don't! Instead, we prevent the avatar's ray from pointing through the tree's circle.



If, at any time, we find that the next movement would place the avatar's ray so that it points through the circle, we stop the avatar from moving. That's all there is to it!

Star Trek II: The Wrath of Khan

It may seem strange, but watching certain science-fiction movies will make your life easier as a programmer. Sometimes programmers say odd things that turn out to be quotes from movies. It is not a requirement to watch or even like these movies, but it can help.

One such quote is from the classic *Star Trek II: The Wrath of Khan*. The quote is "He is intelligent, but not experienced. His pattern indicates two-dimensional thinking."

The bad guy in the movie was not accustomed to thinking in three dimensions, and this was used against him. In this case, we *want* to think about collisions in only two dimensions even though we are building a three-dimensional game. We're thinking about collisions only in two dimensions (X and Z), completely ignoring the up-and-down Y dimension.

This is yet another example of cheating whenever possible. *Real* 3D collisions are difficult and require new JavaScript libraries. But we can cheat and get the same effect in many cases using easier tricks.

At this point, a picture of what to do next should be forming in your mind. We'll need a list of these tree-circle boundaries that our avatar won't be allowed to enter. We'll need to build those circle boundaries when we build the trees, and detect when the avatar is about to enter a circle boundary. Last, we need to stop the avatar from entering these forbidden areas.

• Click HERE to purchase this book now. discuss

Let's establish the list that will hold all forbidden boundaries. Just below the START CODING ON THE NEXT LINE line, add the following.

```
var not_allowed = [];
```

Recall from Section 7.5, *Listing Things*, on page ?, that square brackets are JavaScript's way of making lists. Here, our empty square brackets create an empty list. The not_allowed variable is an empty list of spaces in which the avatar is not allowed.

Next, find where makeTreeAt() is defined. When we make our tree, we'll make the boundaries as well. Add the following code after the line that adds the treetop to the trunk, and before the line that sets the trunk position.

```
var boundary = new THREE.Mesh(
  new THREE.CircleGeometry(300),
  new THREE.MeshNormalMaterial()
);
boundary.position.y = -100;
boundary.rotation.x = -Math.PI/2;
trunk.add(boundary);

not_allowed.push(boundary);
```

There's nothing superfancy there. We create our usual 3D mesh—this time with a simple circle geometry. We rotate it so that it lays flat and position it below the tree. And, of course, we finish by adding it to the tree.

But we're not quite done with our boundary mesh. At the end, we push it onto the list of disallowed spaces. Now every time we make a tree with the makeTreeAt() function, we're building up this list. Let's do something with that list.

At the very bottom of our code, just above the </script> tag, add the following code to detect collisions.

```
function detectCollisions() {
  var vector = new THREE.Vector3(0, -1, 0);
  var ray = new THREE.Ray(marker.position, vector);
  var intersects = ray.intersectObjects(not_allowed);
  if (intersects.length > 0) return true;
  return false;
}
```

This function returns a Boolean—a yes-or-no answer—depending on whether the avatar is colliding with a boundary. This is where we make our ray to see if it points through anything. As described earlier, a ray is the combination of a direction, or *vector* (down in our case), and a point (in this case, the

avatar's marker.position). We then ask that ray if it goes through (intersects) any of the not_allowed objects. If the ray does intersect one of those objects, then the intersects variable will have a length that is greater than zero. In that case, we have detected a collision and we return true. Otherwise, there is no collision and we return false.

Collisions are a tough problem to solve in many situations, so you're doing great by following along with this. But we're not quite finished. We can now detect when an avatar is colliding with a boundary, but we haven't actually stopped the avatar yet. Let's do this in the keydown listener.

In the keydown listener, if an arrow key is pressed, we change the avatar's position.

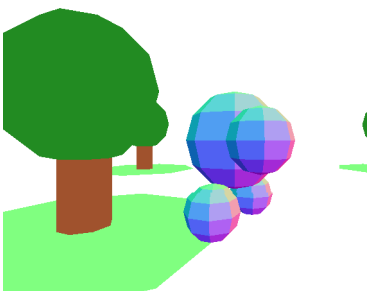
```
if (code == 37) {                                    // left
  marker.position.x = marker.position.x-5;
  is_moving_left = true;
}
```

Such a change might mean that the avatar is now in the boundary. If so, we have to undo the move right away. Add the following code at the bottom of the keydown event listener (just after the if (code == 70)).

```
if (detectCollisions()) {
  if (is_moving_left) marker.position.x = marker.position.x+5;
  if (is_moving_right) marker.position.x = marker.position.x-5;
  if (is_moving_forward) marker.position.z = marker.position.z+5;
  if (is_moving_back) marker.position.z = marker.position.z-5;
}
```

Read through these lines to make sure you understand them. That bit of code says *if we detect a collision, then check the direction in which we're moving. If we're moving left, then reverse the movement that the avatar just did—go back in the opposite direction the same amount.*

With that, our avatar can walk up to the tree boundaries, but go no farther.



Yay! That might seem like some pretty easy code, but you just solved a very hard problem in game programming.

10.3 The Code So Far

In case you would like to double-check the code in this chapter, it's included in Section A1.10, *Code: Collisions*, on page ?.

10.4 What's Next

Collision detection in games is a really tricky problem to solve, so congratulations on getting this far. It gets even tougher once you have to worry about moving up and down in addition to left, right, back, and forward. But the concept is the same. Usually we rely on code libraries written by other people to help us with those cases. In some of the games we'll experiment with shortly, we'll use just such a code library.

But first we'll put the finishing touch on our avatar game. In the next chapter we'll add sounds and scoring. Let's get to it!

• Click HERE to purchase this book now. discuss