==========

CopperCube Building Blocks

==========

A linear guide to CopperCube 6.6

==========

J.H. Reidhead III (RedBerylFTW)

==========

==========

PREFACE:

==========

I remember picking up CopperCube for the first time a few years ago. This was before I started to develop a more complex understanding of programming and the game development process. One of the most favorable things about an engine like CopperCube is just how easy and convenient it is to use, despite its many powerful capabilities. With absolutely no knowledge, you can pick it up and start creating a game. Those games will probably be bad for a while, but eventually you'll start to develop your own methods and styles for solving problems.

The current problem with CopperCube for any user who wants a beginner-to-advanced experience is the lack of tutorials or documentation. The CopperCube documents are unhelpful or incomplete in a lot of areas, which makes even simple concepts seem more complicated than they actually are. The few video tutorials that exist are very helpful, but there is still a lot of uncovered territory. The aim of this guide is to give the reader a general understanding of how to use CopperCube with simple explanations and easy-to-follow examples. By the end of this book, the reader will have created a complete game with a main menu, an objective, an ending, and more. This doesn't aim to be the complete end-all tutorial for CopperCube. Instead, this guide aims to be friendly enough for beginners and complete enough for intermediates. There is no assumption that the reader will already know how to code or how to use CopperCube at all. Or even a program, for that matter.

I would like to thank Ambiera for creating such a convenient and powerful piece of software. Without the hard work of Niko, none of this would even be possible. I would also like to thank the entire CopperCube community for their invaluable collection of information spread between the forums, Discord, Steam discussions and otherwise. Without them, it wouldn't be nearly so easy to develop games with this engine. So much work has been done to experiment with every bit of functionality, and so many impressive games have been made in the process. The community is the heart and backbone of this engine, and they all deserve credit for their amazing efforts.

Contents:

==========

Chapter 1:
Introduction | The CopperCube Editor

==========

IN THIS CHAPTER YOU'LL LEARN:
-How to navigate and use the CopperCube editor
-The concept of "Parent and Child" nodes
-An introduction to "Behaviors and Actions"

==========

You're probably here reading this document because you wanted to use CopperCube but had no clue where to start. Adversely; you may be an advanced CopperCube user just looking to refresh yourself on some of the basics. This document is intended to provide detailed information about CopperCube in a way that is linear and easy to understand. We'll begin with an overview of the CopperCube editor, and over the course of the guide we will work our way up from creating a simple menu screen to creating a complete simple game. Because the flow of this guide is linear, you are heavily discouraged from skipping around unless you are certain of what you are looking for. To get started, let's open up CopperCube and create a new blank project.

The first quest on our journey is to familiarize ourselves with the CopperCube editor. Take a moment and just look at it. In traditional format: The top of the window has a toolbar full of drop-down menus with tools that we will use throughout this course. Don't concern yourself with it all now, but open the "File" menu and notice the "Save As" button. This will let you save a new file without overwriting the previous one. Just below the top toolbar, we have the editor's standard functions. To the far left, we have "Save" as a floppy disk, "Load" as a folder, and "Undo and Redo" buttons. Those are self explanatory. Below that, we have "Select", "Move", "Rotate", and "Scale" which can all be used to manipulate objects within the scene editor. Directly to the right we have the editor viewpoint selectors. "Perspective" is the default flying camera. "Top" will show you the scene from a top-down perspective. As you might have guessed, "Front" and "Left" will show you the scene from either the front or the left. Have you familiarized yourself with these buttons? Good, let's move on.

To the right, you may have noticed that this bar has multiple tabs. This collection is the "Editor toolkit". I'll give you a boilerplate description of what each tab is for, but we'll wait to talk about what each thing actually does until we start using stuff. The "Create" tab is where you'll find most of your tools to create primitive objects. This includes things like basic 3D shapes, 2D items, 3D sound emitters and more. Coppercube has some other nifty primitives that we'll talk about in a while. The "Polygon Editing" tab is exactly what it sounds like. It provides you with a selection of tools for editing 3D models and UV texture maps from within the editor. Cool stuff! The "Light Mapping" tab contains a generator that will create special textures called "Light Maps" based on the lights that you create in your scene. The "Scene Editor" tab has a few useful tools. First is the "Scene selector" where you can choose what scene to edit from a list of scenes that you've created. Remember this one, we'll need it soon. The "Settings" button will let you change the name of the currently selected scene.

The "Scene Metrics" button is a handy tool that we won't mess with right now. Last, the PostFX tool. You may or may not have access to this, so we won't talk about it right now either.

On the left side of the editor window below the tool bars, you'll find the "SceneGraph Explorer" which displays a list of objects that are currently in the scene. CopperCube refers to objects as "Nodes" so we will also call them nodes from this point forward. The first node in the list is the Main scene node. If you're familiar with writing code, you can consider this the program entry point. Everything contained within the main scene node is called a "child node" and each child node can have its own children. This allows us to do more complicated things that involve multiple objects. We'll talk about that more in depth later. You may have noticed that CopperCube starts with a default set of child nodes. Like most 3D editors, you get a single cube in the center of the world space. There is also a default skybox that will always draw at the farthest coordinates from the camera position.

Click on your main scene node and turn your attention to the "Properties" window. In the "Attributes" tab, you'll see what the "Type" of node is. You'll also see some very basic global things you can change. Click on the default skybox child node and press your "Del" key. Poof, it's gone. Back in the Properties window, change the "BackgroundColor" attribute to something else. You'll immediately see the change in the viewport to the right. This is a very basic demonstration of a node attribute. Later on, your node attributes will include things like "Scale" or "Rotation". The next tab is the "Materials" tab. Click on your cubeMesh1 child node, and then click on the Materials tab. This is where you can assign a texture to a node. You can also change what kind of lighting the node will use. We'll talk about that later when we discuss lighting in more depth. The final tab is the "Behavior" tab. It's mostly empty right now, but this is the most important part of the Properties window. This is where we can tell nodes what they are supposed to be doing. The editor comes with a selection of readily-available behaviors that your node can use for simple things, or you can go the extra mile and code your own behaviors with Javascript. This gives you a lot of flexibility as a programmer to create more complex games. We'll talk more about Javascript later.

Some behaviors have a special attribute called an "Action". Actions can be used to make a behavior more complex. Click on the "+" icon next to the empty list. Find "Behaviors triggered by events" and then click "When clicked on this do something". The last attribute in the behavior should be called "Action" with a little "..." icon at the end. Click on that icon. This will bring up the "Edit Actions" window. Once again, click on the "+" icon. You will see a wide variety of pre-made actions available for you to use, and just like behaviors; You can code a new action yourself in Javascript. From this point forward, I will tell you to do things like "Assign a behavior" or "Assign an action" but I will not explain how to do these things again. For convenience's sake, I hope you haven't been skipping around.

Below the Properties window and the viewport, you'll find two more windows. First is the "Prefabs" window. You don't have to worry about it right now, but later I will talk about what it is and teach you how to make one. To the right of that, you have the "Textures" window. In this window, you can add or remove textures that you will use in your game. I'll go ahead and tell you right now that it is generally good practice to remove all of the unused default textures if you aren't going to use them, otherwise they will be packed into your game as "dead data" or data that will never be used. You can remove textures by right clicking on the texture at the end of the list and then clicking "Remove Unused Textures". You cannot remove every texture. There are six skybox textures, one empty white texture, one metal block texture, and the coppercube logo texture. Those 9 textures cannot be removed.

We've finally reached the end of chapter one. Hopefully you've paid great attention and you've become familiar with the editor. It's exercise time! Learning how to develop a game is a participation sport, so at the end of every chapter I will provide you with a few exercises to secure what you have learned. Don't move forward until you've done the following things:

-Save two .ccb project files
-Remove unused default textures
-Add one new texture
-Give the scene fog and give that fog a color

==========

Chapter 2:
Your First CopperCube Program

==========

IN THIS CHAPTER YOU'LL LEARN:
-How to assign a behavior to a node
-How to pass an action to a behavior
-How to make a functional simple menu

==========

Hopefully you took the time to work through those exercises before you started reading this. It might seem silly to have you doing menial tasks like saving a file or removing the textures, but I assure you that repeating these basic tasks will structure your mind and your workflow with basic philosophies. We saved two project files first because we always want to save before we make changes to a project. It's good practice to make a backup of your last best save, so you can go back to it if you make a mistake that you don't know how to fix. We removed every unused texture because we want to remember that we do not need unused assets when we create a completely new project. Ideally we don't want to have any dead data. Do not leave dead models or textures in your project. Add one new texture just so you can get a feeling for how textures must be added before they can be applied to models or 2D graphics. Finally, give the scene fog and give the fog a color will test what you remember about the SceneGraph explorer and the Properties window. Please go back and make sure that you've done those before proceeding, if only for the sake of memorizing the editor.

When you begin using a new programming language, it's typical to begin that journey with a simple "Hello World" program that displays an identical message in a console window. We aren't going to do that because this isn't a language, this is an editor. But we are going to do something just as simple. Our first project will be a very basic title screen with a "start" button and a "quit" button. I promise it isn't rocket science. We will walk through the procedure step by step. By the end, we will have a working start menu that can load a new scene and terminate the program.

We'll begin by creating a new empty project. Remember to remove all of the default textures. Also, please delete the default skybox and the starting cube. Finally, set the viewport perspective to front. We now have a fresh development environment. Navigate to the "scenes" tab of the toolbox, press the "settings" button, and rename your new scene to "titleScreen" exactly as it is written here. This style of naming is called "Camel Case" and we will be using camelCase conventions for naming all of our scenes. This clean aesthetic is optional, but will be beneficial to you in the long run. It will help you stay organized, and will make it easier to access scenes with Javascript.

Back in the "Create" tab of the toolbox, find the "2D Item" primitive. Click on it, and you will automatically create a 2D rectangle on the screen. In the Properties window, set the name to "Backdrop". Then, set the Pos X and Pos Y attributes to 0. Set the Width to 100, and the height to 100. Finally, set the Background Color to Black and the Alpha to 255. Create a second 2D Item. Name it "Header". Set the Pos X to 40, Pos Y to 0, Width to 20 and Height to 15. Set the alpha to 0. You may have noticed that the size and position attributes are calculating the percentage instead of the pixels. That's because these 2D objects are meant to be scalable for multiple resolutions. This menu should look similar whether we export in 480p or 1080p. A very convenient solution.

Turn your eyeballs down toward the "Draw Text" option. Tick the box and you'll have access to a few new attributes. First, click on the empty space after "Text" and type "My First Menu". Next, move down to the Font attribute. You can change this in two ways, we'll just use the simple one. Click the "+" icon and change the Point Size to 22. You now have a title header on your screen. Congrats, but we aren't done yet. Create another 2D image, name it "Exit", make the text say Exit, make the font size 30 and position it %80 below the Header. This will be our first button. We'll talk about it more in just a second, but first; Repeat this process for a "Start" button and place it %40 below the Header.

Click on your "Exit" node in the SceneGraph explorer. Navigate to the Behavior tab of the properties window, and give your button the behavior "When clicked on this do something". Open the actions window, and under the "Special" category of the actions menu, select "Quit Application". This will terminate the process when the button is clicked. This will give our button functionality, but we still cannot use it. First, we need to unhook the mouse cursor from the window. To do this; Navigate up to your main titleScreen node. Give it the behavior "Before first draw do something" and tick the "AlsoOnReload" to make this behavior happen whenever the game returns to this screen.

We are going to pass the action "Execute Javascript" to the behavior, but don't feel intimidated by that. You do not need to write complete blocks of Javascript code to use the CopperCube API functions. We are going to write a couple of lines though, so in the action properties, click on the "..." icon. First, write the line:

```
ccbCleanMemory();
```

This will unload everything that isn't being used when the game returns to this menu. This is a form of "Garbage Collection". You don't need it for every scene, but it is good to have for your starting menu. It affects different platforms in different ways, so you may or may not need it. But something that you absolutely DO need is that pretty little semicolon at the end of the line. You don't want to risk your game breaking later on, so make sure to remember to end every function call with a semicolon. Next write the line:

```
ccbSetCursorVisible(true);                    //This one explains itself.
```

Finally, navigate to the create tab of the toolbox and create a new "Camera" node. Select "Simple Camera" and don't do anything else to it. Navigate to the "Publish" tab, and click on the gear-shaped icon to bring up the publishing settings. Set your application name to whatever you want, and then navigate to the tab that is relevant to your target operating system. Set the "Window/Resolution" properties to 1280 width and 720 height. Click the "Ok" button, and then finally click on the triangle-shaped "Run" button to the right. Congratulations, you've just built a functional program with CopperCube! If you click on the "Exit" text, the application will shut down. Pretty cool, huh? But we aren't done yet.

Back in the editor, click on the Scenes tab of the toolbox, and click the "add" button. Remember camelCase. Name the new scene "gameScene1" and click "OK". Once you've loaded into your new scene, set the viewport perspective to "front" and create a new simple camera. In the Properties window, set the camera position to "0, 0, -20" and do not worry about the decimal spaces. The editor will automatically fill those in for you. Select your cubeMesh1 node in the SceneGraph explorer. Give it the behavior "when clicked on this do something" and pass it the action "Cameras and Scenes >> Switch to another scene". Then set the scene attribute to our titleScreen scene. Back up in the editor toolbox, navigate to the Scenes tab and switch back to the titleScreen scene. Select the "Start" node in the SceneGraph explorer. Give it the "when clicked on this do something" behavior, and pass it the action "Switch to another scene". Select the scene attribute and set the "gameScene1" scene. Finally, run your program.

You now have a completely functional title screen that can start and exit the game. You also have an object within your game scene that will return you to the main menu. You have now gone through the entire process of creating a simple menu, creating multiple scenes, and switching between those scenes. Do you feel accomplished? Well you should, because you've just completed a functional program! Congratulations, but don't start making out with yourself just yet. We still have a long way to go before this application becomes a complete game. For now, be happy about your menu. You made it. It's yours.

We've reached the end of this chapter, but that isn't the end of this adventure. I want you to know that you've been paying attention. Before we move on into actually making a game, make sure to complete this one essential exercise.

- Create a new menu from beginning to end without referencing this document
- Change the layout of your menu buttons to be horizontal.

==========

Chapter 3:
Game State and Variables

==========

IN THIS CHAPTER YOU'LL LEARN:
How to declare variables in Javascript
How to declare CopperCube variables
The difference between Javascript and CopperCube variables
Variable data types

==========

In this chapter, we're going to discuss how to define and use variables within the context of CopperCube. If you're familiar with writing code, then you are probably already familiar with this concept. If you aren't familiar with variables, then I'll do my best to explain it in a simple way. After that, we'll talk a little bit about the concept of a "Game State" and we'll go over how to declare a game state variable. By the end, you will have a working demonstration of a local variable, as well as a game state variable. Still have your "My First Menu" project open? Good! Use the top toolbar to save a new .ccb file and name it "Variables" so that we don't mess up any of that work that you've done so far. The file that you most recently saved will automatically be your current active project file, so don't worry about reloading it. Make sure you are viewing the titleScreen scene and that your menu is not missing any components.

If you are already familiar with Variables, then skip this paragraph. For you noobs, here's the rundown; A variable is a name that stores a value. Here is an example: APPLES = 1. In English: There is one apple. Simple, right? If a thing equals a number, you have a number of that thing. Javascript is a little bit more simple than some other programming languages with what data types a variable can hold. We can declare a variable with one of three different words. The first is var, which will define a global variable. That means it can be used outside of its own function. Next is let, which will define a local variable. That means it can only be used within the function where it's defined. Last is Const. Const is only usable within the script where it is written, and it cannot be updated or changed at any time. Next, let's discuss the data types. First is numbers. Javascript allows you to use numbers with or without decimal points and there is no distinction. Next is strings. A string is a word or line of text contained inside of quotation marks. You can use the variable as a shorter way to use a lot of text quickly. The last data type is boolean. This just means true or false. This is used for yes or no conditions. If you still don't understand how variables and data types work, then please read this paragraph again or please read the MDN web documents for Javascript variables and data types.

Coppercube has two different ways to declare and manipulate variables. You can declare any kind of variable within an "execute javascript" action, or you can use the visual scripting tools. For this demonstration, we are actually going to use both. First, we'll implement a global variable and modify it; all using Javascript. After that, we'll use the visual scripting tools to declare a local variable within our gameScene1 scene. All of the results will be printed into the debug console at the top-left corner of the game window, so you can see your variable manipulation in action. Are you ready? Good, let's get started.

Make sure you're viewing your titleScreen scene in the editor. Select your titleScreen node in the SceneGraph explorer, and flip to the behavior tab of the properties window. Open up the action window, and edit the same Javascript action that we used to make the mouse cursor visible. At the bottom of the text editor, add the lines:

```
var GAME_START = 0;
print(GAME_START);
```

Remember to use your semicolons. You might have noticed that the name of the variable is typed in a very specific way. This is an aesthetic choice. We will be defining all global variables in the SNAKE_CASE convention so that we always know exactly what they are and how to find them. If you run your program now, your console will display the number 0. If you don't see the number, make sure you've edited the correct Javascript action and check to make sure you haven't forgotten a semicolon or any parenthesis. If you've got it, then good job! The next step is to modify this variable.

Select your Start node in the SceneGraph explorer, and again open the actions panel. This time, we want to remove the "Switch to another scene" action temporarily. This is because CopperCube passes actions in order from top to bottom. Add the "execute javascript" action first. Then add back the switch scene action, and remember to set the next scene to gameScene1. Edit the execute javascript action with the following code:

```
if (GAME_START = 0)
{
GAME_START = 1;
};

if (GAME_START = 1)
{
print(GAME_START);
};
```

Now let me explain what is happening here. The first if statement is checking for if the GAME_STATE variable is equal to zero. The conditions for the if statement are contained within parenthesis, and that is followed by a set of curly braces. In the middle of those curly braces, we wrote GAME_START = 1 without using the word var. That is because we do not need to declare a new variable. All we need to do is modify an existing one. If you wrote GAME_START = 1 but you didn't enclose it inside of the if statement, then it may declare a new undefined variable and we don't want that to happen. Computers are stupid. We have to tell it that it only wants to change the variable if it isn't already at the value we want.

We have one major limitation with this way of declaring variables. Even though we declared a global variable; It was a Javascript global variable. The Javascript that we executed was bound to the main scene node, and that scene node will not exist in the next scene. That means all vanilla Javascript variables in CopperCube are actually "Local Global Variables". This can be very helpful for controlling complex variables in one scene, but we cannot use it to create persistent variables between multiple scenes. To do that, we first need to declare a CopperCube variable. CopperCube variables are special in that you can store them temporarily and load them between multiple scenes. This is especially helpful for making more complex games. We're going to convert our current Javascript variable into a CopperCube variable, and I promise it won't be that complicated. The available data types are the same.

In your main titleScreen node Javascript action, highlight and cut your first two lines of code because you'll need them in a second. Remove the Javascript action and create a new "Variables and Text >> Set or Change a Variable" action. In the name attribute, write GAME_START just like we did before. Set the "Value" attribute to 0 Create a new Javascript action and paste in your code from the old Javascript action that we deleted. We're going to write a completely new line so that we can see our variable work. You want to add:

```
var x = ccbGetCopperCubeVariable("GAME_START");
print(x);
```

This will print out 0 in the console, same as before. Except this time, we can store the value in memory temporarily and load it into the next scene. Select your Start node and edit the Javascript action. It only needs to do one thing, so write this;

```
if (x = 0)
{
ccbSetCopperCubeVariable("GAME_START",1);
};
```

Next, add the action "Variables and Text >> Load or Store a Variable from or to disk". Write GAME_START in the VariableName attribute, and change the mode attribute to "Save Variable".Finally, select your Switch to another scene action, delete it, and create a new one at the bottom of the list. Now that we've gone through the process, let me explain what's going on here.

In the first line of code that we wrote, var x = ccbGetCopperCubeVariable, we defined the local global X to mean the same thing as the coppercube variable GAME_START. We print that new local global variable to the console, which will return 0. The next code change is the same idea as when we checked for the variable value earlier. If GAME_STATE is equal to 0, then we set GAME_STATE to equal 1. The difference is that we have now used the CopperCube API function to change a CopperCube variable, instead of just writing and changing a standard Javascript variable. It takes a little bit longer, but the benefit of saving and loading global CopperCube variables between scenes is more valuable than the few seconds you'll spend typing extra letters.

Now that we have defined and saved the variable, we need to have it load and print in the next scene. So switch over to gameScene1, click on your gameScene1 node in the SceneGraph explorer, and add a new "Before First Drawing" behavior. The first action should be "Load or Store a Variable from or to disk". Type GAME_START into the VariableName attribute, and leave the default "Load Variable" mode. Next, create a new Javascript action and make it look exactly like this:

```
var x = ccbGetCopperCubeVariable("GAME_START");

if (x = 1)
{
print(x);
}
```

Again, we have defined x as a local global variable to be used in this scene. If we successfully loaded the GAME_START variable, then x should be equal to 1. We will know for sure using the if statement. If x is equal to 1, then the console will print 1. Switch back to your titleScreen scene. There is one final thing to do. Disable the "Also on Reload" attribute as we will almost never use this again. We will make a habit of always manually controlling the state of every variable, which will give us much more control over every piece of game data. We never want the computer to automatically manage crucial variables that we may save or load later.

It's finally time to run your program again. Remember to always run your program while viewing the titleScreen scene, because CopperCube will automatically start at the scene that is currently selected. We will take advantage of this later for testing and debugging purposes. If you successfully followed the instructions, then your console should start by printing 0. If you click your start button, a 1 will be printed in the console. These numbers will only be printed one time, so if you are seeing 1 print multiple times, then you've made a mistake. You should only have a single 0 and a single 1 no matter how many times you change between each scene.

This is the end of the chapter. Hopefully you've been paying great attention, because I've prepared a few exercises to test your understanding of variables and how to manipulate them. If you still

aren't sure how variables work, please read the chapter again and follow along to the best of your ability. Before you move on to the next chapter, please do the following:

- Write a simple script using two variables to print the sum of 5 + 5 in the console
- Save, load and print four CopperCube variables in the console between two scenes
- Divide the 9th Fibonacci number by four and print the remainder.
---Fibonacci is simple. The sum of the last two numbers is the next number. 0, 1, 1, 2, 3, 5, 8, 13, etc.

ALSO: I hope you've been experimenting with these ideas. Remember that a tutorial like this will only be helpful to you if you apply what you learn. Don't be afraid to not follow the rules exactly. If you think your title screen font would look better in red, or you hate SNAKE_CASE like any typical Typescript user, then go ahead and change it to fit your personal style!

==========

Chapter 4:
Creating a Simple Scene

==========

In this chapter, we will turn our gameScene1 scene into a more complete storyboard for what will later become a complete level. Having a storyboard made of primitives is a necessary step in the process of world design. It will allow us to design the layout of the level while also working on some of the core functionality without having any art or assets to work with. It is essential that game programmers have the ability to work on the game even if they do not have the finished models or textures.

Go ahead and switch over to gameScene1. This entire chapter will be done in that scene. Do not worry about removing the unnecessary variable print functions right now. We'll do that later. For now, click on the default cube in the center of the screen and delete it. Don't worry about going back to the menu right now. You can close your application with the ESC key while in debug mode, and we've freed the mouse so you can also use the regular X to close the window. Go ahead and delete the default skybox as well, since we don't need it. Now that we have an empty space to work with, we can get started.

Create a new box with an initial size of 10, and set the scale attribute to "10, 0.1, 1". This will be our starting platform. Name it platform1, respectively. Create a second box with an initial size of five. Name this box "player" and position it slightly above the platform node. We now have the scene set for a platformer. Before we start doing anything though, we need to declare a few variables. First, select your gameScene1 node "Before First Drawing" behavior, and pass a new "Set or Change a Variable" action. We'll name this variable ON_GROUND, set the "Value Type" attribute to "Variable", and give it a "false". careful to remember our SNAKE_CASE typing conventions for declaring a globally scoped variable.

We'll also go ahead and pass one more "Set or Change a Variable" action. Name this new variable GAME_WON. Just like before, set the Value Type attribute to Variable and give it a value of false. Now that we have the necessary variables, let's get to work on making this box move. Select your platform1 node and give it the behavior "Behaviors Triggered by Events >> On Proximity Do Something". This is perhaps one of the most useful pre-made items in the CopperCube toolkit. Proximity triggers allow you to create complex and dynamic scenes with various camera angles, cutscenes and events.

Hopefully you've got that proximity behavior ready. Change the "TestArea" attribute to "Box". Set the "Size" attribute to "10, 15, 10". We want the proximity trigger to be slightly taller than our platform. This is our window to press a button for jumping. Change the "Near to what" attribute to "A scene node", and finally open "Near to which" with the "..." icon to select your player node. Before we move on to the next step, please make an identical trigger on the exact same platform. Change the second behavior's "Triggered when" attribute to "Leaves Radius". We now have a trigger with an Enter and Exit function. The last thing to do is pass some actions to our proximity triggers.

We could just use the visual scripting actions to change our ON_GROUND variable, but I want to familiarize you with Javascript so that you can do more advanced things without it being too complicated. So we'll manipulate our variable by passing a new "Execute Javascript" action. All we want to do is change the value of the variable, so type in:

```
ccbSetCopperCubeVariable("ON_GROUND", true);
```

As you can see, we have the CopperCube API function to change our CopperCube variable. This is essentially the same as using a "Set or change a variable" behavior, but we can use it with more complex conditions like "if" statements and "for" loops. We'll talk more about for loops in a while.

Now we need to do the same thing in reverse for our Exit trigger. Pass a new "Execute Javascript" action to your "Leave Radius" behavior. This new script will look almost identical to the previous one, but with a couple of minor changes. So go ahead and edit your new Javascript with:

```
ccbSetCopperCubeVariable("ON_GROUND", false);
```

As you can see, all we've done is change one word. We set our ON_GROUND variable to false. But this is meaningless unless you can actually see it work, so we'll do a little test to make sure our variables are changing. Navigate to the "Create" tab of the toolkit, and create a new "Path" item. Click the center point of the path, and drag it below the platform. Now edit both of your Proximity behavior Javascript actions to say:

```
let x = ccbGetCopperCubeVariable("ON_GROUND");
print(x);
```

The last thing we need to do is make the player cube follow the path. To do that, give it a "Follow a path" behavior. If you've done it correctly, your player cube will now hook to the farthest right point of the path, and it will fly below the platform and then back up to the left. If you run your program now, it should be printing "true" and "false" every time the cube enters and leaves the radius of the platform. If you aren't seeing this result, then you've missed a step somewhere so please go back and check.

Now that you've seen your work in action, let's make this game into a game. Save a new .ccb project file before moving on. The next step is to remove the "Follow a path" behavior from our player node. Once you've done that, go ahead and delete the path since we don't need it anymore. Give your player node a new "Collide when moved" behavior. This behavior will give us our gravity. Set the"Size" attribute to "2.5, 2.5, 2.5". Set the "Relative position" attribute to "0, 0, 0". If you run your program now, you'll see the player box fall and land on the platform.

We won't dive into designing a new control scheme yet. That can be somewhat complicated. For now, just give your player node the "Scripted Behaviors >> 2D Jump'n'run player" behavior. Set the "Jump Speed" attribute to" 0.15". Set the "Jump Length" attribute to "300". It isn't perfect yet, but don't worry. We aren't trying to make it pretty yet. All we want is something functional enough to design more of the level.

Before we get into making a level though, we need to make sure the camera is following the player. Delete that simple camera that we've been using for so long. Create a new "Third Person Style Controlled" camera and select your player node as the camera target. Set your viewport perspective to "Left" and move the camera to the left of your player node. Change the "3rd Person Camera" behavior's "Follow Mode" attribute to "Follow From Behind, fixed rotation". This will make the camera follow the player node without swinging around or moving in a disorienting way.

Now let's make something resembling a game level. Select your platform1 node and set the "Scale" attribute to "3, 0.1, 1". Right click on that node in the SceneGraph explorer and select "Clone". Move your platform a little to the right, and leave a small gap between both platforms. Run and test your game until you can comfortably jump from one platform to the other. Once you've finished positioning the platforms, turn your attention back to the "Create" tab in the toolbox. We're going to create a couple of new blocks to give the level itself some more functionality.

First, create one new block with a default size. Set the “Scale” attribute of the new block to “20, 0.1, 1”. Name this block “deathBox”. Our deathBox here will act as a trigger to respawn the player when they fall. Speaking of which, we need to create another box with a size of 5. Name this one “spawnBox” and position it where you want the player to respawn. Disable the “Collision” attribute and disable the “Visible” attribute. Now we have a death window and a respawn point, so let’s make them work how they’re supposed to.

Go ahead and give the deathBox a new “On Proximity” behavior. Make the shape a box, and set the “Size” attribute to “20, 300, 20”. Position the box so that your proximity trigger is slightly below your level. Set the “Near to What” attribute to “a Scene Node” and select your player node. Pass in a new “Execute Javascript” action to your new proximity trigger, and get ready to type in this block of code;

```
let die = ccbGetSceneNodeFromName("spawnBox");
let char = ccbGetSceneNodeFromName("player");
let pos = ccbGetSceneNodeProperty(die,'Position');

ccbSetSceneNodePositionWithoutCollision(char, pos.X, pos.Y, pos.Z );
```

If you’ve been paying attention up until this point, then you probably already know what’s happening here. I’ll break it down again for the sake of cementing these ideas. First, we declared three new local variables called “die”, “char”, and “pos”. Those variables are now equal to the values returned by each function. We can now use those values to set the player node position to the exact position of the spawn block. You might have noticed that pos is followed by a .X, .Y, or .Z. Those “methods” are equal to the individual X, Y and Z axis. If we don’t specify the axis, the variable will assume that we mean all 3. The final function does not take two arguments, it will only accept four. So we must distinguish between the X, Y and Z axes manually.

If you run your program now and jump off of the edge, you will respawn at the point of your spawn block. If you aren’t respawning, you may have missed a step. You can use multiple spawn blocks and death boxes throughout your level, but later we will use a little bit of code to make a single death box respawn at multiple hitboxes. For now, be proud of what you’ve accomplished here.

We've reached the end of this chapter, and hopefully you've been paying attention. Like the last few chapters, I have a couple of exercises prepared for you to test what you've learned so far. Hopefully you've been following along and paying attention. DON'T FORGET TO SAVE A NEW PROJECT FILE!

-Create a "Cone" primitive that will respawn the player when touched.
===If you feel confident, make it work so that it only respawns the player if they jump on it. We already defined a variable for this.
-Create a second deathBox and a second spawnBox. Make them functional.
-Create an infinitely falling platform and an infinitely rising platform
===Trust me, you know how to do this.
-Assign a unique texture to your player and your platforms