Learning Construct 2

Design and create your own engaging, extensible, and addictive game using Construct 2

Aryadi Subagio
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 "Making a Breakout Clone"
- A synopsis of the book’s content
- More information on Learning Construct 2

About the Author

Aryadi Subagio is the second among seven brothers. He has loved video games ever since he was little, and he wanted to make his own game in his teenage years. This led him to pursue a Diploma in Computer Programming after he graduated from high school, where he spent years learning about software development techniques. He learned about game development from online sites, because at that time, there was no book on game development in Indonesia.

After graduating, he immediately jumped into the world of game development. During this time, he made a few shooting games for the Flash Platform. After a year, he decided to gain more experience by working in a game studio; this is when he joined Esabra Studio. After working at Esabra Studio, he worked at Enthrean Guardian, a game studio based in Semarang. He has now quit working and is trying to set up his own studio.

Aryadi has a passion for sharing his knowledge. He has written a lot of blog posts about game development on Indonesian and English sites. He also shares his knowledge on his own personal blog at http://daggio21.blogspot.com/.

I would like to thank my mother who supported me while I was writing this book, my brother for letting me borrow his laptop sometimes, and my friends who encourage me.
Learning Construct 2

Construct Classic was released in 2007. It was initially created as a hobby project by a group of students working in their spare time. However, the project was really buggy at first, with the team spotting many flaws in the design of Construct Classic, such as absence of support for platforms other than Windows. The team decided that supporting Construct Classic was not a good decision and decided to halt development in April 2013. The entire source code was published on SourceForge, and people can still freely access it if they want, although now there is no official support being provided. Construct 2 was already developed in 2011 with major design changes in mind. One of the design changes proposed was to use HTML5 as the technology behind the tool instead of DirectX that was used by Construct Classic, which allowed the software to support lots of different platforms.

Construct 2 makes it easy for people to make 2D games, regardless of their background. It comes with a complete set of powerful features, has the ability to support multiple platforms and app stores, and possesses an easy-to-understand visual programming system. It is also extensible using a plugin system, with lots of plugins being developed by the community to extend the capabilities of Construct 2.

Learning Construct 2 will introduce you to Construct 2's interface and workflow, and at the end, it will provide you with the skills and knowledge you need to develop your own games, even if you don't know programming at all. This book will guide you through the features of Construct 2, and it uses Construct 2 to create features that are available in popular games, for example, physics, high scores, and AI.

What This Book Covers

Chapter 1, Downloading and Understanding Construct 2, introduces you to Construct 2. This chapter will make you familiar with the interface and terms that Construct 2 uses, as well as give you a quick overview of the event system.

Chapter 2, Creating Your First Game Design, will teach you what you need to know about game designing. We will start with the definition of a game, make challenges and rewards, and finally move on to create our game design document.

Chapter 3, Creating Diverse Player Experiences with a Flappy Bird Clone, will examine what makes Flappy Bird addictive and how we can attempt to make a similar experience. We will use a technique called procedural generation to create random objects.

Chapter 4, Making a Breakout Clone, will teach you how to modify gameplay elements. We will use instance variables to modify the state of game objects.
Chapter 5, *Making a Platformer Game*, will teach you about the physics engine inside Construct 2, how it behaves, and how you can use it to incorporate physics in your game objects.

Chapter 6, *Creating a Space-shooter Game*, will teach you how Construct 2 stores data locally and how to read this data in a game. In this chapter, we will create a leaderboard in our game to demonstrate it.

Chapter 7, *Making a Battle Tank Game*, will teach you about the basics of AI and how to make an enemy object with its own AI. We will make an AI that makes the enemies shoot at the player when they see the player.

Chapter 8, *Debugging Your Game*, will teach you about events that usually cause bugs and how to avoid them. Alongside this, we will use debugging features inside Construct 2.

Chapter 9, *Mastering the Best Practices*, will teach you about the best practices in game development in general and in Construct 2, revealing some techniques that developers use to make their game in an efficient way.

Chapter 10, *Publishing Your Game*, will teach you how to export your game to the Web, desktop, and mobile platforms.
Making a Breakout Clone

We created our first game in the previous chapter. If you still remember what we did there, we used instance variables to do two things: flap the plane and make the ground scroll. We used only Boolean type instance variables to do what we want, while there are still two other variable types.

These other types can be used to manipulate the state of an object; if you ever see Mario becoming big after eating a mushroom or becoming invincible after getting a star, then you've seen the example of changing the state of an object. We will create something similar in this chapter.

In this chapter, we will cover the following topics:

- More ways to use instance variables
- What a constant variable is
- What an expression is and how to use it
- How to make an object bounce off other objects
- What sub-events are
Creating the design

Just like the previous chapter, we will start this chapter by making a design of the game we want to make. You probably would have guessed by reading the chapter title that we are going to make a breakout clone. *Breakout* is a game where you move a paddle in the bottom part of the screen to bounce a ball up to destroy blocks. However, some blocks will be able to add some variation to the gameplay. Let's start with the following criteria:

- **Elevator pitch**: An elevator pitch is the description of your game as you would explain it to someone in an elevator, short and concise. This game is a game where players move a paddle to bounce balls off to destroy blocks. Some of the blocks will have special effects to change the balls or the paddle.
- **Theme**: The theme for this game is puzzle.
- **Genre**: The genre for the game is puzzle game.
- **Challenges**: These are used to destroy all of the blocks in a level.
- **Rewards**: Players are given scores depending on what blocks are destroyed and how long they take to complete a level; some blocks will also change the score.
- **Winning condition**: All the blocks are destroyed.
- **Losing condition**: Each time a ball falls below the game screen, the player loses a life; if the player loses all three lives, then they lose the game.
- **Dynamics**: Dynamics are the rules that apply in the game; they form the instructions on how the game is played. For our game, the dynamics are as follows:
  - The ball bounces off the paddle, the blocks, and the three sides of the game area
  - The ball doesn't bounce in the bottom part of the game area
  - The paddle can only move to the left and to the right
  - Some of the blocks will produce a special effect when destroyed
- **Main game flow**: This defines how the game plays from beginning to end, from the start of the level until either the player wins or loses:
  - The ball moves downward at an angle at the start of the level
  - If the ball falls into the bottom part of the game area, the player loses a life
Chapter 4

- If the ball bounces off one of the blocks, that block is destroyed
- For some blocks, a special effect will occur if it's destroyed
- Some blocks will need to be bounced off more than once to be completely destroyed
- If all the blocks are destroyed before the player loses, the player wins

This is pretty much our design for a breakout clone; we will use this design to create our example game for this chapter.

**Designing the reward system**

Now, before we actually get our hands on Construct 2, I want to lay out the design of this game in more detail. First, I want to explain the way in which this game rewards the players. The idea I had in mind was to make each block contain a value so that every time it's destroyed, the score will be added by the same value. However, as the number of blocks in a level will be the same for everyone playing, which would result in everyone getting the same score, I decided to add an element to the final score calculation that would vary the score. So, I added the time it takes for a player to finish a level to the score calculation formula.

However, this is still not enough. What if there are people who cleared a level at the same time? Or is it only slightly slower than the other? The gap between each score will be so small that it will be easy to beat the higher ranking player. Also, the skilled players will eventually be able to beat the level in the fastest possible time, making a score limit to the game. To prevent this, I decided to give the game special power ups that can increase the scores. These power ups should appear randomly, and once they appear, they should disappear within a few seconds.

**Setting up our layout**

Now, we will set up our layout for the game. Unlike the previous chapter where I explained how to add game objects in detail, in this chapter, I will only tell you to add objects when I want you to add them to the layout. I'll only give more details when I'm explaining something new. You can download the sample code from the book's website.
We will use the sprites from the `freebundle.zip` file, which we downloaded earlier. However, this time, we will use the sprites present in the `Puzzle assets` folder under the `Sprites` folder. Open up Construct 2 and create a new empty project. Just like we did the last time, we will add layers to this layout, but now, we will create only three of them:

- HUD
- Main
- Background

Keep HUD as the top layer and Background as the bottom one.

Add a sprite game object to the Main layer and use the paddleBlue sprite from the `Puzzle assets` folder; this is going to be our paddle. Then, add another sprite object that will be our ball; use ballBlue as the sprite. Now, we have a paddle and a ball in the layout. Next, we will create the border for our level, but instead of using one long sprite to do this, we will create the border by lining several small sprites. This is a commonly used technique when making games.

Making small sprites that will then be lined together is more efficient than making one long sprite, because if you want to make another object that uses the same sprite (for example, with a longer border), you don't need to redraw or stretch it. This will decrease the sprite quality; just line the sprites, and you will get a longer border.

In Construct 2, there's a way of lining a few sprites together, without making the job difficult; we can do this using a tiled background object.

### Using a tiled background object

A tiled background object is just like what the name implies. It is a background made up of tiled sprites. If you want to reuse the same sprite over and over, then using one tiled background object is better than making plenty of instances of a sprite object for two reasons:

- **It's easier**: To make new tiles, you just need to drag the tiled background object around and it will create new rows and columns for the tiles; no need to make new instances of a sprite. It's also easier to move the object, if you need to.
• **It counts as one object**: As opposed to multiple instances of an object, this makes it easier for Construct 2 to manage the object technically under the hood and prevents your game from lagging.

Apart from being used as the game area's border, a tiled background object can be used to place tiles in a top-down RPG / adventure game, or it can be used to construct levels in a platformer game.

Add a new game object, select a tiled background from the **General** section, and rename it as `areaBorder`. You will see an edit image window much like the one you see when you want to add a sprite game object; the difference here is that there's no animation because you can't add animations to a tiled background. You also can't set origins and collision polygons for it. Use `element_grey_square` for the sprite; this is usually for game blocks, but we don't have the appropriate sprite for a border, so we'll use this instead.

One thing to note is that you can't use any image for the tiled background's sprite; it has to be a square image with a power of 2 as its width and height. A power of 2 in mathematics is $2^n$, where $n$ can be any number. So, it's 8 x 8 squares, 16 x 16 squares, 32 x 32 squares, and so on.

Close the edit image window, and we have a tiled background in our layout. You can click-and-drag one of the eight points that surround the game object (you can see it when the object is selected), and instead of making the image bigger, you can actually add more rows and columns to it.

Using images with the size of power of 2 is also a good habit when creating sprites and assets used in game development. It is a great optimization trick.

Perhaps you've realized this, but when you drag one of the eight points to add more tiles, the newly added tiles are not always at full size, but only a portion of it.
Making a Breakout Clone

If this is the kind of image we want, there's no problem, but often, this is not the case. We want the object to add new rows and columns exactly by its tile's width or height. Luckily, there's an easy way to do this in Construct 2: by snapping it into grids. Perform the following steps:

1. In the upper part of Construct 2, you'll see three menus near the File menu: Home, View, and Events.
2. Click on the View menu and select the Snap to grid checkbox to snap object placements and resize to the grid.

Which grid? If you want to see the grid, you can select the Show grid checkbox. This setting only works per layout. So, if you want to do the same to other layouts, you have to check it manually on each layout.

![Menu Options]

To make it easy to place objects in the layout, you can set the layout to Snap to grid. Doing so will make object placements and resizing follow the grid.

Now, move areaBorder and create more instances of it until it makes a game border as follows:

![Game Border]

Now that this is done, we can move to the next part of the game: moving the paddle.
Moving the paddle in only two directions

Moving the paddle isn't actually a difficult thing; we can do this using a movement behavior like we did in Chapter 3, Creating Diverse Player Experiences with a Flappy Bird Clone. The important thing is that we only want the paddle to move in two directions: left and right. I will explain how to do this by introducing you to another game object: the Mouse object.

Insert a new game object as usual, but select the Mouse object from the input category. It will be added to the entire project, so you won't see anything new on the layout. We want the paddle to follow the mouse; if the mouse moves to the left, then the paddle will move to the left, and the same will happen if the mouse moves to the right. However, we only want the paddle to move along the $x$ axis. So now, switch to the event sheet and add the following line of code:

This will make our paddle follow the mouse on the $x$ axis only. If you test it now, the paddle will move like we want it to, only in two directions. However, there's something wrong with it; it follows the mouse even until outside of the game border. This is something we don't want, so we will stop this from happening. To prevent this, we will simply say, "Don't go to the right if you have reached the position I decide." and "Don't go left if you have reached the position I decide." to the paddle. If you translate them into code language, the first event would be as follows:

The second event would be as follows:

Try testing it now, and now, the paddle only moves inside the game border. We have finished making our paddle move; the next thing we're going to do is move the ball.

Making the ball bounce

We will do two things: move the ball automatically at the start of a level and make it bounce off objects. These two things can be accomplished using a movement behavior we learned in the previous chapter. We will add a Bullet behavior to the ball so that it always moves in one direction. However, making it bounce can't be done just by the Bullet behavior alone.
If you take a look at the **Bullet behavior** properties in the **Properties** bar, you can see that one of its properties is to bounce off solids, which defaults to **No**. By switching this to **Yes**, we can make an object with Bullet behavior bounce off solids, but what actually is a solid?

A **solid** is just another behavior that can be added to a game object. An object with such behavior is considered a **solid** object, which means other game objects can't walk past it, can't jump through it, and so on. Since we want to make the ball bounce off the paddle and the game border, we'll add this behavior to those objects.
After adding the Solid behavior to the paddle and area border, we will have to change the angle of the ball a little so that it moves at an angle at the beginning of the level. So, set it to 45 degrees to make it move to the lower-right angle (or you can set it to any angle you want), and after setting the bounce off solid property to true, we are all set. Test the game now, and the ball should bounce on the paddle and off the borders; what's missing now are the blocks.

Adding the blocks

Now comes the time when we need to add the blocks to the game area. Before adding the blocks, let's remember that they have several characteristics such as the following ones:

- They give scores when destroyed
- Most of them are destroyed when they collide with the ball once; others need to be bounced off more than once
- Some of them have special effects when destroyed

We must keep these in mind when making our blocks. There are several block shapes we can use in the Puzzle assets folder from freebundle.zip, but for simplicity, we'll only be using the rectangle shape here. We'll classify them based on their colors as follows:

- Blue blocks are the normal blocks; they don't give special effects, and they are destroyed on the first collision with the ball
- Purple blocks are the stronger blocks; they need to be collided into twice before they are destroyed, and they don't have special effects
- Green blocks are destroyed on a single collision, but they give a score bonus
- Red blocks are also destroyed on a single collision, and they give additional life
- Yellow blocks give a time bonus on being destroyed

Now that we've planned it like this, let's add blocks to our layout. Insert a new sprite object with blocks with its name, and in the edit image window, load a blue rectangle sprite. We'll also load the other colors in this object so that we can easily set it up and, probably, change it when the game is running. We can do this by adding new animations to this object.
Making a Breakout Clone

Right-click on the Animations window and select Add animation to add a new animation, rename it to Purple, and give a purple rectangle image to this animation. Ensure that you click on this animation first before adding an image to it because Construct 2 will automatically select the default animation after adding new ones. Repeat this for the rest of the colors until you have all the required animations.

Close the edit image window, and we'll have our block's object in the layout. Next, we'll add three instance variables to implement the characteristics of this object:

- First, add an instance variable named score with a default value of 100; this will add to the score every time this block is destroyed
- Second, add a health instance variable with a default value of 1; this determines how many times the block must be hit with the ball before it is destroyed
- Finally, add an effect instance variable and leave the initial value as 0; we will manipulate this later to add special effects when the block is destroyed
Destroying the blocks

You may think it's starting to get difficult, but don't give up because I'm going to explain everything it in a way that is easy to understand. Now, we can use instance variables to determine which block to destroy, but first, let's put the blocks in place in the layout as follows:

We will first make the blocks collide with the ball and destroy them. Just like the areaBorder object, we will simply add a Solid behavior to the blocks. This will make the ball bounce off the blocks, but it won't destroy them. To destroy them, we will subtract their health every time the blocks collide with the ball, and after that, we'll check if their health has gone to 0; if it has, then we will destroy them. Translate this into code language, and you will get two new lines of code. The first one is as follows:

The second event will be as follows:

If you test it now, you can see the basic gameplay of the game: you can move the paddle to bounce the ball in order to destroy the blocks.
Making a Breakout Clone

Setting up the power up blocks

Now, we'll set up the power up blocks at the beginning of a level. There are two ways to do this: set it randomly or define it manually. We'll do it manually in this example, because this kind of game is usually level-based. To do this, we'll first add an instance variable to the blocks called `blockColor` (it's actually a Text variable) and leave the initial value to empty. We'll use this instance variable to determine what kind of block it is.

After adding this instance variable, close the instance variable window and select the block whose color you want to change. Then, in the Properties bar, change the value of `blockColor` to a color you want. The color must be one of the animations we set earlier, and when you set a Text variable in the Properties bar, you don't need to type the quote symbol. If you set a purple block, don't forget to change the `health` variable as well, so it needs two collisions to destroy it.

<table>
<thead>
<tr>
<th>Instance variables</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>scores</td>
<td>100</td>
</tr>
<tr>
<td>health</td>
<td>2</td>
</tr>
<tr>
<td>effect</td>
<td>0</td>
</tr>
<tr>
<td>blockColor</td>
<td>Purple</td>
</tr>
<tr>
<td>Add / edit</td>
<td>Instance variables</td>
</tr>
</tbody>
</table>

However, just because we change an instance variable, it doesn't mean the blocks' animation will immediately change. We need to add something on the event sheet. We want the game to check the `blockColor` instance variable at the start of the level and then change the blocks' animation to that color. So, we will simply add the following line:

```
Set animation to blocks.blockColor (play from beginning)
```

Perhaps, you've realized that there are blocks with their `blockColor` instance variable values empty, but their color is blue. This is because when Construct 2 doesn't find a matching animation, it reverts to the default color, which is blue.
Writing an expression

In the code we wrote previously, we used an expression to retrieve a value of an instance variable.

An expression is any legal mathematical combination that represents a value. All Construct 2 objects have their own expressions.

One of those mathematical combinations is a variable, so a single instance variable also counts as an expression. For example, you can retrieve an object's position from its $x$ and $y$ values. You can also perform mathematical operations in an expression, such as addition, subtraction, multiplication, and division.

Understanding constant variables

Now, we will start changing the game state. We will do this by utilizing two kinds of variables: instance variables and global variables. We'll make several global variables that will be checked later with an instance variable. However, we will change the global variable to a special type of variable: constant variable.

So, what are constant variables? They are variables, global or local, whose values do not change after they are declared. Both the text and number type variables can become constant variables. To make a constant variable, just select the Constant checkbox when creating a new global variable. Constant variables in the event sheet are indicated by their names, which appear capitalized.

Let's try it now. Go to the event sheet and create a few variables needed to create the blocks' special effect. As you remember, the default value for the blocks' effect instance variable is 0; so we'll make a value of zero when the block has no special effect at all. Then, we'll set incrementing values for the other special effects, as shown in the following screenshot:
This is the bonus that players get when a certain block is destroyed. So, let's change the value of the effect property of the blocks in the layout. Give a value of 1 to the green blocks, 2 to the red blocks, and a value of 3 to the yellow blocks.

**Adding sub-events**

Now that we've assigned values to the effect property, we can start adding code to the event sheet. The logic is if the block is destroyed, we'll check the value of its `effect` instance variable; if it is the same as one of the constant effect global variables, we'll change the value of a variable depending on the effect. To do something when an object is destroyed, we will use the `On destroyed` event. Let's add an `On destroyed` event to the event sheet.

Now, we want to compare the value of an instance variable after an `On destroyed` event; this means that we want to create an event inside another event, or in other words, we want a sub-event. To create a sub-event, first select the event you want to make a sub-event from and then press S on your keyboard. A new window will open up; here, you can select a new event for the sub-event. Create your sub-event until it looks like this:

![Event Sheet with sub-events]

Once again, because adding an image every time I teach you a code will be troublesome, this is how I write it in the code form:

```plaintext
blocks | On destroyed  
blocks | effect = EFFECT_SCORE_BONUS  
blocks | effect = EFFECT_LIFE_BONUS  
blocks | effect = EFFECT_TIME_BONUS
```
Note that there's no limit on how many sub-events can be made inside an event. You can create a sub-event inside a sub-event, but it is suggested that you do not make too many sub-events because the code will be too confusing to understand.

Now, to make these blocks really work, we need to make three global variables: \texttt{score}, \texttt{life}, and \texttt{gameTime}; all are number types. For life and time bonuses, we want them to increase after a fixed number; life will increase by 1, and time will increase by 10 seconds. For score bonuses, we'll use a simple formula to make it vary per player. The score bonus will be based on the player's life and the current time remaining; the code will be as follows:

![Table](image)

We have successfully added special effects that give a bonus when the blocks are destroyed. Now, what if we want to change something about the paddle when a block is destroyed?

**Changing a game object's state**

Other than adding bonuses to the game, we can make a special effect that changes the state of an object. In this example, we will change the state of the paddle. While changing the state of the paddle, we will determine the duration of the new state, and after this duration ends, the paddle will revert to its original state. To make it simple, the duration of this new state stays until the player destroys another block that doesn't have a special effect. For now, we'll make the paddle wider.

To do this, we'll create a Text type instance variable called \texttt{state} on \texttt{paddleBlue}. The value of this instance variable will determine the state of the \texttt{paddleBlue} object. Now, we need a place to change it; let's change it when the purple block is destroyed, because right now, nothing happens when we destroy it.
To select a purple block, we'll take a look at its `blockColor` instance variable; if it says purple, then we got the block we want. After that, we can change the state of this block. One more thing we want to do is change the `paddleBlue` state back to its default value. We can do this by checking both the effect and the `blockColor` instance variable to find a block that isn't purple and has no effect. So, we will add these sub-events to the latest written code:

<table>
<thead>
<tr>
<th>Blocks</th>
<th>On destroyed</th>
<th>Add action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks</td>
<td><code>effect = EFFECT_SCORE_BONUS</code></td>
<td>Add life <code>gameTime * 10</code> to score</td>
</tr>
<tr>
<td>Blocks</td>
<td><code>effect = EFFECT_LIFE_BONUS</code></td>
<td>Add 1 to life</td>
</tr>
<tr>
<td>Blocks</td>
<td><code>effect = EFFECT_TIME_BONUS</code></td>
<td>Add 10 to <code>gameTime</code></td>
</tr>
<tr>
<td>Blocks</td>
<td><code>blockColor = &quot;Purple&quot;</code></td>
<td>Set state to <code>&quot;wider&quot;</code></td>
</tr>
<tr>
<td>Blocks</td>
<td><code>blockColor = &quot;Purple&quot;</code></td>
<td>Set state to <code>&quot;&quot;</code></td>
</tr>
</tbody>
</table>

We have changed the state of the `paddleBlue` object; now, to make this really work, we need to add two more events: one when the state instance variable's value is wider and another when it is an empty string. We want to make it wider when the state is wider and return the width to normal when it is an empty string. Note that the `paddleBlue` object's normal width is 104 pixels wide, and we want to make the width 184 pixels wide when it's wider (or you can set your own value). The first event will be as follows:

<table>
<thead>
<tr>
<th>← paddleBlue</th>
<th>state = &quot;wider&quot;</th>
<th>← paddleBlue</th>
<th>Set width to 184</th>
</tr>
</thead>
</table>

The second event will be as follows:

<table>
<thead>
<tr>
<th>← paddleBlue</th>
<th>state = &quot;&quot;</th>
<th>← paddleBlue</th>
<th>Set width to 104</th>
</tr>
</thead>
</table>

If you test the game now, you can see that the paddle becomes wider after the purple block is destroyed and reverts to normal when the blue block is destroyed, but the paddle stays wide when the destroyed block is not blue.
Adding more states

You already know how to change a state of an object and revert it to its original state. However, what happens when you add state-changing blocks? We’ll answer this by giving another state to the paddle. First, let’s add another animation called Grey to the block’s object; give this animation a gray rectangle sprite.

Just like we did earlier, for some blocks, change the blackColor instance variable’s value to Grey. We want to change the angle of the paddle after the player destroys this block, so first, we will add another sub-event when the block is destroyed.

Just like we did earlier, to actually make the state have an effect, we will make another event that checks the paddleBlue state. This new event will set the angle of the paddle if the state value is angle:

Don’t forget to change the angle back to 0 degrees when the state is empty again:

Test your game now; if you destroy the purple block and then the grey block, or the other way around, you’ll find that the paddle becomes big and tilted at 45 degrees. This is because two effects of state-changing blocks are applied.
Making a Breakout Clone

As you can remember, we only changed the width and angle of the paddle back to its original value when the state instance variable is an empty string. This time, the state instance variable didn't have a chance to become empty before the state change, thus applying two state changes at the same time.

When making your game, always look out for small gotchas like this. This is because Construct 2 doesn't always know what you want to do, which is a common thing when making a game, no matter what tool you use. This usually happens when you change the value of a variable.

To fix this, we need to deactivate other state-change effects when we switch to a new state. We'll set the angle to 0 when we make the paddle wider, and we'll change the width back to the original one when we tilt the angle. So, we'll add two new actions to two different events. The first event will be as follows:

<table>
<thead>
<tr>
<th>paddleBlue</th>
<th>state = &quot;wider&quot;</th>
<th>paddleBlue</th>
<th>Set width to 104</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>paddleBlue</td>
<td>Set angle to 0 degrees</td>
</tr>
</tbody>
</table>

The second event will be as follows:

<table>
<thead>
<tr>
<th>paddleBlue</th>
<th>state = &quot;angle&quot;</th>
<th>paddleBlue</th>
<th>Set angle to 45 degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>paddleBlue</td>
<td>Set width to 104</td>
</tr>
</tbody>
</table>

Ending a game

We have done a lot of things in this chapter, and I have explained many things. I am afraid this chapter is a bit hard to understand. Don't give up, and don't stop because we are nearing the end of the chapter. Now that we have added our state-change feature, it is time to apply a losing condition to the game. There are two losing conditions in this game: when the player loses all lives and when the time runs out. To do this, we'll give default values to the global variables related to the losing conditions. We'll give three lives and start the time countdown at 60 at the start of the level, like this:

<table>
<thead>
<tr>
<th>System</th>
<th>On start of layout</th>
<th>System</th>
<th>Set gameTime to 60</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add action</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>Set life to 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add action</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Then, we'll represent these values on screen like we did in our first game. So, we'll create two new Text objects to be put on the HUD layer, name them txtTime and txtLife, and make their values show gameTime and life instance variables as follows:

<table>
<thead>
<tr>
<th>System</th>
<th>Every tick</th>
<th>txtTime</th>
<th>Set text to &quot;Time = &quot; &amp; gameTime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>txtLife</td>
<td>Set text to &quot;Life = &quot; &amp; life</td>
</tr>
</tbody>
</table>

To show that the player has lost, we will display GAME OVER text in large when one of the losing conditions is met. So, create another Text object called txtGameOver, add it on the HUD layer, give it the GAME OVER text, and change the font to make it bigger.

To change the size of a text, we will change the Font property on the Text object. When you click on the Font property, you'll see a small box with three dots on it; clicking on this will open a window where you can change the font:

In the new window, you'll be able to change the font type, font style, and font size. Set the new size to 36 and put it off screen. We'll also change the color of the text; to do this, click on the Color property of the text, and you'll see a small box on the right-hand side. Click on it, and a box with lots of colors to choose from will pop up. For now, make the GAME OVER text red. Move the txtGameOver object off the screen, because we will only need it when the player loses.

**Losing by time**

The first losing condition we want to make is losing by time. The logic is simple: if the gameTime global variable drops below 1, which can mean 0 or a negative value, then we'll show the GAME OVER text in the middle of the game area. We must also stop the movement of the ball, because we don't want it to keep moving and bouncing around when the player is declared lost. If we write this in code form, it would be as follows:

<table>
<thead>
<tr>
<th>System</th>
<th>gameTime &lt; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When you click on the gameTime < 1 action, you'll see a small box on the right-hand side. Click on it, and a box with lots of colors to choose from will pop up. For now, make the GAME OVER text red. Move the txtGameOver object off the screen, because we will only need it when the player loses.
I want to tell you something about how Construct 2 creates an object. When you want to create an object, this object must already be put in a layout. Any layout is fine; it doesn't have to be the layout that the object is created in. This is because when Construct 2 creates an object, it needs to know the default properties of the object. If it's a Text object, then Construct 2 needs to know the initial text value, color, font style, font size, and so on. It is advised that you have a separate layout specifically to place objects.

When creating an object, this object must be put somewhere in a layout. It is advised that you have a layout only to store game objects. Another reason to do this is because even off-screen assets in your starting layout will instantiate themselves when the layout started. If you're not careful, you can have a bullet dashed from one side of the screen to the other at the start of the game.

Now, the only thing that's missing is that we still haven't reduced gameTime; thankfully, Construct 2 has an event that does something every second (or every few seconds, depending on what you want):

### Losing by life

Now, we'll make the player lose by life. Players initially have three lives; every time the ball falls below the screen, we'll reduce the player's life by 1, and if the life goes down to 0, we can say that the player lost. The question is how does the game continue after the ball falls below the screen. The answer is: we will create a new ball.

So, the logic is like this: if the ball falls below the screen, we'll reduce the player's life by 1; if it is still above 0, then we'll create another ball to continue playing; and if it is 0, then the player loses. So, this is going to be as follows:
If you test it now, the game should be working as we want it to. However, there's still one thing missing: the score. So, let's make this.

**Calculating the score**

Calculating the score is really easy because we already have an instance variable in the `blocks` object that we can use to increment the score when it's destroyed. So, we'll basically just add the `score` global value every time a block is destroyed:

<table>
<thead>
<tr>
<th>System</th>
<th>On destroyed</th>
<th>Add blocks.scores to score</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Here, we will make a new block, an `On destroyed` event, besides the previous one we made. It's okay to make several events, which are the same as this one; this can be used to chop down the number of actions in an event to make the code easier to read and understand.

You can create two or more events from the same event. Use this to group actions and sub-events that do the same thing to make the code easier to read and understand.

After we have made the code, it's time to show the score. We'll show it when the level ends, that is, either the player wins or loses. We'll create a new `Text` object called `txtScore` on the HUD layer, and we'll put it off the screen somewhere. First, let's show this object when the losing condition is met. Create a new `txtScore` object when either `gameTime` is below 1 or when the player's life is equal to or less than 0. So, we'll add an action to two events.
Making a Breakout Clone

After creating the text score object, we will make it show its initial value. This initial value is not the one we set in its Properties bar, but will be based on the score global variable. As a result of this, we'll create its initial value at another place when it is created. Construct 2 has an On created event that fires when an object is first created; we'll set the initial value there:

Test the game now. You'll see that the Score textbox shows up along with the GAME OVER text when the player's life is 0. Next, we'll show the score when the player wins.

Comparing an expression

A player wins this game when all the blocks in a level are destroyed. To know whether all the blocks have been destroyed or not, we will count the number of instances of the block. If it's 0, then we know that all the blocks have been destroyed. Construct 2 has an expression to count the number of instances of an object; to compare an expression instead of a variable, we'll use the system's compare two values event, and create a condition "System blocks.Count = 0".

Also, we only want to count the blocks when the game is going on, not after the game is over. So, we'll add another condition that will serve as a marker when the game is over. We want to stop the movement of the ball after the player wins, so let's use this as a marker and set the ball's speed to 0. We'll stop the ball's movement when the player wins; with actions and conditions combined together, our code will look like this:

Play the game until you win, and you can see that the score is shown as we wanted it to. The ball has also stopped moving, so we don't have to worry about it falling below the game area after we win.
Killing the bug

Well, we've got a perfect working small game. Awesome, right! However, just like any other game, this one potentially has bugs. There are two bugs here: the timer doesn't stop when the game ends, and the timer can count down to below 0. We can fix the first bug by making sure that we only subtract the gameTime variable while the game is still on, in other words, when the player's life and number of blocks are more than 0.

The second bug is fixed simply by changing the text value when gameTime is less than 1. The resulting code is as follows:

```
<table>
<thead>
<tr>
<th>System</th>
<th>Every 1.0 seconds</th>
<th>Subtract 1 from gameTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>life &gt; 0</td>
<td>Add action</td>
</tr>
<tr>
<td>System</td>
<td>blocks.Count &gt; 0</td>
<td></td>
</tr>
<tr>
<td>System</td>
<td>gameTime &lt; 1</td>
<td></td>
</tr>
</tbody>
</table>

```

Summary

In this chapter, you learned about a new object, the tiled background, and how to use it. You learned about a new Solid behavior and how to use it to bounce off objects. You also learned about expressions, how to use them, and how to compare them like instance variables. You learned about constant variables and how to use them to change the state of an object. Moreover, you also learned about sub-events for the first time.

In our next chapter, I will introduce you to the Physics object and a few things you can make with it.
Where to buy this book

You can buy Learning Construct 2 from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.