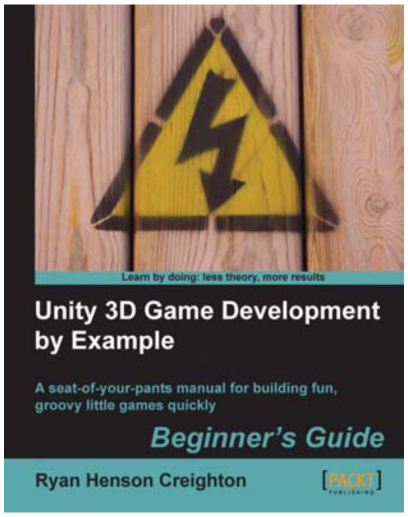
Unity 3D Game Development by Example

Beginner's Guide

Ryan Henson Creighton


Learn by doing: less theory, more results
Unity 3D Game Development by Example
A seat-of-your-pants manual for building fun, groovy little games quickly
Beginner's Guide
Ryan Henson Creighton
PACKT

Chapter No.7

"Don't Be a Clock Blocker"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.7 "Don't Be a Clock Blocker"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Ryan Henson Creighton is a veteran game developer, and the founder of Untold Entertainment Inc. (`http://www.untoldentertainment.com/blog`) where he creates games and applications for kids, teens, tweens, and preschoolers. Prior to founding Untold, Ryan worked as the Senior Game Developer at Canadian media conglomerate Corus Entertainment, creating advergames and original properties for YTV and Treehouse TV using Flash. Ryan is hard at work developing a suite of original products with Untold Entertainment. He maintains one of the most active and enjoyable blogs in the industry.

When Ryan is not developing games, he's goofing off with his two little girls and his fun-loving wife in downtown Toronto.

Big thanks to Cheryl, Cassandra, and Isabel for their love, their support, and their cinnamon rolls. Thanks to Jean-Guy Niquet for introducing me to Unity; to Jim "McMajorSupporter" McGinley for help with the book outline and ongoing mentorship; to the technical reviewers and Packt staff for letting me leave a few jokes in the book; and to David Barnes, for having such a great sense of humor in the first place. Special thanks to Michael Garforth and friends from the #Unity3D IRC channel on Freenode. I also want to thank Mom, God, and all the usual suspects.

Unity 3D Game Development by Example Beginner's Guide

"Game Developer" has rapidly replaced "firetruck" as the number one thing that kids want to be when they grow up. Gone are the days when aspiring developers needed a university education, a stack of punch cards, and a room-sized computer to program a simple game. With digital distribution and the availability of inexpensive (or free) games development tools like Unity 3D, the democratization of game development is well underway.

But just as becoming a firetruck is fraught with perils, so too is game development. Too oft en, aspiring developers underestimate the sheer enormity of the multi disciplinary task ahead of them. They bite off far more than they can chew, and eventually drift away from their game development dreams to become lawyers or dental hygienists. It's tragic. This book bridges the gap between "I wanna make games!" and "I just made a bunch of games!" by focusing on small, simple projects that you can complete before you reach the bottom of a bag of corn chips.

What This Book Covers

Chapter 1, That's One Fancy Hammer!, introduces you to Unity 3D—an amazing game engine that enables you to create games and deploy them to a number of different devices, including (at the time of writing) the Web, PCs, iOS platforms, and WiiWare, with modules for Android and Xbox Live Arcade deployment in the works. You'll play a number of browser-based Unity 3D games to get a sense of what the engine can handle, from a massively-multi player online game all the way down to a simple kart racer. You'll download and install your own copy of Unity 3D, and mess around with the beautiful Island Demo that ships with the product.

Chapter 2, Let's Start with the Sky, explores the difference between a game's skin and its mechanic. Using examples from video game history, including Worms, Mario Tennis, and Scorched Earth, we'll uncover the small, singular piece of joy upon which more complicated and impressive games are based. By concentrating on the building blocks of video games, we'll learn how to distil an unwieldy behemoth of a game concept down to a manageable starter project.

Chapter 3, Game #1: Ticker Taker, puts you in the pilot seat of your first Unity 3D game project. We'll explore the Unity environment and learn how to create and place primitives, add Components like physic materials and rigidbodies, and make a ball bounce on a paddle using Unity's built-in physics engine without ever breaking a sweat.

Chapter 4, Code Comfort, continues the keep-up game project by gently introducing scripting. Just by writing a few simple, thoroughly-explained lines of code, you can make the paddle follow the mouse around the screen to add some interactivity to the game. This chapter includes a crash course in game scripting that will renew your excitement for programming where high school computer classes may have failed you.

Chapter 5, Game#2: Robot Repair, introduces an oft en-overlooked aspect of game development: "front-of-house" User Interface design—the buttons, logos, screens, dials, bars, and sliders that sit in front of your game—is a complete discipline unto itself. Unity 3D includes a very meaty Graphical User Interface system that allows you to create controls and fiddly bits to usher your players through your game. We'll explore this system, and start building a complete two-dimensional game with it! By the end of this chapter, you'll be halfway to completing Robot Repair, a colorful matching game with a twist.

Chapter 6, Game#2: Robot Repair Part 2, picks up where the last chapter left off . We'll add interactivity to our GUI-based game, and add important tools to our game development tool belt, including drawing random numbers and limiting player control. When you're finished with this chapter, you'll have a completely playable game using only the Unity GUI system, and you'll have enough initial knowledge to explore the system yourself to create new control schemes for your games.

Chapter 7, Don't be a Clock Blocker, is a standalone chapter that shows you how to build three different game clocks: a number-based clock, a depleting bar clock, and a cool pie wedge clock, all of which use the same underlying code. You can then add one of these clocks to any of the game projects in this book, or reuse the code in a game of your own.

Chapter 8, Ticker Taker, revisits the keep-up game from earlier chapters and replaces the simple primitives with 3D models. You'll learn how to create materials and apply them to models that you import from external art packages. You'll also learn how to detect collisions between Game Objects, and how to print score results to the screen. By the end of this chapter, you'll be well on your way to building Ticker Taker—a game where you bounce a still-beating human heart on a hospital dinner tray in a mad dash for the transplant ward!

Chapter 9, Game#3: The Break-Up is a wild ride through Unity's built-in particle system that enables you to create effects like smoke, fi re, water, explosions, and magic. We'll learn how to add sparks and explosions to a 3D bomb model, and how to use scripting to play and stop animations on a 3D character. You'll need to know this stuff to complete The Break-Up—a catch game that has you grabbing falling beer steins and dodging explosives tossed out the window by your jilted girlfriend.

Chapter 10, Game#3: The Break-Up Part 2, completes The Break-Up game from the previous chapter. You'll learn how to reuse scripts on multiple different Game Objects, and how to build Prefabs, which enable you to modify a whole army of objects with a single click. You'll also learn to add sound effects to your games for a much more engaging experience.

Chapter 11, Game #4: Shoot the Moon, fulfills the promise of Chapter 2 by taking you through a re-skin exercise on The Break-Up. By swapping out a few models, changing the background, and adding a shooting mechanic, you'll turn a game about catching beer steins on terra firma into an action-packed space shooter! In this chapter, you'll learn how to set up a two-camera composite shot, how to use code to animate Game Objects, and how to re-jig your code to save time and effort.

Chapter 12, Action!, takes you triumphantly back to Ticker Taker for the coup de grace: a bouncing camera rig built with Unity's built-in animation system that flies through a model of a hospital interior. By using the two-camera composite from The Break-Up, you'll create the illusion that the player is actually running through the hospital bouncing a heart on a tin tray. The chapter ends with a refresher on bundling your project and deploying it to the Web so that your millions of adoring fans can finally experience your masterpiece.

7
Don't Be a Clock Blocker

We've taken a baby game like Memory and made it slightly cooler by changing the straight-up match mechanism and adding a twist: matching disembodied robot parts to their bodies. Robot Repair is a tiny bit more interesting and more challenging thanks to this simple modification.

There are lots of ways we could make the game even more difficult: we could quadruple the number of robots, crank the game up to a 20x20 card grid, or rig Unity up to some peripheral device that issues a low-grade electrical shock to the player's nipples every time he doesn't find a match. NOW who's making a baby game?

These ideas could take a lot of time though, and the Return-On-Investment (ROI) we see from these features may not be worth the effort. One cheap, effective way of amping up the game experience is to add a clock. Games have used clocks to make us nervous for time immemorial, and it's hard to find a video game in existence that doesn't include some sort of time pressure—from the increasing speed of falling Tetris pieces, to the countdown clock in every Super Mario Bros. level, to the egg timers packaged with many popular board games like Boggle, Taboo, and Scattergories.

Apply pressure

What if the player only has x seconds to find all the matches in the Robot Repair game? Or, what if in our keep-up game, the player has to bounce the ball without dropping it until the timer runs out in order to advance to the next level? In this chapter, let's:

- Program a text-based countdown clock to add a little pressure to our games

- Modify the clock to make it graphical, with an ever-shrinking horizontal bar
- Layer in some new code and graphics to create a pie chart-style clock

That's three different countdown clocks, all running from the same initial code, all ready to be put to work in whatever Unity games you dream up. Roll up your sleeves—it's time to start coding!

Time for action – prepare the clock script

Open your Robot Repair game project and make sure you're in the **game Scene**. As we've done in earlier chapters, we'll create an empty **GameObject** and glue some code to it.

1. Go to **GameObject | Create Empty**.
2. Rename the empty **Game Object Clock**.
3. Create a new JavaScript and name it **clockScript**.
4. Drag-and-drop the **clockScript** onto the **Clock Game Object**.

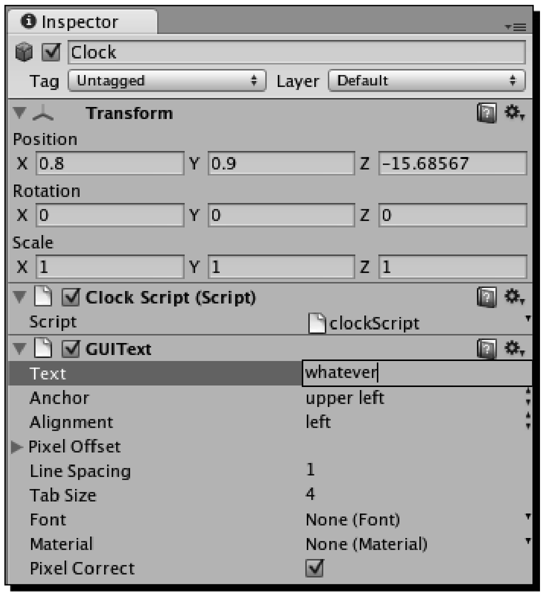
No problem! We know the drill by now—we've got a **Game Object** ready to go with an empty script where we'll put all of our clock code.

Time for more action – prepare the clock text

In order to display the numbers, we need to add a **GUIText** component to the **Clock GameObject**, but there's one problem: **GUIText** defaults to white, which isn't so hot for a game with a white background. Let's make a quick adjustment to the game background color so that we can see what's going on. We can change it back later.

1. Select the **Main Camera** in the **Hierarchy** panel.
2. Find the **Camera** component in the **Inspector** panel.
3. Click on the color swatch labeled **Back Ground Color**, and change it to something darker so that our piece of white **GUIText** will show up against it. I chose a "delightful" puce (R157 G99 B120).
4. Select the **Clock Game Object** from the **Hierarchy** panel. It's not a bad idea to look in the **Inspector** panel and confirm that the **clockScript** script was added as a component in the preceding instruction.
5. With the **Clock Game Object** selected, go to **Component | Rendering | GUIText**. This is the **GUIText** component that we'll use to display the clock numbers on the screen.

6. In the **Inspector** panel, find the **GUIText** component and type **whatever** in the blank **Text** property.


Inspector

Clock

Tag Untagged Layer Default

Transform

Position X 0.8 Y 0.9 Z -15.68567

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

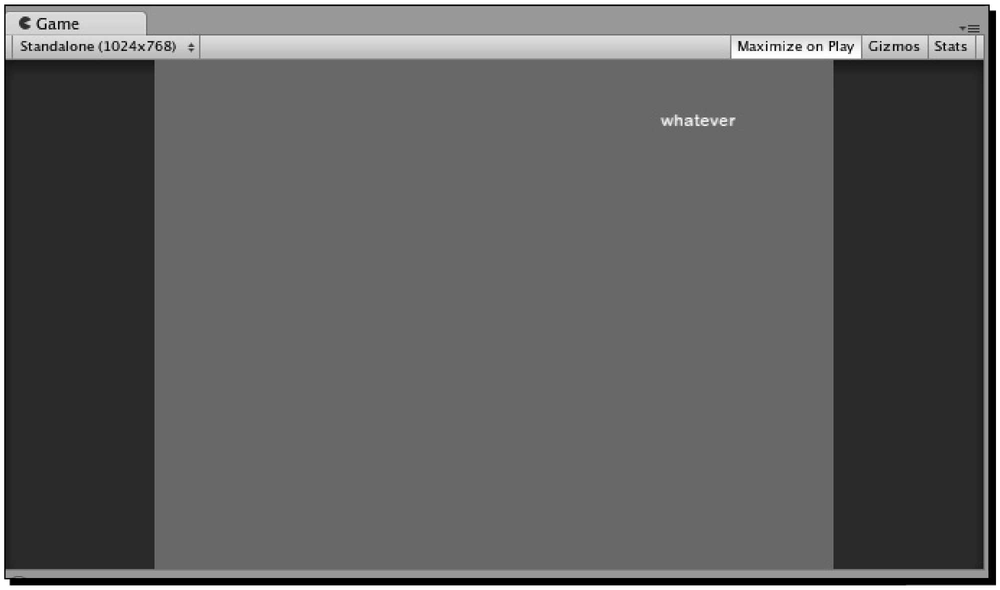
Clock Script (Script)

Script clockScript

GUIText

Text	whatever
Anchor	upper left
Alignment	left
Pixel Offset	
Line Spacing	1
Tab Size	4
Font	None (Font)
Material	None (Material)
Pixel Correct	✓

In the **Inspector** panel, change the clock's **X** position to **0.8** and its **Y** position to **0.9** to bring it into view. You should see the word **whatever** in white, floating near the top-right corner of the screen in the **Game** view..



Right, then! We have a **Game Object** with an empty script attached. That **Game Object** has a **GUIText** component to display the clock numbers. Our game background is certifiably hideous. Let's code us some clock.

Still time for action – change the clock text color

Double-click the **clockScript**. Your empty script, with one lone `Update()` function, should appear in the code editor. The very first thing we should consider is doing away with our puce background by changing the **GUIText** color to black instead of white. Let's get at it.

1. Write the built-in `Start` function and change the **GUIText** color:

```
function Start()
{
   guiText.material.color = Color.black;
}
function Update() {
}
```

2. Save the script and test your game to see your new black text.

If you feel comfy, you can change the game background color back to white by clicking on the **Main Camera Game Object** and finding the color swatch in the **Inspector** panel. The white **whatever GUIText** will disappear against the white background in the **Game** view because the color-changing code that we just wrote runs only when we test the game (try testing the game to confirm this). If you ever lose track of your text, or it's not displaying properly, or you just really wanna *see* it on the screen, you can change the camera's background color to confirm that it's still there.

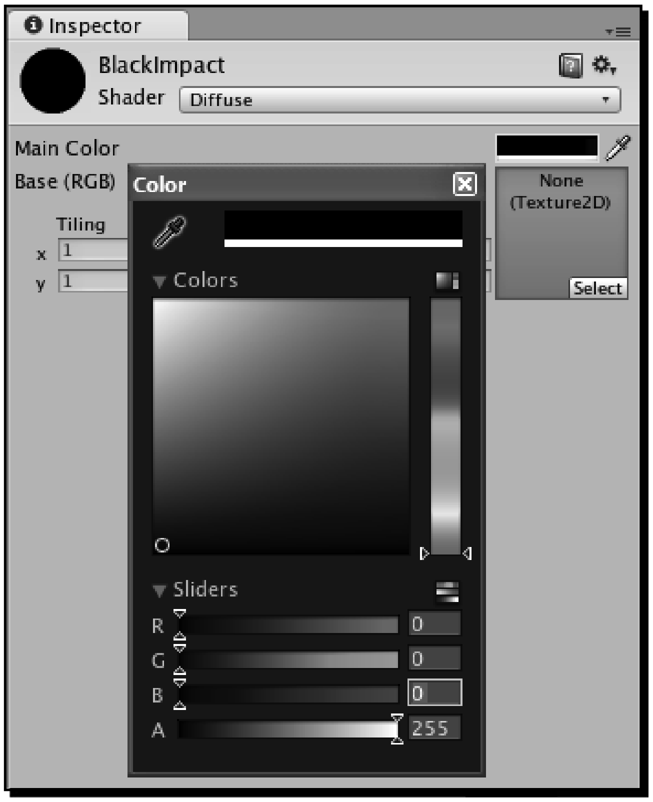
If you're happy with this low-maintenance, disappearing-text arrangement, you can move on to the *Prepare the clock code* section. But, if you want to put in a little extra elbow grease to actually *see* the text, in a font of your choosing, follow these next steps.

Time for action rides again – create a font texture and material

Ha! I knew you couldn't resist. In order to change the font of this **GUIText**, and to see it in a different color without waiting for the code to run, we need to import a font, hook it up to a **Material**, and apply that **Material** to the **GUIText**.

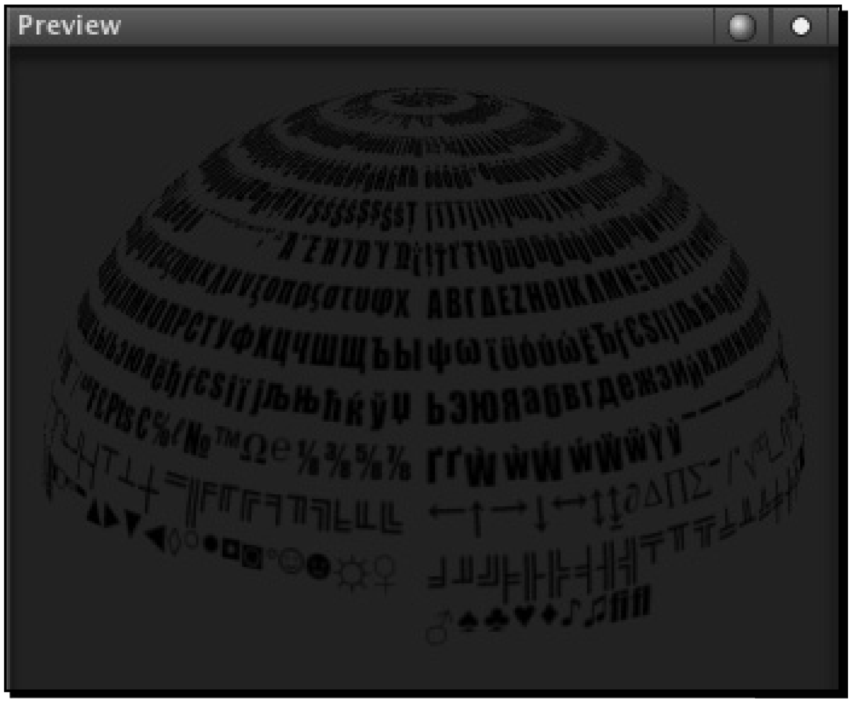
1. Find a font that you want to use for your game clock. I like the LOLCats standby Impact. If you're running Windows, your fonts are likely to be in the `C:\Windows\Fonts` directory. If you're a Mac user, you should look in the `\Library\Fonts\` folder.

2. Drag the font into the **Project** panel in Unity. The font will be added to your list of **Assets**.
3. Right-click (or secondary-click) an empty area of the **Project** panel and choose **Create | Material**. You can also click on the **Create** button at the top of the panel.
4. Rename the new **Material** to something useful. Because I'm using the Impact font, and it's going to be black, I named mine "BlackImpact" (incidentally, "Black Impact" is also the name of my favorite exploitation film from the 70s).
5. Click on the **Material** you just created in the **Project** Panel.
6. In the **Inspector** panel, click on the color swatch labeled **Main Color** and choose black (R0 G0 B0), then click on the little red **X** to close the color picker.



7. In the empty square area labeled **None (Texture 2D)**, click on the **Select** button, and choose your font from the list of textures (mine was labeled **impact - font texture**).

8. At the top of the **Inspector** panel, there's a drop-down labeled **Shader**. Select **Transparent/Diffuse** from the list. You'll know it worked when the preview sphere underneath the **Inspector** panel shows your chosen font outline wrapped around a transparent sphere. Pretty cool!



9. Click on the **Clock Game Object** in the **Hierarchy** panel.
10. Find the **GUIText** component in the **Inspector** panel.
11. Click and drag your font—the one with the letter **A** icon—from the **Project** panel into the parameter labeled **Font** in the **GUIText** component. You can also click the drop-down arrow (the parameter should say **None (Font)** initially) and choose your font from the list.
12. Similarly, click-and-drag your **Material**—the one with the gray sphere icon—from the **Project** panel into the parameter labeled **Material** in the **GUIText** component. You can also click on the drop-down arrow (the parameter should say **None (Material)** initially) and choose your **Material** from the list.

Just as you always dreamed about since childhood, the **GUIText** changes to a solid black version of the fancy font you chose! Now, you can definitely get rid of that horrid puce background and switch back to white. If you made it this far and you're using a **Material** instead of the naked font option, it's also safe to delete the `guiText.material.color = Color.black;` line from the **clockScript**.

Time for action – what's with the tiny font?

The Impact font, or any other font you choose, won't be very... impactful at its default size. Let's change the import settings to biggify it.

1. Click on your imported font—the one with the letter **A** icon—in the **Project** panel.
2. In the **Inspector** panel, you'll see the True Type Font Importer. Change the **Font Size** to something respectable, like 32, and press the *Enter* key on your keyboard.
3. Click on the **Apply** button. Magically, your **GUIText** cranks up to 32 points (you'll only see this happen if you still have a piece of text like "whatever" entered into the **Text** parameter of the **GUIText** of the **Clock Game Object** component).

What just happened - was that seriously magic?

Of course, there's nothing magical about it. Here's what happened when you clicked on that **Apply** button:

When you import a font into Unity, an entire set of raster images is created for you by the True Type Font Importer. **Raster images** are the ones that look all pixelly and square when you zoom in on them. Fonts are inherently vector instead of raster, which means that they use math to describe their curves and angles. Vector images can be scaled up any size without going all Rubik's Cube on you.

But, Unity doesn't support vector fonts. For every font size that you want to support, you need to import a new version of the font and change its import settings to a different size. This means that you may have four copies of, say, the Impact font, at the four different sizes you require.

When you click on the **Apply** button, Unity creates its set of raster images based on the font that you're importing.

Time for action – prepare the clock code

Let's rough in a few empty functions and three variables that we'll need to make the clock work.

1. Open up the **clockScript** by double-clicking it. Update the code:

```
var isPaused : boolean = false;
var startTime : float; //(in seconds)
var timeRemaining : float; //(in seconds)
function Start()
{
}

function Update() {
  if (!isPaused)
  {
     // make sure the timer is not paused
     DoCountdown();
  }
}

function DoCountdown() {
}

function PauseClock()
{
   isPaused = true;
}

function UnpauseClock()
{
   isPaused = false;
}

function ShowTime()
{
}

function TimeIsUp()
{
}
```

What just happened – that's a whole lotta nothing

Here, we've created a list of functions that we'll probably need to get our clock working. The functions are empty, but that's okay—roughing them in like this is a really valid and useful way to program. We have a `doCountdown()` function that we call on every update, as long as our `isPaused` flag is `false`. We have `pauseClock()` and `unpauseClock()` functions—each of them needs only one simple line of code to change the `isPaused` flag, so we've included that. In the `showTime()` function, we'll display the time in the **GUIText** component. Finally, we'll call the `timeIsUp()` function when the clock reaches zero.

At the top of the script are three hopefully self-explanatory variables: a Boolean to hold the clock's paused state, a floating point number to hold the start time, and another to hold the remaining time.

Now that we have the skeleton of our clock code and we see the scope of work ahead of us, we can dive in and flesh it out.

Time for action – create the countdown logic

Let's set the `startTime` variable, and build the logic to handle the counting-down functionality of our clock.

1. Set the `startTime` variable:

```
function Start() {
   startTime = 5.0;
}
```

[Note: five seconds to beat the game is a bit ridiculous. We're just keeping the time tight for testing. You can crank this variable up later.]

2. Decrease the amount of time on the clock:

```
function DoCountdown()
{
   timeRemaining = startTime - Time.time;
}
```

3. If the clock hits zero, pause the clock and call the `TimeIsUp()` function:

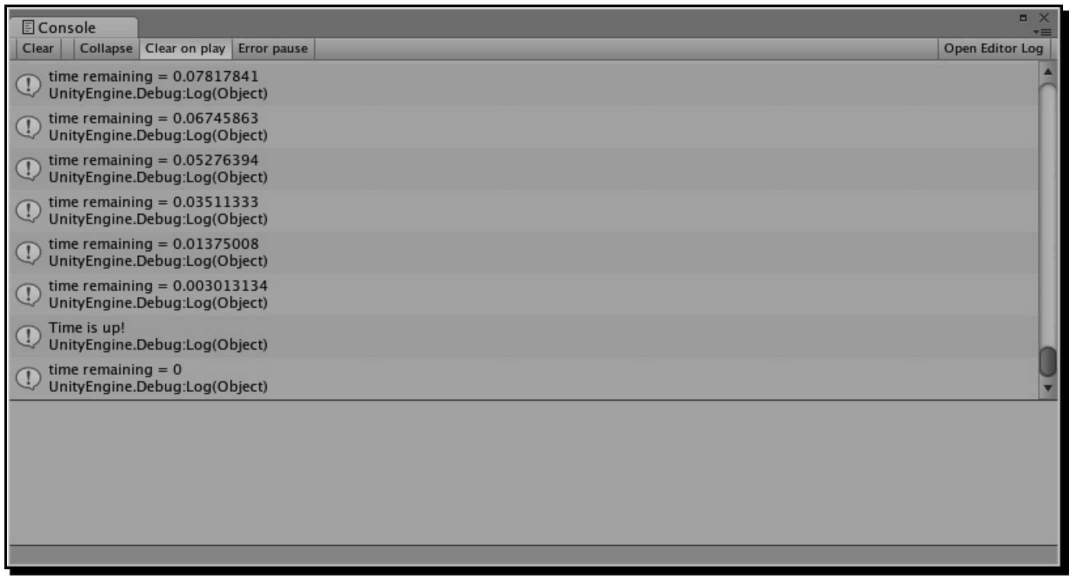
```
timeRemaining = startTime - Time.time;
if (timeRemaining < 0)
{
    timeRemaining = 0;
    isPaused = true;
    TimeIsUp();
}
```

4. Add some `Debug` statements so that you can see something happening:

```
function DoCountdown()
{
   // (other lines omitted for clarity)
   Debug.Log("time remaining = " + timeRemaining);
}
function TimeIsUp()
{
   Debug.Log("Time is up!");
}
```

5. Save the script and test your game.

You should see the `Debug` statements in the Console window (**Window | Console**) or in the information bar at the bottom of the screen. When the clock finishes ticking down through five seconds, you'll see the **Time is up!** message… *if you're a mutant with super-speed vision*. The **Time is up!** message gets wiped out by the next "time remaining" message. If you want to see it with normal-people vision, open the Console window (**Window | Console** in the menu and watch for it in the list of printed statements.



```
time remaining = 0.07817841
UnityEngine.Debug:Log(Object)
time remaining = 0.06745863
UnityEngine.Debug:Log(Object)
time remaining = 0.05276394
UnityEngine.Debug:Log(Object)
time remaining = 0.03511333
UnityEngine.Debug:Log(Object)
time remaining = 0.01375008
UnityEngine.Debug:Log(Object)
time remaining = 0.003013134
UnityEngine.Debug:Log(Object)
Time is up!
UnityEngine.Debug:Log(Object)
time remaining = 0
UnityEngine.Debug:Log(Object)
```

Time for action – display the time on-screen

We know from the `Debug` statements that the clock is working, so all we need to do is stick the `timeRemaining` value into our **GUIText**. But, it won't look very clock-like unless we perform a tiny bit of math on that value to split it into minutes and seconds so that five seconds displays as 0:05, or 119 seconds displays as 1:59.

1. Call the `ShowTime()` function from within the `DoCountdown()` function (you can delete the `Debug.Log` statement):

```
function DoCountdown()
{
  timeRemaining = startTime - Time.time;
  // (other lines omitted for clarity)
  ShowTime();
  Debug.Log("time remaining = " + timeRemaining);
}
```

2. Create some variables to store the minutes and seconds values in the `ShowTime()` function, along with a variable to store the string (text) version of the time:

```
function ShowTime() {
   var minutes : int;
   var seconds : int;
   var timeStr : String;
}
```

3. Divide the `timeRemaining` by `60` to get the number of minutes that have elapsed:

```
   var timeStr : String;
   minutes = timeRemaining/60;
```

4. Store the remainder as the number of elapsed seconds:

```
   minutes = timeRemaining/60;
   seconds = timeRemaining % 60;
```

5. Set the text version of the time to the number of elapsed minutes, followed by a colon:

```
   seconds = timeRemaining % 60;
   timeStr = minutes.ToString() + ":";
```

6. Append (add) the text version of the elapsed seconds:

```
   timeStr = minutes.ToString() + ":";
   timeStr += seconds.ToString("D2");
```

7. Finally, push the `timeStr` value to the **GUIText** component:

```
timeStr += seconds.ToString("D2");
guiText.text = timeStr; //display the time to the GUI
```

8. To gaze at your clock longingly as it counts down every delectable second, crank up the `startTime` amount in the `Start` function:

```
startTime = 120.0;
```

9. Save and test.

Your beautiful new game clock whiles away the hours in your own chosen font at your own chosen size, formatted with a colon like any self-respecting game clock should be.

What just happened – what about that terrifying code?

There are a few kooky things going on in the code we just wrote. Let's hone in on the new and weird stuff.

```
minutes = timeRemaining/60;
```

We set the value of the minutes variable by dividing `timeRemaining` by `60`. If there were 120 seconds left, dividing by 60 means there are two minutes on the clock. If there were only 11 seconds left on the clock, dividing by 60 gives us 0.183 repeating.

The reason why we don't see 0.1833333 in the "minutes" portion of our clock is a sneaky data type trick: `timeRemaining` is a `float` (floating point number), so it can remember decimal places. Our `minutes` variable is an `int` (integer), which *can't* remember decimal places. So, when we feed 0.183333 into an `int` variable, it just snips off everything after the decimal. It can't remember that stuff, so why bother? The result is that `minutes` is set to 0, which is what we see on the clock.

```
seconds = timeRemaining % 60;
```

This unnerving little percent-symbol beastie performs a modulo operation, which is like those exercises you did in grade three math class when you were just about to learn long division. A *modulo* operation is like a division, except it results in the *remainder* of your operation. Here are a few examples:

- 4 % 2 = 0 because 2 goes into 4 twice with no remainder
- 11 % 10 = 1 because 10 goes into 11 once with 1 as the remainder (10+1 = 11)

- 28 % 5 = 3 because 5 goes into 28 five times (5*5=25), with a remainder of 3 (25+3 = 28)

```
timeStr = minutes.ToString() + ":";
```

The `int.ToString()` method works exactly as advertised, converting an integer to a String. Many data types have `ToString()` methods. At the end of this line, we're tacking on a colon (`:`) to separate the minutes from the seconds.

```
timeStr += seconds.ToString("D2");
```

The `+=` operator, you'll remember, takes what we've already got and adds something new to it—in this case, it's `seconds.ToString();`. We're passing the special argument `D2` to the `ToString()` method to round to the nearest two decimal places so that the clock looks like this:

4:02

instead of this:

4:2

Nifty!

Whatever is the problem?

If you still have **whatever** as your placeholder text, now's a good time to consider ditching it. It shows up on-screen for a split-second before the numbers appear. Yuck! Click on the **Clock GameObject** in the **Hierarchy** panel, and clear out the value marked **Text** in the **GUIText** component in the **Inspector** panel.

Picture it

Number clocks look alright, but graphical clocks really get me revved up and ready for some pressurized pants-wetting. Nothing denotes white-knuckled urgency like a bar that's slowly draining. We don't need to do much extra work to convert our number clock into a picture clock, so let's go for it!

Time for action – grab the picture clock graphics

As per our agreement, all the pretty pictures are pre-drawn for you. Download the Unity assets package for this chapter. When you're ready to import it, click on **Assets | Import Package...** and find the `.unitypackage` file. Open it up, and thar she be! If only obtaining game graphics in a real production environment were this easy.

Let's get right to work by creating some code to make use of two of the graphics in the package—a blue clock background bar, and a shiny yellow foreground bar that will slowly shrink as time marches on.

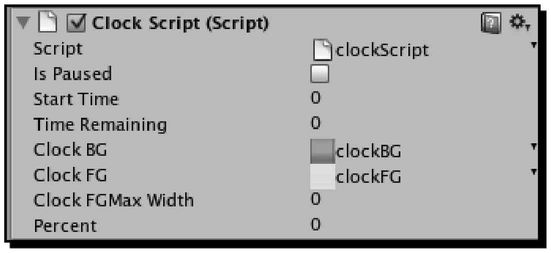
1. Create a variable to store the elapsed time as a percentage:

```
var isPaused : boolean = false;
var startTime : float; //(in seconds)
var timeRemaining : float; //(in seconds)
var percent:float;
```

2. Create some variables to store the textures, as well as the initial width of the yellow foreground bar:

```
var percent:float;
var clockBG:Texture2D;
var clockFG:Texture2D;
var clockFGMaxWidth:float; // the starting width of the foreground
bar
```

3. Save the script and go back to Unity.
4. Click on the **Clock Game Object** in the **Hierarchy** panel.
5. Find the **clockScript** component in the **Inspector** Panel.
6. The new `clockBG` and `clockFG` variables we just created are now listed there. Click-and-drag the `clockBG` texture from the **Project** panel to the `clockBG` slot, and then click-and-drag the `clockFG` texture into its slot.



Clock Script (Script)

Script	clockScript
Is Paused	☐
Start Time	0
Time Remaining	0
Clock BG	clockBG
Clock FG	clockFG
Clock FGMax Width	0
Percent	0

What just happened – you can do that?

This is yet another example of Unity's drag-and-drop usefulness. We created the two `Texture2D` variables, and the variables appeared on the **Script** component in the **Inspector** panel. We dragged-and-dropped two textures into those slots, and now whenever we refer to `clockBG` and `clockFG` in code, we'll be talking about those two texture images. Handy, yes?

Time for action – flex those GUI muscles

Let's take a trip down memory lane to the previous chapter, where we became `OnGUI` ninjas. We'll use the GUI techniques we already know to display the two bars, and shrink the foreground bar as time runs out.

1. In the `DoCountdown` function, calculate the percentage of time elapsed by comparing the `startTime` and the `timeRemaining` values:

```
function DoCountdown()
{
  timeRemaining = startTime - Time.time;
  percent = timeRemaining/startTime * 100;
  if (timeRemaining < 0)
  {
    timeRemaining = 0;
    isPaused = true;
    TimeIsUp();
  }
  ShowTime();
}
```

2. Store the initial width of the `clockFG` graphic in a variable called `clockFGMaxWidth` in the `Start` function:

```
function Start()
{
  startTime = 120.0;
  clockFGMaxWidth = clockFG.width;
}
```

3. Create the built-in `OnGUI` function somewhere apart from the other functions in your script (make sure you don't create it inside the curly brackets of some other function!)

```
function OnGUI()
{
  var newBarWidth:float = (percent/100) * clockFGMaxWidth; // this
is the width that the foreground bar should be
  var gap:int = 20; // a spacing variable to help us position the
clock
}
```

4. Create a new group to contain the `clockBG` texture. We'll position the group so that the `clockBG` graphic appears 20 pixels down, and 20 away from the right edge of the screen:

```
function OnGUI()
{
  var newBarWidth:float = (percent/100) * clockFGMaxWidth; // this
is the width that the foreground bar should be
  var gap:int = 20; // a spacing variable to help us position the
clock
  GUI.BeginGroup (new Rect Screen.width - clockBG.width - gap,
    gap, clockBG.width, clockBG.height));
  GUI.EndGroup ();
}
```

5. Use the `DrawTexture` method to draw the `clockBG` texture inside the group:

```
  GUI.BeginGroup (new Rect (Screen.width - clockBG.width - gap,
   gap, clockBG.width, clockBG.height));
  GUI.DrawTexture (Rect (0,0, clockBG.width, clockBG.height),
    clockBG);
  GUI.EndGroup ();
```

6. Nest another group inside the first group to hold the `clockFG` texture. Notice that we're offsetting it by a few pixels (5,6) so that it appears inside the `clockBG` graphic:

```
  GUI.BeginGroup (new Rect (Screen.width - clockBG.width - gap,
    gap, clockBG.width, clockBG.height));
  GUI.DrawTexture (Rect (0,0, clockBG.width, clockBG.height),
    clockBG);
  GUI.BeginGroup (new Rect (5, 6, newBarWidth, clockFG.height));
  GUI.EndGroup ();
  GUI.EndGroup ();
```

7. Now, draw the `clockFG` texture inside that nested group:

```
GUI.BeginGroup (new Rect (5, 6, newBarWidth, clockFG.height));
GUI.DrawTexture (Rect (0,0, clockFG.width, clockFG.height),
  clockFG);
GUI.EndGroup ();
```

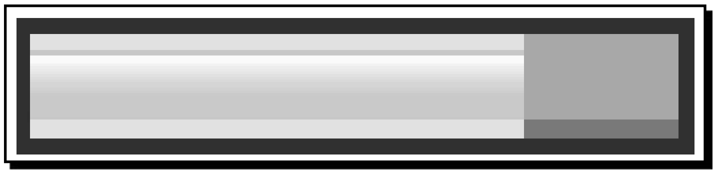
Did you punch that in correctly? Here's what the entire function looks like all at once:

```
function OnGUI()
{
  var newBarWidth:float = (percent/100) * clockFGMaxWidth; // this
   is the width that the foreground bar should be
```

```
        var gap:int = 20; // a spacing variable to help us position the
         clock
        GUI.BeginGroup (new Rect (Screen.width - clockBG.width - gap,
          gap, clockBG.width, clockBG.height));
        GUI.DrawTexture (Rect (0,0, clockBG.width, clockBG.height),
         clockBG);
           GUI.BeginGroup (new Rect (5, 6, newBarWidth,
            clockFG.height));
           GUI.DrawTexture (Rect (0,0, clockFG.width, clockFG.height),
            clockFG);
           GUI.EndGroup ();
        GUI.EndGroup ();
      }
```

Save the script and jump back into Unity. Let's turn off the old and busted number clock so that we can marvel at the new hotness—our graphical clock. Select the **Clock GameObject** in the **Hierarchy** panel. Locate the **GUIText** component in the **Inspector** panel, and uncheck its check box to turn off its rendering. Now, test your game.

Fantastic! Your new graphical clock drains ominously as time runs out. That'll really put the heat on, without compromising your player's doubtless lust for eye candy.



What just happened – how does it work?

The math behind this magic is simple ratio stuff that I, for one, conveniently forgot the moment I escaped the sixth grade and ran away to become a kid who enjoys watching circus clowns. We converted the time elapsed into a percentage of the total time on the clock. Then, we used that percentage to figure out how wide the `clockFG` graphic should be. Percentage of time elapsed is to 100, as the width of the `clockFG` graphic is to its original width. It's simple algebra from there.

The only math you need to know

I often say that this dead-simple ratio stuff is the only math you need to know as a game developer. I'm half-kidding, of course. Developers with a firm grasp of trigonometry can create billiard and pinball games, while Developers who know differential calculus can create a tank game where the enemy AI knows which way to aim its turret, taking trajectory, wind, and gravity into account (hey, kids! Stay in school).

But, when people ask sheepishly about programming and how difficult it is to learn, it's usually because they think you can only create games by using complex mathematical equations. We've hopefully debunked this myth by creating two simple games so far with equally simple math. These ratio calculations are used ALL THE TIME in gaming, from figuring out health bars, to player positions on a mini-map, to level progress meters. If you're arithmophobic like I am, and you relearn only *one* piece of math from your long-forgotten elementary school days, make it this one.

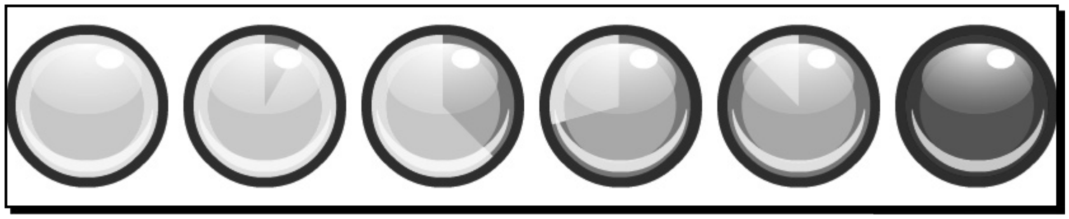
The incredible shrinking clock

We've seen in earlier chapters that the `GUI.BeginGroup()` and `GUI.EndGroup()` functions can wrap our fixed-position UI controls together so that we can move them around as a unit. In the preceding code, we made one group to hold the background bar, and another inside it to hold the foreground bar, offset slightly. The outer group is positioned near the right edge of the screen, using the gap value of 20 pixels to set it back from the screen edges.

When we draw the foreground clock bar, we draw it at its normal size, but the group that wraps it is drawn at the shrinking size, so it cuts the texture off. If you put a 500x500 pixel texture inside a 20x20 pixel group, you'll only see a 20x20 pixel portion of the larger image. We use this to our advantage to display an ever-decreasing section of our clock bar.

Keep your fork—there's pie!

The third type of clock we'll build is a pie chart-style clock. Here's what it will look like when it's counting down:



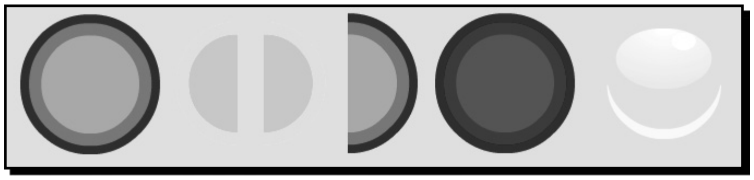
Pop quiz – how do we build it?

Before reading any further, stare at the picture of the pie clock and try to figure out how you would build it if you didn't have this book in front of you. As a new game developer, you'll spend a significant amount of time trying to mimic or emulate different effects you see in the games you enjoy. It's almost like watching a magic show and trying to figure out how they did it. So, how'd they do it?

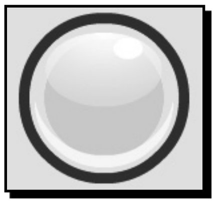
(If you need a hint, take a look at the unused textures in the **Project** panel that you imported earlier in the chapter).

How they did it

The pie clock involves a little sleight of hand. Here are the pieces that make up the clock:

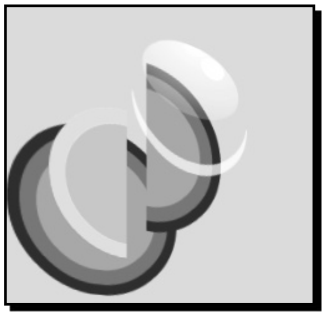


It's a bit like a sandwich. We start by drawing the blue background. Then, we layer on the two yellow half-moon pieces. To make the clock look pretty, we apply the lovely, shiny gloss picture in front of all these pieces.

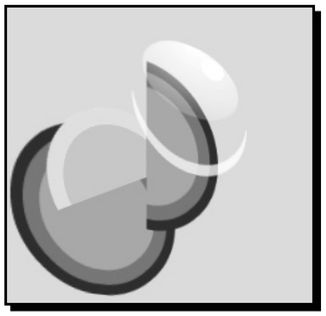


We rotate the right half-moon piece halfway around the circle to make the slice of yellow time appear to grow smaller and smaller.

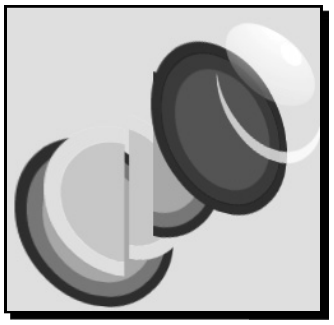
At the halfway point, we slap a half-moon section of the blue background on top of everything, on the right side of the clock. We're all done with the right half-circle, so we don't draw it.



Now, we rotate the left half-moon piece, and it disappears behind the blocker graphic, creating the illusion that the rest of the clock is depleting.



When time is up, we slap the red clock graphic in front of everything.



A little flourish of the wrist, a puff of smoke, and the audience doesn't suspect a thing!

Time for action – rig up the textures

We know most of what we need to get started. There's just a little trick with rotating the textures that we have to learn. But, first, let's set up our clock with the variables we'll need to draw the textures to the screen.

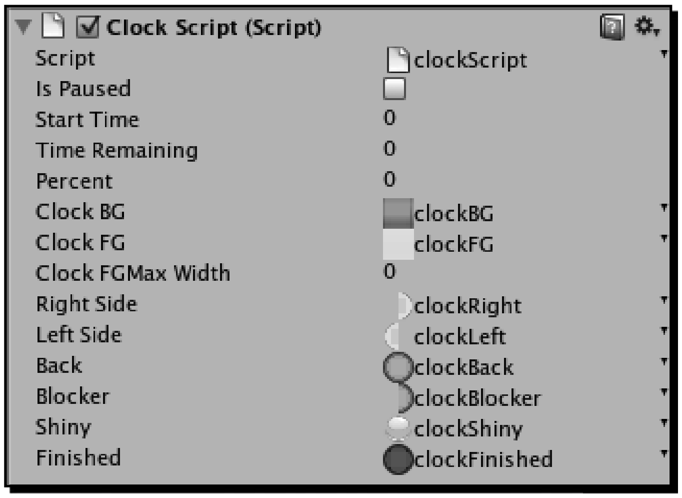
1. Add variables for the new pie clock textures at the top of the **clockScript**:

```
var clockFGMaxWidth:float; // the starting width of the foreground
bar

var rightSide:Texture2D;
var leftSide:Texture2D;
var back:Texture2D;
var blocker:Texture2D;
var shiny:Texture2D;
var finished:Texture2D;
```

2. In the **Hierarchy** panel, select the **Clock** Game Object.

3. Just as we did with the bar clock, drag-and-drop the pie clock textures from the **Project** panel into their respective slots in the **clockScript** component in the **Inspector** panel. When you're finished, it should look like this:



Clock Script (Script)

Script	clockScript
Is Paused	☐
Start Time	0
Time Remaining	0
Percent	0
Clock BG	clockBG
Clock FG	clockFG
Clock FGMax Width	0
Right Side	clockRight
Left Side	clockLeft
Back	clockBack
Blocker	clockBlocker
Shiny	clockShiny
Finished	clockFinished

Time for action – write the pie chart Script

With these Texture2D variables defined, and the images stored in those variables, we can control the images with our script. Let's lay down some code, and pick through the aftermath when we're finished:

1. Knowing when the clock has passed the halfway point is pretty important with this clock. Let's create an isPastHalfway variable inside our OnGUI function:

   ```
   function OnGUI ()
   {
      var isPastHalfway:boolean = percent < 50;
   ```

   (Confused? Remember that our percent variable means "percent remaining", not "percent elapsed". When percent is less than 50, we've passed the halfway mark).

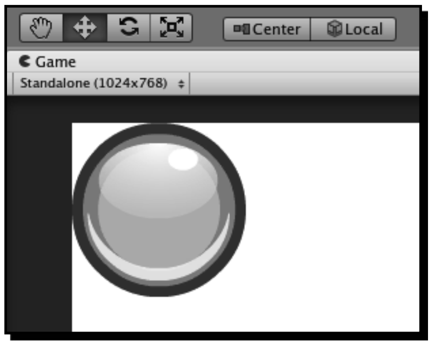
2. Define a rectangle in which to draw the textures:

   ```
   var isPastHalfway:boolean = percent < 50;
   var clockRect:Rect = Rect(0, 0, 128, 128);
   ```

3. Draw the background blue texture and the foreground shiny texture:

   ```
   var clockRect:Rect = Rect(0, 0, 128, 128);
   GUI.DrawTexture(clockRect, back, ScaleMode.StretchToFill, true,
      0);
   GUI.DrawTexture(clockRect, shiny, ScaleMode.StretchToFill, true,
      0);
   ```

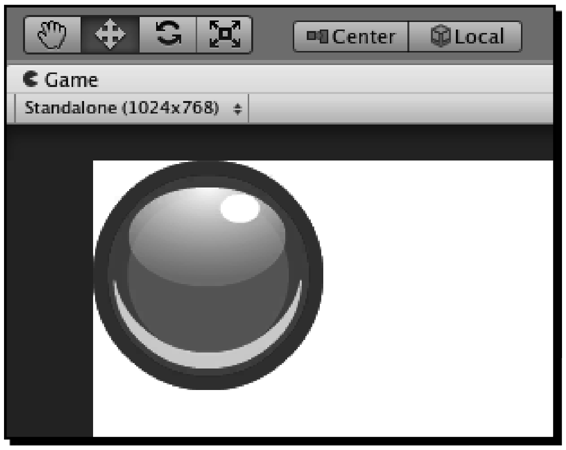
4. Save the script and play the game. You should see your shiny blue clock glimmering at you in the top-left corner of the screen:



5. Next, we'll add a condition check and draw the red "finished" graphic over the top of everything when percent goes below zero (which means that time is up). Add that bit just above the "shiny" texture draw so that the shiny picture still layers on top of the red "finished" graphic:

```
if(percent < 0)
{
   GUI.DrawTexture(clockRect, finished, ScaleMode.StretchToFill,
    true, 0);
}
GUI.DrawTexture(clockRect, shiny, ScaleMode.StretchToFill, true,
  0);
```

6. Save the script and play the game again to confirm that it's working. When time runs out, your blue clock should turn red:



7. Let's set up a `rotation` variable. We'll use the percent value to figure out how far along the 360 degree spectrum we should be rotating those yellow half-circle pieces. Note that once again, we're using the same ratio math that we used earlier with our bar clock:

```
var clockRect:Rect = Rect(0, 0,128,128);
var rot:float = (percent/100) * 360;
```

If you want to see it working, try adding a `Debug.Log` or `print` statement underneath to track the value of `rot`. The value should hit 360 when the clock times out.

8. We have to set two more variables before the fun begins—a `centerPoint` and a `startMatrix`. I'll explain them both in a moment:

```
   var rot:float = (percent/100) * 360;
   var centerPoint:Vector2 = Vector2(64, 64);
   var startMatrix:Matrix4x4  = GUI.matrix;
```

What just happened?

One important thing to know is that, unlike 3D Game Objects like the **Paddle** and **Ball** from our keep-up game, GUI textures can't be rotated. Even if you apply them to a Game Object and rotate the Game Object, the textures won't rotate (or will skew in a very strange way). We know we need to rotate those two half-circle pieces to make the pie clock count down, but because of this limitation, we'll have to find a creative workaround.

Here's the game plan: we're going to use a method of the `GUIUtility` class called `RotateAroundPivot`. The `centerPoint` value we created defines the point around which we'll rotate. `RotateAroundPivot` rotates the entire GUI. It's as if the GUI controls were stickers on a sheet of glass, and instead of rotating the stickers, we're rotating the sheet of glass.

So, we're going to follow these steps to rotate those half-circles:

1. Draw the blue clock background.
2. Rotate the GUI using the `rot` (rotation) value we set.
3. Draw the yellow half-circle pieces in their rotated positions. This is like stamping pictures on a piece of paper with a stamp pad. The background is already stamped. Then, we rotate the paper and stamp the half-circles on top.
4. Rotate the GUI back to its original position.
5. Draw or stamp the "finished" graphic (if the timer is finished) and the shiny image.

So, the background, the red "finished" image, and the shiny image all get drawn when the GUI is in its normal orientation, while the half-circle pieces get drawn when the GUI is rotated. Kinda neat, huh? That's what the `startMatrix` variable is all about. We're storing the matrix transformation of the GUI so that we can rotate it back to its "start" position later.

Time for action – commence operation pie clock

Let's lay down some code to make those half-circle pieces rotate.

1. Set up a conditional statement so that we can draw different things before and after the halfway point:

```
GUI.DrawTexture(clockRect, back, ScaleMode.StretchToFill, true,
  0);
  if(isPastHalfway)
  {
  } else {
  }
```

2. If we're not past halfway, rotate the GUI around the `centerPoint`. Draw the right side half-circle piece on top of the rotated GUI. Then, rotate the GUI back to its start position.

```
if(isPastHalfway)
{
}
else
{
  GUIUtility.RotateAroundPivot(-rot, centerPoint);
  GUI.DrawTexture(clockRect, rightSide, ScaleMode.StretchToFill,
    true, 0);
  GUI.matrix = startMatrix;
}
```

3. Save the script and test your game. You should see the right side half-circle piece rotating around the clock. But, we know it's not really rotating—the entire GUI is rotating, and we're stamping the image on the screen before resetting the rotation.

4. Draw the left half-circle once the GUI is back into position.

```
GUI.matrix = startMatrix;
GUI.DrawTexture(clockRect, leftSide, ScaleMode.StretchToFill,
   true, 0);
```

5. You can save and test at this point, too. The right half-circle disappears behind the left half-circle as it rotates, creating the illusion we're after. It all comes crashing down after the halfway point, though. Let's fix that:

```
 if(isPastHalfway)
 {
    GUIUtility.RotateAroundPivot(-rot-180, centerPoint);
    GUI.DrawTexture(clockRect, leftSide, ScaleMode.StretchToFill,
     true, 0);
    GUI.matrix = startMatrix;
```

6. Save and test. Once we're past the halfway point, the left half-circle does its thing, but the illusion is ruined because we see it rotating into the right side of the clock. We just need to draw that blocker graphic to complete the illusion:

```
  GUI.matrix = startMatrix;
  GUI.DrawTexture(clockRect, blocker, ScaleMode.StretchToFill,
    true, 0);
```

7. Save and test one last time. You should see your pie clock elapse exactly as described in the sales brochure.

What just happened – explaining away the loose ends

Hopefully, the code is straightforward. Notice that in this line:

```
GUIUtility.RotateAroundPivot(-rot, centerPoint);
```

We're sticking a minus sign on the `rot` value—that's like multiplying a number by negative-one. If we don't do this, the half-circle will rotate in the opposite direction (try it yourself! Nix the minus sign and try out your game).

Similarly, in this line:

```
GUIUtility.RotateAroundPivot(-rot-180, centerPoint);
```

We're using the negative `rot` value, and we're subtracting 180 degrees. That's because the left half-circle is on the other side of the clock. Again, try getting rid of the `-180` and see what effect that has on your clock.

Another thing that you may want to try is changing the `centerPoint` value. Our pie clock graphics are 128x128 pixels, so the center point is at 64, 64. Mess around with that value and check out the funky stuff the clock starts doing.

```
GUI.matrix = startMatrix;
```

It's worth mentioning that this line locks the GUI back into position, based on the `startMatrix` value we stored.

```
GUI.DrawTexture(clockRect, leftSide, ScaleMode.StretchToFill, true,
   0);
```

Did I catch you wondering what `ScaleMode.StretchToFill` was all about? There are three different settings you can apply here, all of which fill the supplied rectangle with the texture in a different way. Try looking them up in the Script Reference to read about what each one does.

Time for action – positioning and scaling the clock

The pie clock is pretty neat, but it's sadly stuck to the top-left corner of the screen. It would be great if we could make it any size we wanted, and if we could move it anywhere on the screen.

We're not far off from that goal. Follow these steps to get a dynamically positioned and scaled pie clock:

1. Create these variables at the top of the `OnGUI` function:

   ```
   var pieClockX:int = 100;
   ```

```
var pieClockY:int = 50;

var pieClockW:int = 64; // clock width
var pieClockH:int = 64; // clock height

var pieClockHalfW:int = pieClockW * 0.5; // half the clock width
var pieClockHalfH:int = pieClockH * 0.5; // half the clock
 height
```

In this example, `100` and `50` are the X and Y values where I'd like the pie clock to appear on the screen. The clock builds out from its top-left corner. 64 and 64 are the width and height values I'd like to make the clock—that's exactly half the size of the original clock.

Note: scaling the clock will result in some ugly image artifacting, so I don't really recommend it. In fact, plugging in non-uniform scale values like 57x64 will destroy the illusion completely! But, learning to make the clock's size dynamic is still a worthwhile coding exercise, so let's keep going.

2. Modify the `clockRect` declaration to make use of the new x, y, width, and height variables:

   ```
   var clockRect:Rect = Rect(pieClockX, pieClockY, pieClockW,
      pieClockH);
   ```

3. Modify the `centerPoint` variable to make sure we're still hitting the dead-center of the clock:

   ```
   var centerPoint:Vector2 = Vector2(pieClockX + pieClockHalfW,
     pieClockY + pieClockHalfH);
   ```

4. Save the script and test your game. You should see a pint-sized clock (with a few ugly pixels here and there) at x100 y50 on your screen.

Have a go hero – rock out with your clock out

There's lots more you could add to your clock to juice it up. Here are a few ideas:

- Add some logic to the `TimeIsUp()` method. You could pop up a new GUI window that says **Time is up!** with a **Try Again** button, or you could link to another **Scene** showing your player's character perishing in a blazing inferno… whatever you like!
- Create a pause/unpause button that starts and stops the timer. The `clockScript` is already set up to do this—just toggle the `isPaused` variable.

- In a few chapters, we'll talk about how to add sound to your games. Bring that knowledge back with you to this chapter to add some ticking and buzzer sound effects to your clock.
- Create a button that says **More Time!**. When you click on it, it should add more time to the clock. When you get this working, you can use this logic to add power-ups to your game that increase clock time.
- Use the skills that you've already acquired to tie this clock into any game that you create in Unity, including the keep-up and robot games you've built with this book.

Unfinished business

With this chapter, you've mastered an important step in your journey as a game developer. Understanding how to build a game clock will serve you well in nearly all of the games you venture off to build. Games without some kind of clock or timer are uncommon, so adding this notch to your game developer tool belt is a real victory. Here are some skills you learned in this chapter:

- Creating a font material
- Displaying values on-screen with **GUIText**
- Converting numbers to strings
- Formatting string data to two decimal places
- Ratios: the only math you'll ever need (according to someone who doesn't know math)!
- Storing texture images in variables
- Scaling or snipping graphics based on script data
- Rotating, and then unrotating, the GUI
- Converting hardcoded script values to dynamic script values

With three chapters behind you on linking scenes with buttons, displaying title screens, adding clocks and on-screen counters, the keep-up game that we started so long ago is starting to seem a little weak. Let's return home like the mighty conquerors we are, and jazz it up with some of the new things we've learned. Then, we'll go even further, and start incorporating 3D models built in an actual 3D art package into our game **Scenes**.