Timekeeper: In-Game Time Simulation and Event Scheduling.

Version: 1.0 (First release)

Timekeeper is a tool that can be used to simulate the passing of time within a video game.. The user can display one or many text objects in a scene showing the date and time and choose the speed at which time increments in-game.

The user can also select the unit of measurement for incrementing time. This means that time can progress forwards by seconds, minutes or hours.

In addition to simulating the passage of time, one of the more useful features of Timekeeper is its ability to extrapolate a future date and time and then schedule an event to occur when that time has been reached.

This means that a player could write their own logic and delay it for a specified amount of in-game hours, minutes, seconds, even days months or years if needed.

Skill Prerequisites.

At the moment, TimeKeeper's time simulation features are widely accessible in the inspector. Whilst scripting can be used, a user that is not comfortable with C# or at a basic level of experience could set this up using this documentation.

However, for event scheduling, this documentation assumes that the user has a basic understanding of C# and the Unity Scripting API. This by no means excludes people who are still learning and I am very happy to provide more support as needed.

In future versions I will be introducing more GUI-based functionality for people who do not wish to write lots of code. Having said that the classes in this asset reduce the amount of code that would be required to implement this functionality from scratch significantly.

For more help and support feel free to contact me using the contact details detailed in the '**Need More Help?**' section further down in this document.

Let's get started!

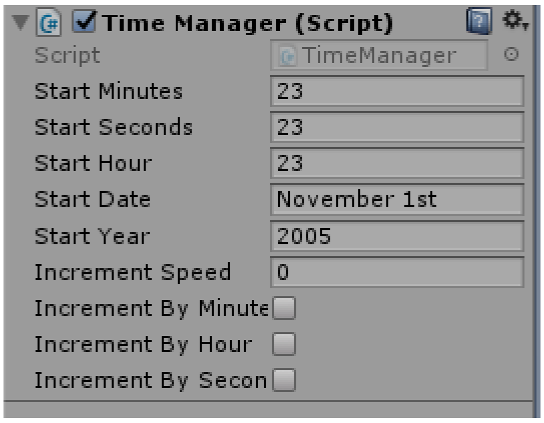
Displaying Time/Date Text in a Scene

With this asset. you use one or many Text objects to display the date and time. This text object is essentially a date calendar and clock combined and each one can move forward through time at its own speed. Moreover, each Text object can be used as a jumping off point for future events. (See the section '**Scheduling Future Events**' further down the document) For the purposes of this documentation I will refer to this object as the Time/Date Object.

Follow the below steps to show an incrementing Time/Date Text object in your game's scene.

1. Create a UI Text object in the scene's hierarchy.
2. Go to Assets/TimeKeeper/Scripts and find the 'TimeKeeper.cs' script
3. Attach this script to the Text object you created.

You should now see the TimeManager script's properties available to you in the Inspector when you select your text object in the hierarchy.


Time Manager (Script)

Script	TimeManager
Start Minutes	23
Start Seconds	23
Start Hour	23
Start Date	November 1st
Start Year	2005
Increment Speed	0
Increment By Minute	☐
Increment By Hour	☐
Increment By Secon	☐

You will need to set a minimum amount of the above properties in order to see a Time/Date Object in the scene and for it to increment. Otherwise if a property is left blank or invalid it will be set to a default value.

Start Minutes: The starting minutes value that the Time/Date Object text will display at runtime.

Eg: Adding '23' to this field will cause the text to show the time in minutes to be at :23 minutes at runtime.

Start Seconds: The starting seconds value that the Time/Date Object text will display at runtime.

Eg: Adding '23' to this field will cause the Time/Date Object text to show the time in seconds to be at :23 seconds.

Start Hours: The starting hour value that the Time/Date Object text will display at runtime. This is in 24-hour format.

Eg: Adding '15' to this field will cause the Time/Date Object text to show the time in minutes to be at 15:00 (or 3pm) at runtime.

Start Year: The starting year value that the Time/Date Object text will display at runtime.

Eg: Adding '1999' to this field will cause the text to show the year to be at 1999 at runtime.

Start Date: The starting day of the month that the text will display at runtime.

Note: The starting month can be invalid if you are not careful. The value must start with the month followed by a space, followed by the day within said month. This is the only currently support date format.

For example:

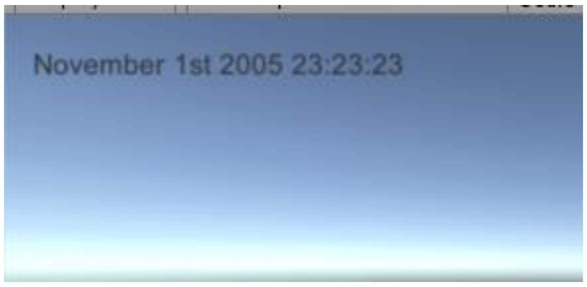
'October 1st' is valid while '1st October' is not and so the default value of '1st January' would be shown. Other formats like '01/10' would also be invalid. (So, to recap, the start month should be formatted as month + space + day)

A further note about minute and second values: Be careful when adding values for starting minutes or seconds which fall between midnight and 12 noon. These values must be preceded with a zero.

So, as an example, for 3am, you need to use 03 as this would be correct for the 24-hour clock. Using the number '3' on its own would be invalid in the current version.

(The 24-hour clock is currently the only hour format supported. In future releases this may change. Keep an eye on release notes in future versions to find out more.)

For the above example image showing the properties added within the inspector, the Time/Date Object text would display as below:



November 1st 2005 23:23:23

Increment speed: The speed at which the time will progress and visibly increment in the scene.

This value accepts float values.

Here are some examples of how different increment speeds would move time forward:

- Using a value of 1 would increment time forward every real-world second.
- Using a value of 1.5 would increment time forward every 1.5 real-world seconds. (Slower than above value)
- Using a value of 5 would increment time forward every 5 real world seconds (Slower than above value).
- Setting this property to 0 will freeze time and is the equivalent of pausing the clock

The key thing to understand from this is that a higher increment speed represents a slower passage of time. E.g.: increment speed of 0.25 is faster than increment speed of 6.

Increment by minute / second / hour: Defines the unit of time which will be incremented at the interval set by the Increment Speed property.

For example:

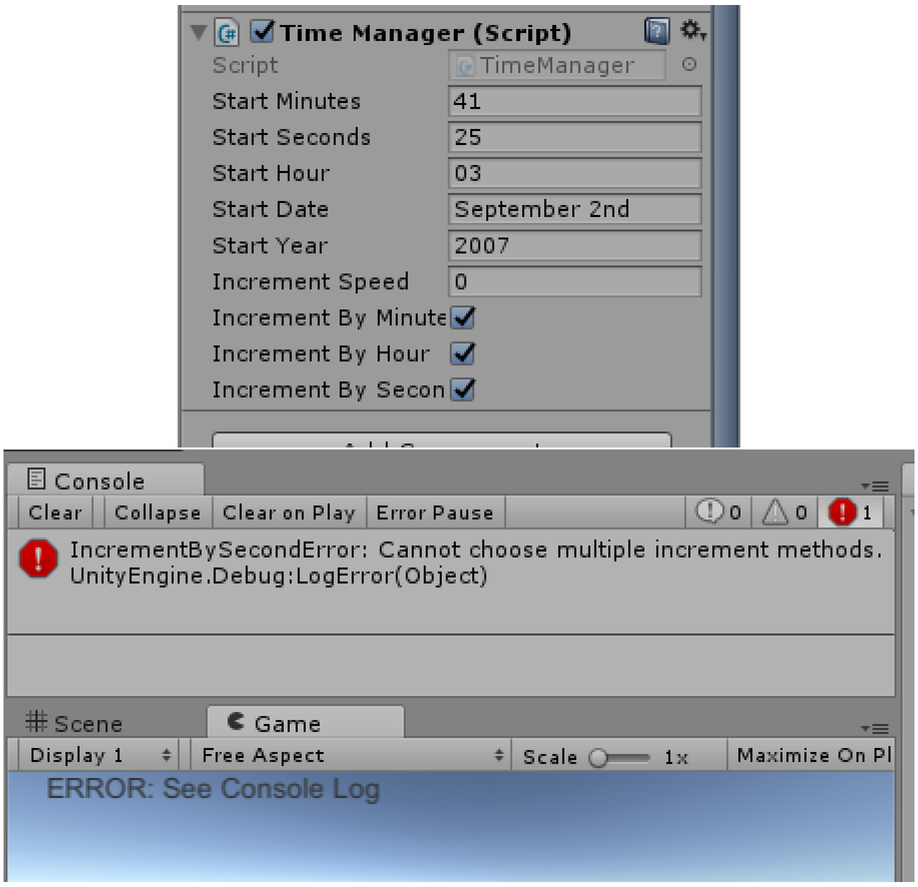
- Selecting 'Increment by Minute' will move time forward based on the minute value in the Time/Date Object. The seconds value will not be visible.
- Selecting 'Increment by Hour' will move time forward on the hour in the Time/Date Object. Minutes will remain at :00 and seconds will not be visible.

Please Note: Incrementing by a particular unit will still increment time units that are above it in terms of measurement. E.g.: if you are using 'Increment by Hour', when the hour hits midnight, the day value will still move forward in the Time/Date Object, as will the year if you hit midnight on December 31st.

Likewise, if you increment by minute, you will still see the hour and day move forward but no seconds will be displayed.

Inspector properties should be loaded before runtime. If you need to change an inspector property, stop the game, change the property and restart.

You cannot select multiple increment units. For example, if you were to select Increment by Hour, Increment by Minute and Increment by Second at the same time, an error will be thrown. (See below)



Time Manager (Script)

Script	TimeManager
Start Minutes	41
Start Seconds	25
Start Hour	03
Start Date	September 2nd
Start Year	2007
Increment Speed	0
Increment By Minute	✓
Increment By Hour	✓
Increment By Secon	✓

```
IncrementBySecondError: Cannot choose multiple increment methods.
UnityEngine.Debug:LogError(Object)
```

ERROR: See Console Log

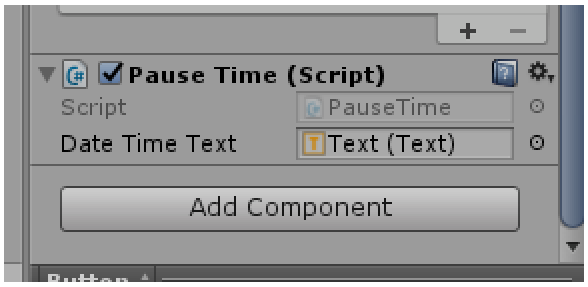
Using Pause, Speed up and Slow Down Buttons

Timekeeper has scripts which allow you to add functionality to buttons so that they can pause, speed up or slow down your Time/Date text objects.

Pause Button

After you have created a pause button somewhere in your scene, it's time to add the pause script and link it to the Time/Date Text object you wish to pause.

1. Go to Assets/TimeKeeper/Scripts and find the 'PauseTime.cs' script.
2. Attach the script to your pause button
3. In the inspector, add a reference to the Time/Date Object you wish to target with the pause button. (This can be done by clicking and holding on the Time/Date Object in the hierarchy and then dragging it to the 'DateTimeText' property in the Inspector. (See below)



Pause Time (Script)

Script	PauseTime
Date Time Text	Text (Text)

When the Pause button script is successfully linked to your Text object and you click it, it will record the previous increment speed before freezing time. When it is clicked again afterwards, it will set the speed back to the time it was before you clicked the button.

Speed Up and Slow Down

In the same way you can attach a pause button script to a button and pause a targeted Time/Date Object you can use the IncreaseTimeSpeed.cs and DecreaseTimeScreed.cs scripts to make a button speed up or slow down time when pressed.

E.g.: if you were to click a button to increase speed, the script will divide the value of the current increment speed by 2, increasing the frequency at which the time is incremented on the screen.

Likewise, using a button to decrease speed would multiply the increment speed by 2, making the time between each increment on the screen longer and therefore slowing down time.

To set these buttons up, follow the steps above for the Pause button, replacing the use of the PauseTime script with either IncreaseTimeSSpeed.cs or DecreaseTimeSpeed.cs as needed.

Scheduling future events.

This part of TimeKeeper's feature-set is current geared more towards users with coding experience. You will need to be able to get scripts attached to other gameobjects and access their public members (variables / functions / methods.) Although I have added a template coroutine which you can re-purpose indefinitely for as many future events as needed, you will need a basic (not advanced) understanding of coroutines and how they work.

See the below Unity documentation for more info on GetComponent and Coroutines.

https://docs.unity3d.com/Manual/Coroutines.html

https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html

You can use any of the Time/Date Object you have in your game to schedule a piece of code to run at any future date or time using a coroutine.

The JobManager Object

The coroutines used to fire future events should be added to the JobScheduler script attached to the JobManager gameobject. This gameobject can be found in the form of a prefab which already has the JobScheduler script attached. Add this prefab as a gameobject to your hierarchy.

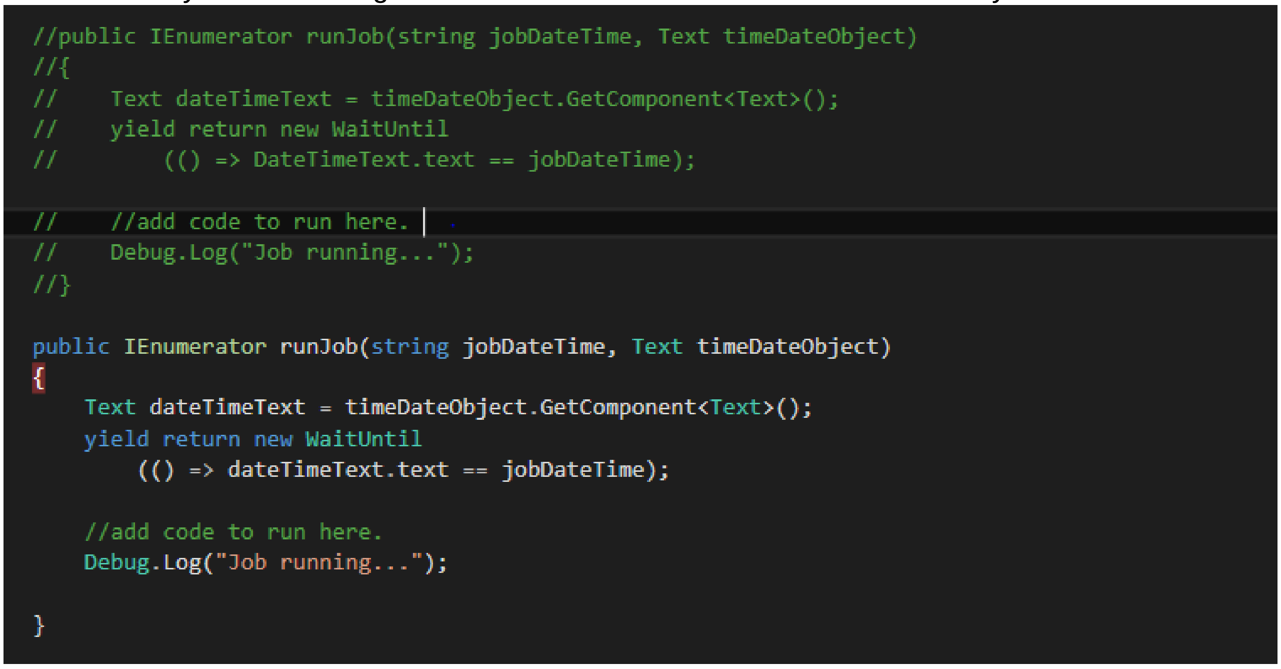
Your code will need to be placed into its own coroutine within the JobScheduler script which is itself attached to the JobManager object as mentioned above. The Jobcheduler script contains a commented example of a coroutine you can use to schedule your future code execution.

Follow the example steps below to set up a piece of code to run in the future.

1. As instructed above, add the JobManager prefab as a gameobject to your hirearchy.
2. Open up JobScheduler.cs in your code editor and copy the example, commented coroutine as shown below. (You will need to uncomment your copy.)

The text shown in green is the commented example. The code below that is the copied code uncommented.

It is this code you can add logic to and call as a coroutine from elsewhere in your codebase.



```
//public IEnumerator runJob(string jobDateTime, Text timeDateObject)
//{
//    Text dateTimeText = timeDateObject.GetComponent<Text>();
//    yield return new WaitUntil
//        (() => DateTimeText.text == jobDateTime);

//    //add code to run here.
//    Debug.Log("Job running...");
//}

public IEnumerator runJob(string jobDateTime, Text timeDateObject)
{
    Text dateTimeText = timeDateObject.GetComponent<Text>();
    yield return new WaitUntil
        (() => dateTimeText.text == jobDateTime);

    //add code to run here.
    Debug.Log("Job running...");

}
```

Setting up and calling the coroutine:

Give your copied coroutine a new name. For the above I used 'RunJob but you can use whichever signature you wish. (See '**Scheduling Multiple Future Events'** further down the document)

You will notice that your coroutine takes a string parameter of *jobDateTime.* This is a string containing the future date and time at which you wish your code to run.

For example, if the Time/Date Object measuring time is currently at January 1st 1999 22:00:32 and you wish to have some code to run in exactly 2 in game hours, you would pass in the string 'January 3rd 2000 22:00:32' for the parameter of jobDateTime.

The second required parameter is *Text timeDateObject*. This is the Time/Date Object you wish to use for triggering the event in the future. It is this object which the coroutine will check to see if it is time to fire your code.

The reason you need to specify which text object you want to base the future time on is that you may want to use the same coroutine with different Time/Date Objects.

Calling the coroutine when you want to schedule the action would look like this:

StartCoroutine(runJob("January 3rd 2000 22:00:32", DateTimeText));

This is saying that you want to use the runJob coroutine in the JobScheduler class (attached to JobManager) to run your code on January 3rd 200 at 22:00:32 and you are using your DateTimeText object (an instance of one of your Time/Date Objects) to evaluate if it is time to run the event.

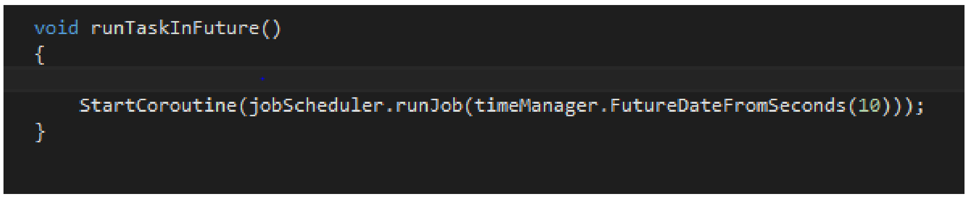
Calling the coroutine from your own class.

Here is an example where I am scheduling a new event using this coroutine from one of my own classes. I added a public Gameobject variable at the top of my class. This meant that I could drag the JobManager object into my class via the inspector. So, I now have an instance of the JobManager object called jobManager.

I then created a reference to the JobScheduler by using the Unity API's GetComponent function to get the JobScheduler script component from JobManager like so:

JobScheduler jobScheduler = jobManager.GetComponent<JobScheduler>();

This allows me to call any coroutines that I have added to the JobScheduler as shown below.



```
void runTaskInFuture()
{

    StartCoroutine(jobScheduler.runJob(timeManager.FutureDateFromSeconds(10)));
}
```

Notice that I did not add a literal string for my jobDateTime parameter. I instead used a function contained within an instance of TimeManager.cs to get the string literal for the Future Date and Time as it will appear in 10 hours.

timeManager.FutureDateFromSeconds(10);

The function was FutureDateFromSeconds(int hours). See the '**Extrapolation Functions**' section further on in the documentation for guidance on how to use these to easily get a future date and time in a specified number of hours, minutes or seconds.

Adding your code to your coroutine

It is no use scheduling a coroutine to run your code if you don't have any of your own code to run in the future!

You can add your logic in the form of a method or any other valid C# within a coroutine as shown in the commented example below:

```
public IEnumerator runJob(string jobDateTime)
  {
      yield return new WaitUntil
         (() => DateTimeText.text == jobDateTime);
      //add code to run here.
      Debug.Log("Job running..."); }
```

The code highlighted in bold in the example above is where you need to add your own code. In the above example, the string "Job running " is printed to the console when the jobDateTime passed into the coroutine is reached.

You could add your logic in the form of a method elsewhere in your codebase and then just call the method within the coroutine in the same place shown above. For example::

```
public IEnumerator runJob(string jobDateTime)
   {
       yield return new WaitUntil
           (() => DateTimeText.text == jobDateTime);


       //add code to run here.
       DoStuff();
   }
```

Note: When using methods for your logic within your coroutine, the methods should usually be void. You could use a function to return a value if it is useful to you further down in the code within your coroutine.

If you wanted to use a method that has parameters in the coroutine, you would need to add a parameter to the coroutine to be passed in when it is called unless the parameter is a value that you can access at the point of execution.

Scheduling Multiple Future Events

It is very likely that you will have need for more than one coroutine as you will want o schedule many different events. If you are scheduling multiple of the same event at different times you can use the same coroutine. (I could use 'RunJob' multiple times for the same event at different points in the future) For other coroutines you should create a new copy of the template (commented version shown above), change its name and add your logic. Having multiple coroutines with the same name will cause a compile error.

As an example. I might name one coroutine 'public IEnumerator startAttack(string jobDateTime' and another 'public IEnumerators spawnEnemies(string jobDateTime)'

In summary, it is best to name your coroutine after the job it is doing. For the purposes of the template, I added a coroutine called 'runJob' but it can be called anything.

Extrapolation Functions (Functions for calculating future dates and times)

As explained above, when you schedule a future piece of logic using a coroutine, you need to pass in a string which represents the Date and Time exactly as it would look in the Time/Date Object at that point in the future.

However, you do not need to work this out in your head. If you wanted to get the string for a future date/time in say three hours or 40 seconds, you can use the below functions from TimeManager.cs to get this value as a string and pass that into your coroutine.

To use the below functions, reference any instance of the TimeManager script attached as a component to a Time/Date Object

string FutureDateFromDays(int days

Gets the future date in days for the amount of days passed in.

Eg: *StartCoroutine(runJob(FutureDateFromDays(4)));*

The above will set a coroutine JobScheduler to run some code in 4 in-game days.

string FutureDateFromHours(int hours)

Gets the future date in hours for the amount of hours passed in.

Eg: *StartCoroutine(runJob(FutureDateFromHours(4)));*

The above will set a coroutine JobScheduler to run some code in 4 in-game hours.

string FutureDateFromMinutes(int minutes)

Gets the future date in minutes for the amount of minutes passed in.

Eg: *StartCoroutine(runJob(FutureDateFromMinutes(4)));*

The above will set a coroutine JobScheduler to run some code in 4 in-game minutes.

Will throw an error if called while incrementing by hours as minutes will not be available to use for calculating the future date.

string FutureDateFromSeconds(int seconds)

Gets the future date in seconds for the amount of seconds passed in.

Eg: *StartCoroutine(runJob(FutureDateFromSeconds(4)));*

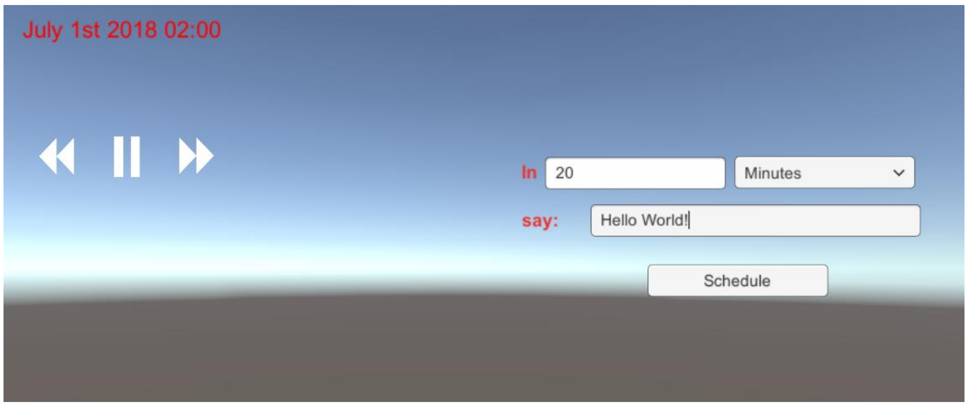
The above will set a coroutine JobScheduler to run some code in 4 in-game seconds.

Each of the above functions can be found in any instance of TimeManager.cs

You must be incrementing time by seconds to use this function, otherwise an error wil be thrown as seconds would not be an available unit for use in future date calculation.

Demo Scene

I have included a demo scene which demonstrates the use of a single Time/Date Object. I have added some input fields for you to test the scheduling of an event which simply prints some text to the screen at a certain point in the future.



You can use the GUI elements I have added to do something like 'In 20 minutes, say "Hello world!"' If you were to then click the 'Schedule' button, the text will be printed to the screen in 20 in-game minutes.

You will notice that there is a second coroutine in JobScheduler below 'runJob.' This is another example of a coroutine that I have added for the demo scene.

Pause Button: Sprite Swapping Script and Sprite Files

I have also included a script which will allow you to toggle your pause button image when clicked. The SpriteSwap script can be attached to your pause button. In the inspector, simply drag in the sprite you want to display when time is paused and the sprite you wish to display when it is resumed.

There is also a sprites folder containing several PNG images you can use for your Pause button (either directly on the button or in the SpriteSwap script attached to the pause button) the Increase Time button and the Decrease Time button.

Need more help?

If you feel like something in this document is not clear or not included and you need to get further assistance with something please feel free to get in touch via the support website detailed on the Unity Asset Store page for this asset.

I'm always very happy to help over email and welcome feedback always. Feature requests will always be carefully considered.

Future planned features.

I am supporting future development of TimeKeeper as of the time of writing this document and I hope to include the following features in future:

- Leap years
- Toggle between 24-hour and 12-hour clock formats.
- Simulation of time moving backwards
- GUI-based event scheduling.