



This cached version of the tutorial may be outdated.

For the latest version, please visit [the live website](#).

TUTORIAL

OBJECTIVE

In this tutorial, you will learn how to use all of the functionalities of Chronos by enabling time-control in a small existing platformer game. For that purpose, we'll use the Unity 2D platformer demo. The concepts you will learn apply to any type of game, of any genre, in 2D or 3D.

Before getting started, we recommend watching the [2D Game Development Walkthrough](#) video from Unity, which gives an overview of the project structure for the platformer demo that we'll be modifying.



This tutorial assumes that you have a basic knowledge of Unity and C#. If not, please see their respective tutorials or documentation before starting.

END RESULT

CLICK TO PLAY

You can play the final game in the web player above. It is essentially the same game as the platformer demo, except that you can manipulate time for enemies, and any bomb you throw will have a "force field" that will slow (and therefore trap!) all enemies around it before exploding.



Pssst... You start the game with 100 bombs for testing. To turn off the music (so you aren't driven insane during the tutorial), just pause time!



Controls

W / ↑: Jump
A / ←: Left
D / →: Right
S / ↓: Place Bomb
Space: Shoot
1: Rewind
2: Pause
3: Slow
4: Normal
5: Accelerate



Overview of the components

Before diving head first into implementing Chronos for our platformer, it's worth taking a minute to understand how each of its parts works.

Just like any Unity game, Chronos works with components that you attach to GameObjects. In total, it provides 6 easy components that, when combined correctly, make the magic happen. In this section, we'll cover each of them briefly; we'll get back to the details later in the tutorial.









Don't worry if you have trouble understanding the whole structure of Chronos at first! It will become much clearer once you start experimenting with it. In fact, using Chronos is much simpler than it seems.

At its root, Chronos requires a single **Timekeeper** object, which is basically just a registry for global clocks. It must be present in every scene.

More importantly, Chronos functions with **Clocks**. A clock is exactly what you would expect it to be: a component that ticks at every frame (based on its speed, i.e. its *time scale*) and gives you time measurements. In Chronos, there are 3 types of clocks: global, local, and area.

To 'read' time measurements from one or many clocks, each GameObject must use its own **Timeline** component. Quite literally, this means every object can progress on its own specific timeline, instead of just one global timeline. This allows for a variety of interesting time effects and gameplay mechanics.

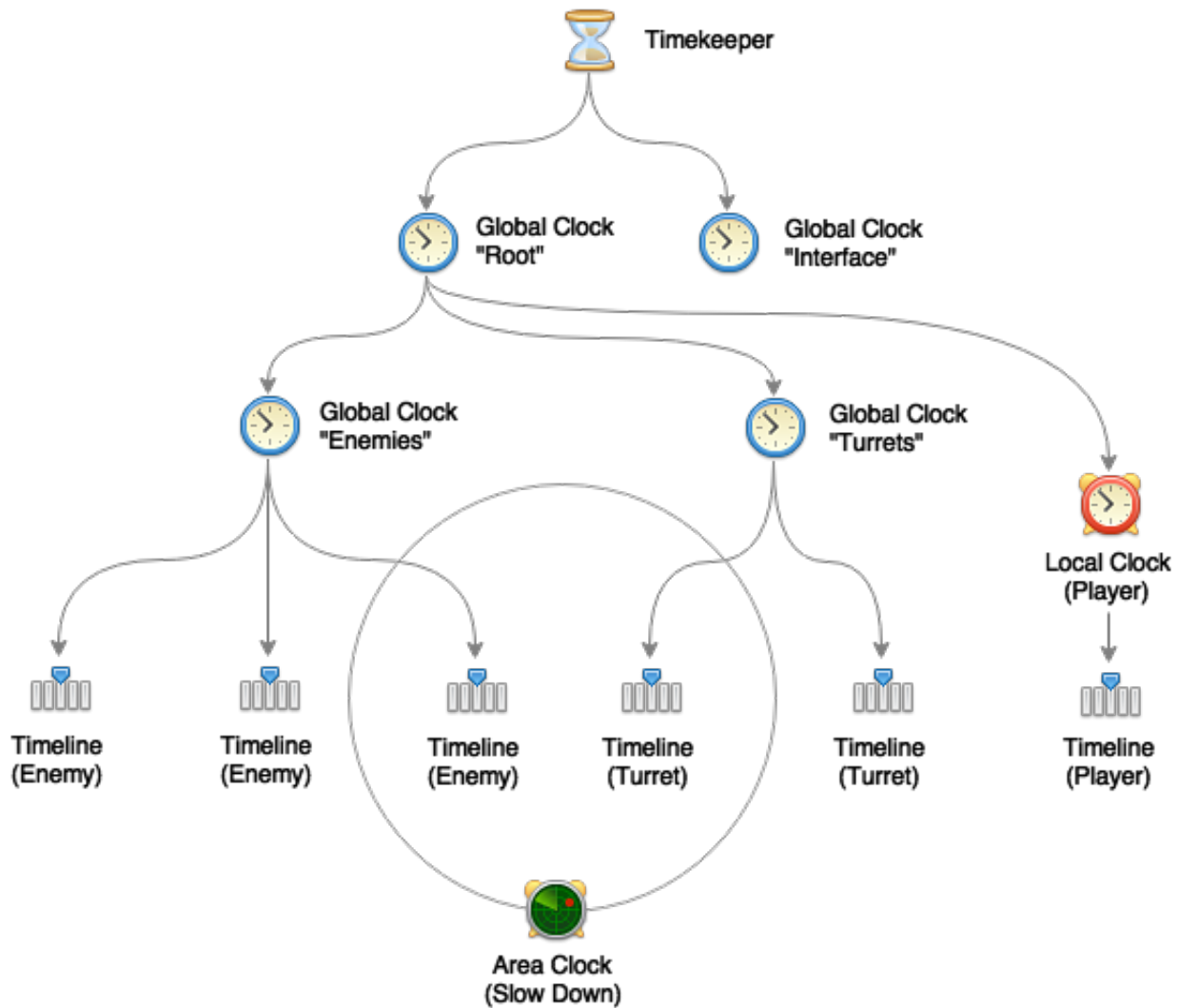
	Component	Description
	Timekeeper	The timekeeper is a singleton tasked with keeping track of global clocks in the scene. One and only one Timekeeper is required per scene.
	Clock (Abstract)	The base class for every other type of clock. It has a <code>timeScale</code> property to adjust at what speed its time progresses. Any clock can also have a parent global clock with which it will multiply its own time scale.
	Global Clock	Global Clocks are clocks that can be accessed and referenced from anywhere by a unique string key (e.g. "Enemies" , "World" , etc). They must always be attached as components of the Timekeeper object. They are the only kind of clock that can be set as parents of other clocks. They can be used to apply time effects to groups of objects.
	Local Clock	Local Clocks are clocks that apply only to the GameObject to which they are attached. They can be used to make individual objects progress at different speeds.
	Area Clock	Area Clocks are clocks that affect all GameObjects within their collider. Their effects stack with any other clock these objects might be observing, and with any overlapping area clock. They can be used to apply time effects to a restricted area.
	Timeline	A Timeline combines measurements from one or more clocks to provide a single but independent time, delta time and time scale for every GameObject. It should be attached to every GameObject that should be affected by Chronos.



You can click on the icon of any component in the previous table to access its documentation.

EXAMPLE

Here's a diagram that shows an overview of how a sample Chronos scene could be configured, in this case a tower defence game:



So, what's going on in this Chronos setup? First, we have our Timekeeper singleton at the top. In it, we registered 4 global clocks (Root, Interface, Enemies and Turrets) with parenting.

The Root and Interface clocks are separate, because we want our user interface elements to behave on a constant time scale (most likely always 1), without being affected by the game's time scale. This way, for example, we can pause the game by setting the Root clock's time scale to 0 and still have an animated menu pop out.

We then have two children of the Root clock, Enemies and Turrets. In our tower defence game, we probably want to be able to affect enemies independently from turrets. This would allow us, for instance, to have an ability that accelerates all turrets while maintaining the enemies at their regular speed.

Then, we configured a local clock that only affects the player avatar. Since there is only one player in the scene at once, there is no need for a global clock.

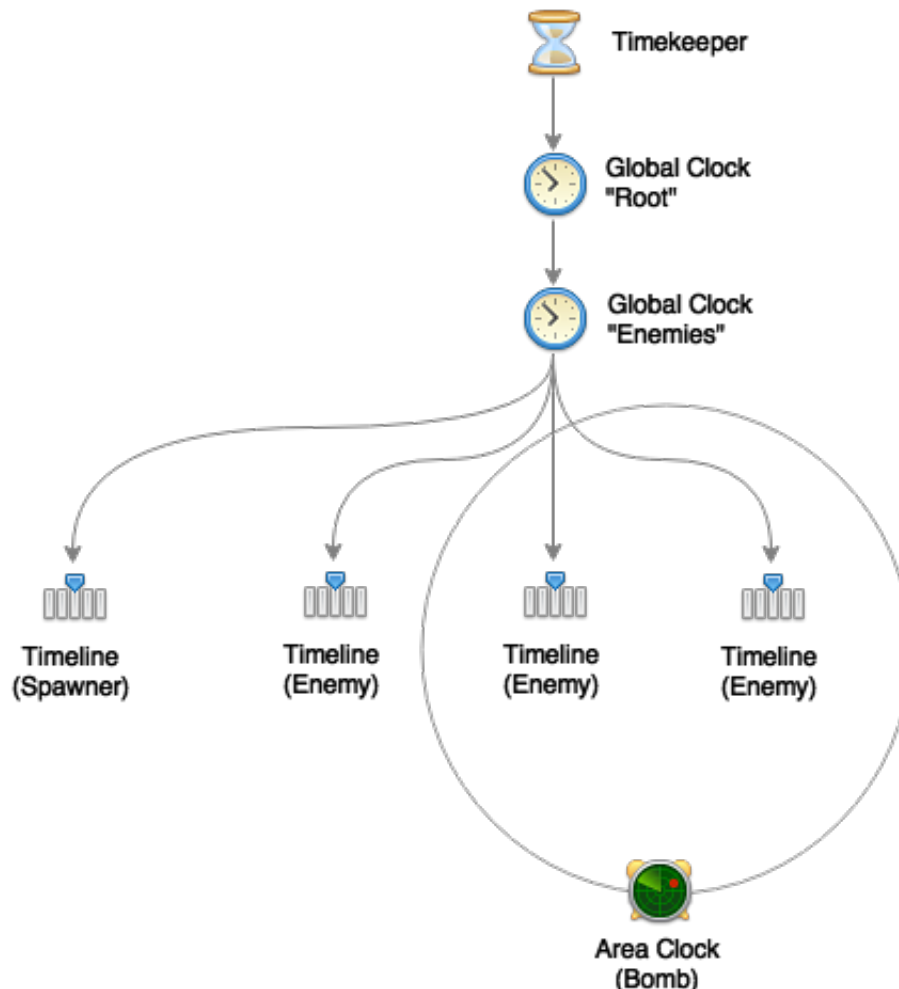
Each of our GameObjects (3 enemies, 2 turrets and 1 player) has its own timeline. This timeline observes any clock to which we made it point and returns the combined measurements that the GameObjects use for gameplay.

Last but not least, at this moment in the game, we used an ability that created a spherical area clock that encompasses the third enemy and the first turret. By setting this clock's time scale between zero and one, these objects will be slowed down.

BACK TO THE TUTORIAL

If all that seems overwhelming, don't worry. Indeed, adding time control to your game is literally a whole new dimension to think about. However, Chronos is built to make that process as easy as possible. We will go through each step in the course of this tutorial.

The structure we will create in this tutorial is a simplified version of the one above. Basically, we will only affect enemies and their spawners. All other objects in the scene (player, missiles, background elements, etc.) will be left untouched, meaning they will progress at their regular speed through Unity's `Time` class. Here is the diagram for the structure we will implement:

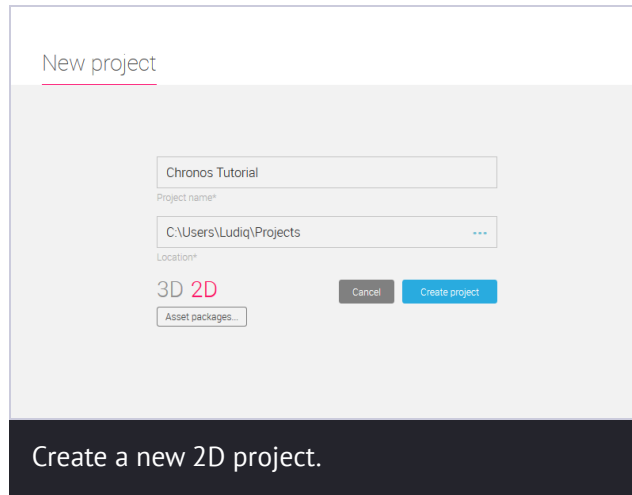


If you've made it this far, you've completed the hardest part of the tutorial – congratulations! Now go grab some coffee, and let's control time!



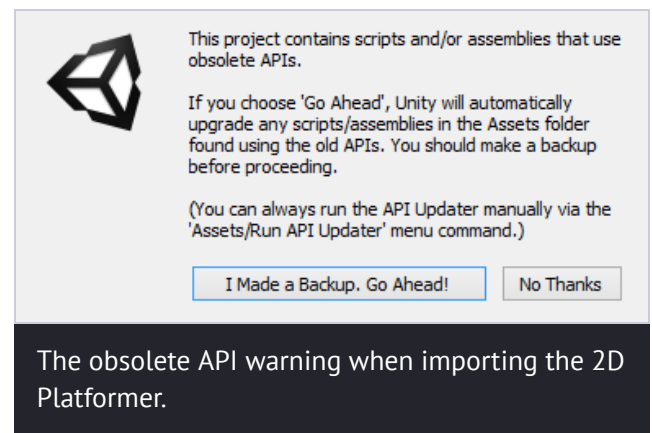
Creating the project

For the purpose of this tutorial, create a new blank Unity project like you usually would. Choose any project name and location and switch the preset to 2D. Next, you'll need to import two packages: the platformer demo and, of course, Chronos. The asset store links to both are below.



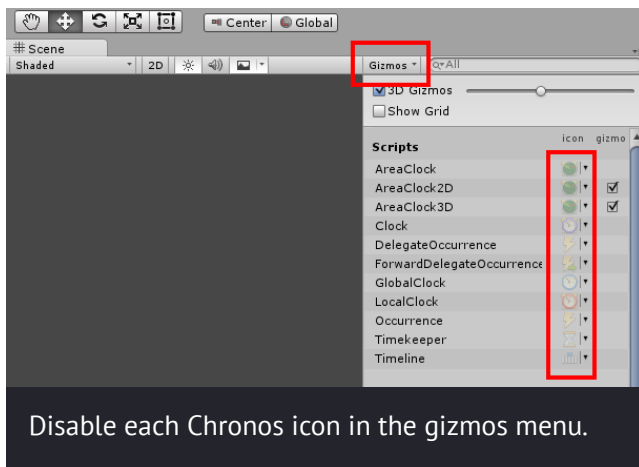
Import the packages in the root assets directory of your newly created project. For the Chronos package, make sure that the **Gizmos** directory is not in any subfolder – otherwise, icons in the scene view may not render properly.

You might get a warning about the platformer asset being made with obsolete APIs, because that package was built for Unity 4.3. Click *I Made a Backup. Go Ahead!* and let Unity port it for you.



If you plan on using Chronos from **UnityScript** (JavaScript), move the **Chronos** folder to **Plugins/Chronos**. That will tell Unity to compile the Chronos components *before* your own scripts, so that the latter can access the former correctly. This is not required for C# users.

As a last step before we get started, disable the script icons from Chronos so that they do not appear in the scene view. To do so, click on the **Gizmos** dropdown in the top right corner of the scene view, then click on the



icon of every Chronos script to render each one faded out. Do not uncheck the gizmo checkboxes.

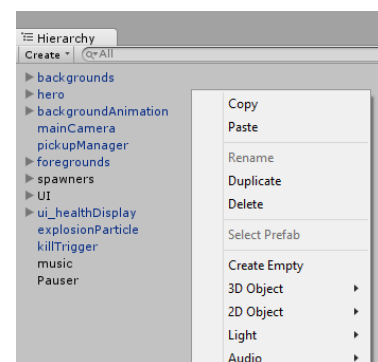
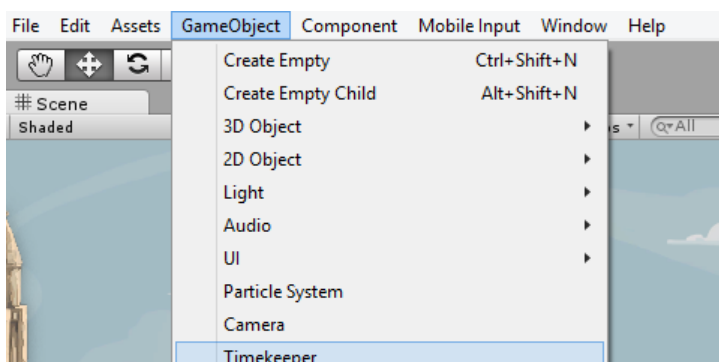
Finally, open up the `Scenes/Level` scene that contains the platformer. Our project is now properly set up and we are ready to use Chronos!

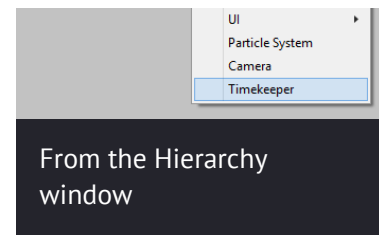
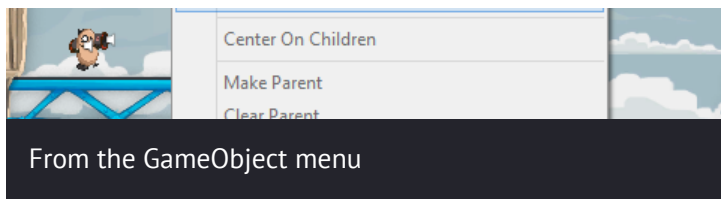


Creating a Timekeeper

The first thing you want to do in any Chronos scene is to create a `Timekeeper` object. Think of it as the main camera, but for time: without it, nothing will function properly.

Creating a Timekeeper is as easy as creating any built-in Unity object. You can create one from the `GameObject` menu, by right-clicking any empty space in the hierarchy window.





Once the timekeeper is in your scene, there is no further configuration needed! We can start adding some clocks.



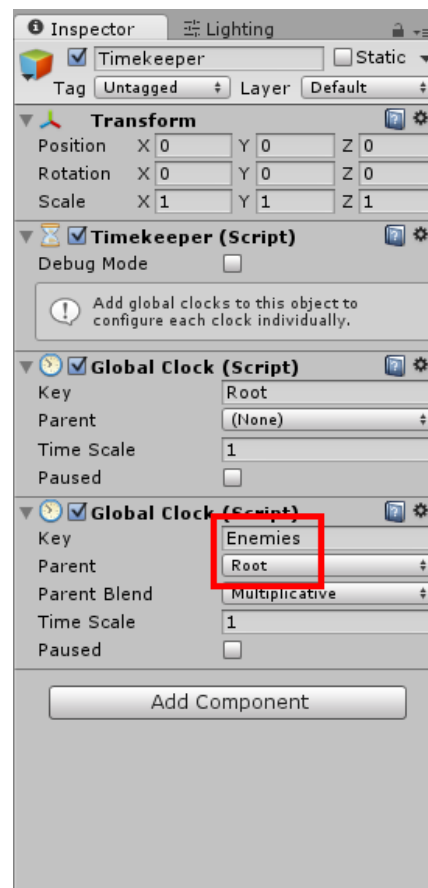
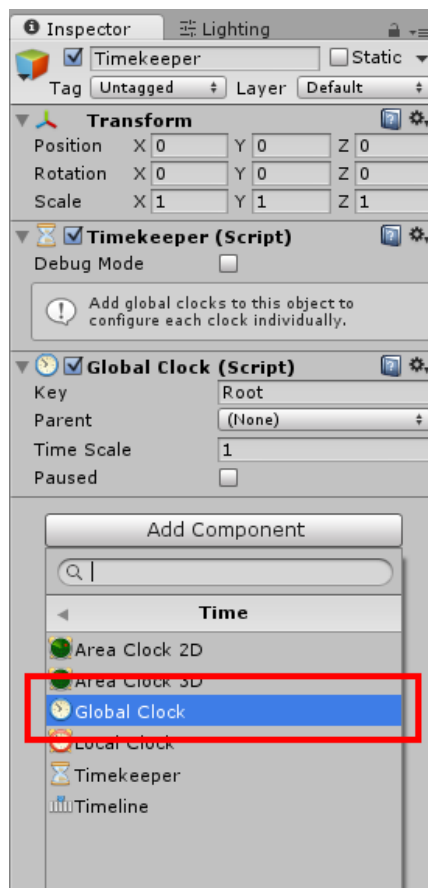
For more information about the **Timekeeper** class, see the [Documentation](#).



Adding a Clock

Select the Timekeeper to display its inspector. There, you will see that it already contains a **Root** clock by default. Let's add our clock by clicking the Add Component button, then choosing **Time > Global Clock**.

Then, set its key to **Enemies** and its parent to **Root**. Since we want our enemies to start at normal speed, there is no need to adjust the **Time Scale** or **Paused** properties.



Add a global clock.

Set its properties.



For more information about the `GlobalClock` class, see the [Documentation](#).



Adding Timelines

ENEMIES

Now that we have our global clock, we need to tell our enemies to look at it! For that, we only need to add a Timeline component to the enemy prefabs.

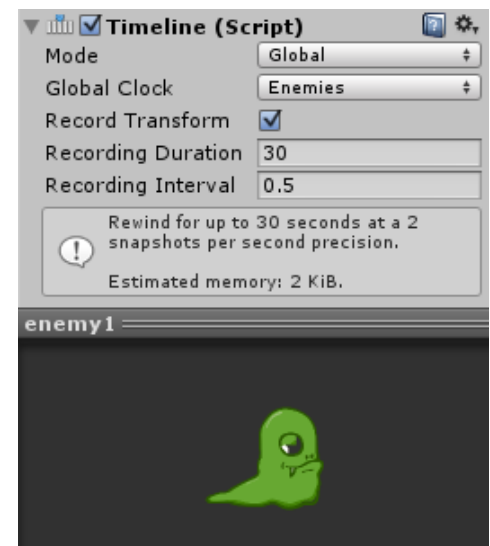
Locate the enemy prefabs: they should be located at `Prefabs/Characters/enemy1` and `Prefabs/Characters/enemy2`. Using the same method as in the previous step, add a Timeline component to each.

Timelines have two modes: one for observing a local clock, and one for observing a global clock. Since our case is the latter (we want to point at the "Enemies" clock we just created), set the mode to Global, and the Global Clock to "Enemies".

The timeline will automatically take care of synchronizing the animation on our enemies, along with the the transform and physics (make sure you check the `Record Transform` option), with its time scale. It will also allow us to use timing measurements in scripting later on.

Feel free to play around with the recording time and interval properties; these will affect how long your rewinds can be, and how precise they feel.

Allowing more rewind time or precision takes more memory (RAM). The default values should offer a decent duration and precision for most cases, but don't be afraid to increase them! Chronos is well optimized in that regard, so you should almost certainly be able to meet the needs of your game. The inspector will instantly show you how much memory will be allocated by the timeline.



Chronos supports multiple object editing for all its components. For example, you can select the `enemy1` and `enemy2` prefabs at the same time and create both timelines at once.

MUSIC

We also want to give an audio feedback to the player so they understand how time is progressing. For this reason, we will synchronize the music with the time scale of our enemies. To do that, simply locate the `music` object in the scene hierarchy, and attach at timeline to it, pointing at the same "Enemies" clock. That's it! Chronos will take care of adjusting the pitch of the audio source for you.



For more information about the `Timeline` class, see the [Documentation](#).



Controlling time in script

Still there? Have you finished that coffee yet? No? Good. Our component setup is now done. Now, it's time to get our hands dirty with scripting.

CREATING THE BASE BEHAVIOUR

The first thing we'll do is create a base behaviour that offers a useful shortcut to use the Timeline component. It's an optional one-time operation, but it will make your code much more legible.

When using Chronos, you will almost always work with the Timeline class, attached as a component to your game objects. It provides all the time measurements that you need, such as `deltaTime` and `timeScale`, and a few more utility methods that we'll cover later on.

Just like you use `transform.position` as a shortcut of `GetComponent<Transform>().position`, you'll be able to use `time.deltaTime` instead of the verbose `GetComponent<Timeline>.deltaTime`.

Create a new script under `Scripts` called `BaseBehaviour`, and give it the following code:

```
using UnityEngine;
using Chronos;

public class BaseBehaviour : MonoBehaviour
{
    public Timeline time
    {
        get
        {
            return GetComponent<Timeline>();
        }
    }
}
```



From now on, **any script inheriting from BaseBehaviour** will have access to the `time` shortcut property.

SETTING THE TIME SCALE VIA INPUT

Next, we want to change the time scale of our Enemies clock by pressing the keys 1 through 5 on the keyboard, from rewind to accelerate. Luckily, since we have done our setup correctly, this is a very trivial script.

Create a new script under `Scripts` called `TimeControl`, and add give it the following code:

```

using UnityEngine;
using Chronos;

public class TimeControl : MonoBehaviour
{
    void Update()
    {
        // Get the Enemies global clock
        Clock clock = Timekeeper.instance.Clock("Enemies");

        // Change its time scale on key press
        if (Input.GetKeyDown(KeyCode.Alpha1))
        {
            clock.localTimeScale = -1; // Rewind
        }
        else if (Input.GetKeyDown(KeyCode.Alpha2))
        {
            clock.localTimeScale = 0; // Pause
        }
        else if (Input.GetKeyDown(KeyCode.Alpha3))
    }
}

```



If you want to make the time scale change smoothly, you can use the [LerpTimeScale](#) method.

Attach this script to any GameObject in the scene. In this case, it would make sense to attach it to the Timekeeper. Now, try playing the game and pressing the keys. You'll notice things don't work quite like they should...

- Accelerating and slowing down time doesn't work
- Pausing time makes enemies ignore gravity
- Rewinding offers some unpredictable behaviour

So, what's wrong? Well, the enemy script is trying to move without any knowledge of Chronos. Luckily, it's an easy fix.

MODIFYING THE ENEMY SCRIPT

Chronos will take care of the animations, physics, audio and particle effects for you. However, you still need to make sure your scripts take into consideration the timing measurements that Chronos calculates!

In the platformer demo, the enemy movement is controlled by setting their velocity in the X axis. This occurs in the file `Scripts/Enemy` at line 48 (line breaks added for legibility):

```
GetComponent<Rigidbody2D>().velocity = new Vector2  
(  
    transform.localScale.x * moveSpeed,  
    GetComponent<Rigidbody2D>().velocity.y  
);
```

We previously used a timeline on our enemy to enable rewinding. The problem here is that the timeline we attached has its own properties and methods for rigidbodies, and therefore becomes confused! We are *bypassing* Chronos by setting the rigidbody's velocity directly. For it to work correctly, we have to tell *the timeline* – not the rigidbody – to change its velocity:

```
// Use Timeline instead of Rigidbody2D  
time.rigidbody2D.velocity = new Vector2  
(  
    transform.localScale.x * moveSpeed,  
    time.rigidbody2D.velocity.y  
);
```



We tried to minimize the amount of code change required to integrate Chronos into a project. However, there are some quirks such as this one that we couldn't avoid.

For a list of all the small changes needed to migrate your script to Chronos, see the [Migration](#) guide.

When rewinding, the timeline will interpolate between recorded snapshots and apply them. In that state, the rigidbody isn't really a physical object anymore; it's set to kinematic. For this reason, if we try change its physical properties, Chronos will throw an exception. Therefore, we need to add an if clause around that movement line to verify that time is going forward:

```
if (time.timeScale > 0) // Move only when time is going forward  
{  
    time.rigidbody2D.velocity = new Vector2  
    (  
        transform.localScale.x * moveSpeed,  
        time.rigidbody2D.velocity.y  
    );  
}
```

That's it! Notice that we used our `time` shortcut, so we need to change the base class of the script (and of course, add Chronos as a using):

```
using UnityEngine;
using System.Collections;
using Chronos;

public class Enemy : BaseBehaviour // ...
```



Recommended: For more examples of using the `Timeline` class, see the [Documentation](#).

If you play the game now, enemies should move as expected in regards to time. There is one last detail that's wrong, however: the spawner keeps spawning at the same speed no matter the time scale, and enemies rewound up to the moment of their spawn are not destroyed! Let's fix that.

MODIFYING THE SPAWNER

First, add a timeline component to the spawner prefab (`Prefabs/spawner`) like we did for the enemies. Set it to observe the global "Enemies" clock as well. There is no need to record the transform here as it won't move.

Open up the spawner script (`Scripts/Spawner`) in your code editor. It should look like this:

```
using UnityEngine;
using System.Collections;

public class Spawner : MonoBehaviour
{
    public float spawnTime = 5f;           // The amount of time between each spawn.
    public float spawnDelay = 3f;          // The amount of time before spawning start
    public GameObject[] enemies;           // Array of enemy prefabs.

    void Start ()
    {
        // Start calling the Spawn function repeatedly after a delay .
        InvokeRepeating("Spawn", spawnDelay, spawnTime);
    }

    void Spawn ()
    {
        // Instantiate a random enemy.
```

We can see that the spawning timing is done through the `InvokeRepeating` method. Chronos doesn't provide an equivalent for this method out of the box, because a simple coroutine can do the same thing – with

more performance and flexibility.

For that reason, let's first change our code to do exactly the same thing, but with a coroutine:

```
using UnityEngine;
using System.Collections;

public class Spawner : MonoBehaviour
{
    public float spawnTime = 5f;           // The amount of time between each spawn.
    public float spawnDelay = 3f;          // The amount of time before spawning start
    public GameObject[] enemies;           // Array of enemy prefabs.

    void Start ()
    {
        // Start calling the Spawn coroutine.
        StartCoroutine(Spawn());
    }

    IEnumerator Spawn ()
    {
        yield return new WaitForSeconds(spawnDelay); // Wait for the delay
    }
}
```

At this point, Chronos still isn't passing timing calculations to the spawner. That's because we're using Unity's `WaitForSeconds` method, which only respects Unity's `Time.timeScale`, a value we haven't even touched. That means our spawner will execute at normal speed no matter what time scale we assign to our Enemies clock.

Fortunately, converting any coroutine to Chronos is easy. Simply replace `new WaitForSeconds()` by `Timeline.WaitForSeconds`. No further work is needed. The updated version of our spawner component is below (notice the using and the base behaviour):

```

using UnityEngine;
using System.Collections;
using Chronos;

public class Spawner : BaseBehaviour
{
    public float spawnTime = 5f;           // The amount of time between each spawn.
    public float spawnDelay = 3f;          // The amount of time before spawning start
    public GameObject[] enemies;           // Array of enemy prefabs.

    void Start ()
    {
        // Start calling the Spawn coroutine.
        StartCoroutine(Spawn());
    }

    IEnumerator Spawn ()
    {

```

For our spawner to be coherent in time, the last thing we need to do is make it able to despawn enemies once they're rewound up to the moment of their instantiation. Chronos makes this process simple with a concept called Occurrences.



Using Occurrences

Chronos introduces a powerful new scripting concept called Occurrences. An occurrence is basically a two-way event (forward and backward) scheduled to any point in time. It allows any piece of code to become easily rewindable without having to think of any time calculation.

Note that occurrences can also be one-way (forward only), for games that don't require rewind but would still like to take advantage of effortless planning. For example, no need to use a coroutine with `WaitForSeconds` to execute code in the future – just use `time.Plan(delay, YourMethod)`.

Using an occurrence, we will easily be able to despawn our enemies. However, you should first read the occurrence documentation and examples (it takes about 10 minutes) to better understand how occurrences work in simple scenarios. Click here to access the documentation:



Please read: [Occurrence Documentation & Examples](#)

Don't worry, we'll be waiting for you right here:

```
yield return WaitForFinishReading(documentation.occurrences);
```

Done reading? Cool. Now that you understand occurrences, let's change our enemies instantiation code from the spawner script. The line we want to change is this one:

```
Instantiate(enemies[enemyIndex], transform.position, transform.rotation);
```

Instead of this line, we want to instantly execute an occurrence that will instantiate the enemy, and destroy it when time rewinds to the current time. This occurrence will be repeatable, because if we let time flow normally after a rewind, we want that enemy to respawn at the same moment. Since this is a one-time piece of code in our game, there's no need to subclass the Occurrence class; we'll just use the delegates shortcut. Here is the framework:

```
time.Do // Instantly
(
    true, // Repeatable

    delegate() // On forward
    {
        // Spawn the enemy.
        // Must return a state transfer object.
        // In this case, we will return the spawned enemy.
    },

    delegate(object transfer) // On backward
    {
        // Destroy the transfered spawned enemy.
    }
);
```

The instantiation code stays exactly the same, except that we return the instantiated GameObject to be passed as `transfer` to the backward delegate. When rewinding, we'll use a simple Destroy on that GameObject.

```

time.Do
(
    true, // Repeatable

    delegate() // On forward
    {
        GameObject enemy = Instantiate(enemies[enemyIndex], transform.position, tra
        return enemy;
    },

    delegate(GameObject enemy) // On backward
    {
        Destroy(enemy);
    }
);

```

That's it! The final code for the spawner is below:

```

using UnityEngine;
using System.Collections;
using Chronos;

public class Spawner : BaseBehaviour
{
    public float spawnTime = 5f; // The amount of time between each spawn.
    public float spawnDelay = 3f; // The amount of time before spawning start
    public GameObject[] enemies; // Array of enemy prefabs.

    void Start ()
    {
        // Start calling the Spawn coroutine.
        StartCoroutine(Spawn());
    }

    IEnumerator Spawn ()
    {
        yield return time.WaitForSeconds(spawnDelay); // Wait for the delay
    }
}

```



Don't forget to add a timeline to the spawner prefab!

We are now done with enemies and spawners. Take some time to play your awesome time-controlled platformer and stretch your legs. When you're done, we'll finish in style with one of the coolest features Chronos has to

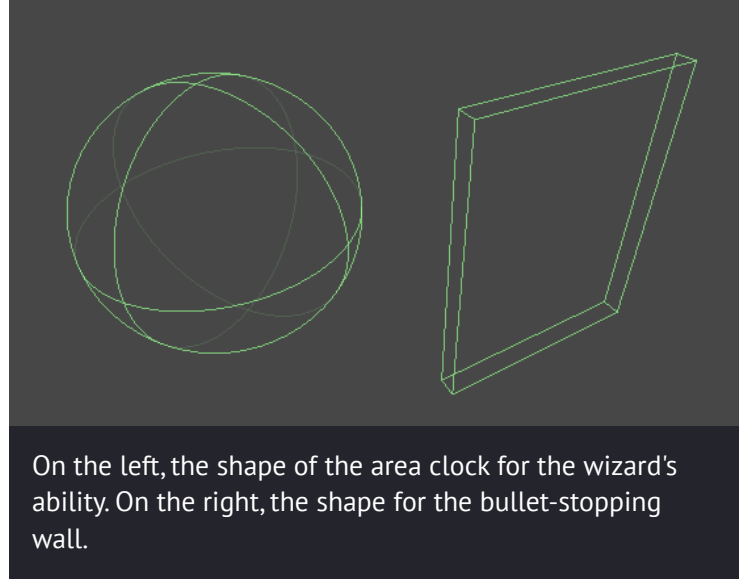
offer: area clocks.



Adding an Area Clock

Weren't you amazed the first time you saw Neo [stop bullets](#) in the Matrix? Or when you saw the Wizard's [slow time bubble](#) in Diablo III? All of this (and more) is possible with Chronos' Area Clock component.

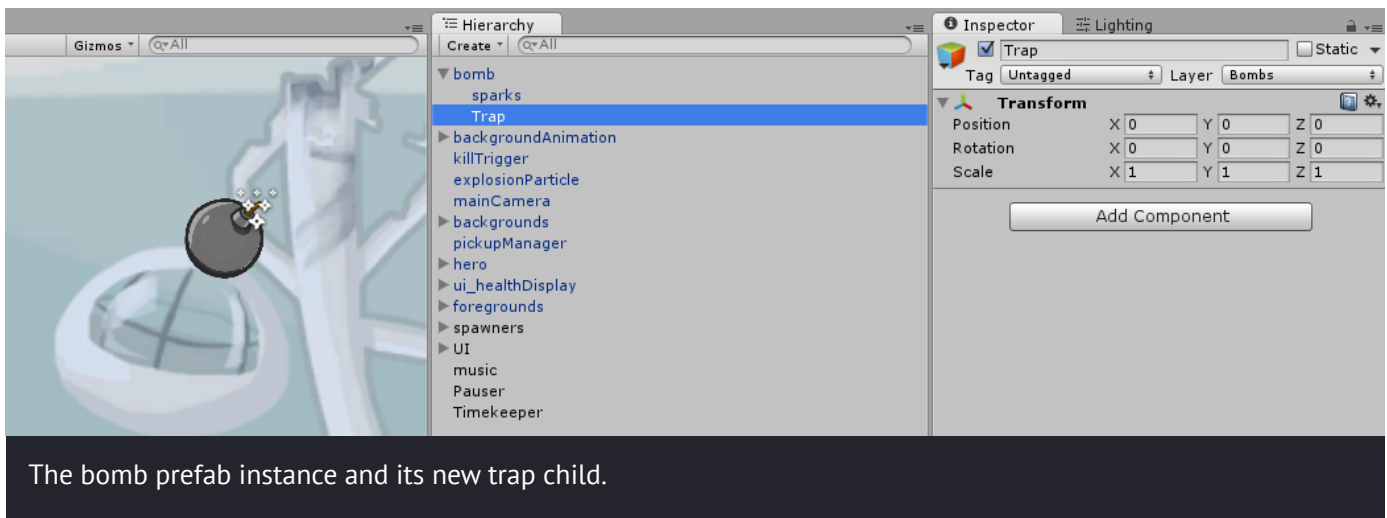
Area clocks are clocks that affect only the objects that enter their collider. They can take any (convex) shape and size, in 2D or 3D, and apply a time scale within. For example, stopping bullets could be implemented by a high, wide and thin cubic area clock with a time scale of 0. The slow time bubble could be created with a spherical area clock with a low time scale (probably around 0.25).



Because bullets have speeds that are too high for Unity's physics engine, they will not be stopped automatically by a thin area clock – they'll go right through the collider! Raycasting must be used for such scenarios.

What we will create in our platformer is a "trap" around the bombs we throw. Whenever an enemy gets near the bomb's "force field", it will be immobilized so that it cannot escape before the bomb explodes. For that purpose, we'll need to use 3 components: a circle collider, an area clock, and a sprite renderer to visually display the force field.

First, place the `Prefabs/Props/bomb` prefab anywhere in your scene view. Then, add an empty GameObject child to it, called `Trap`, which will contain our trap components.



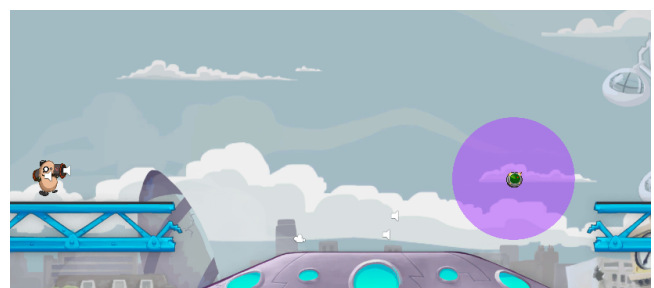
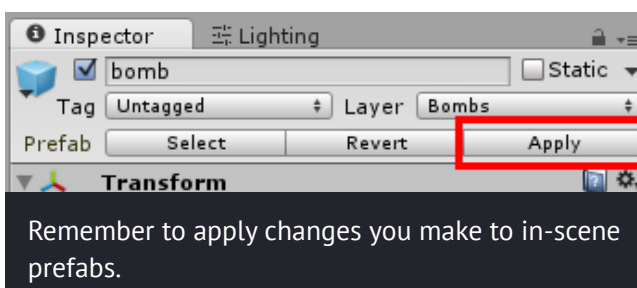
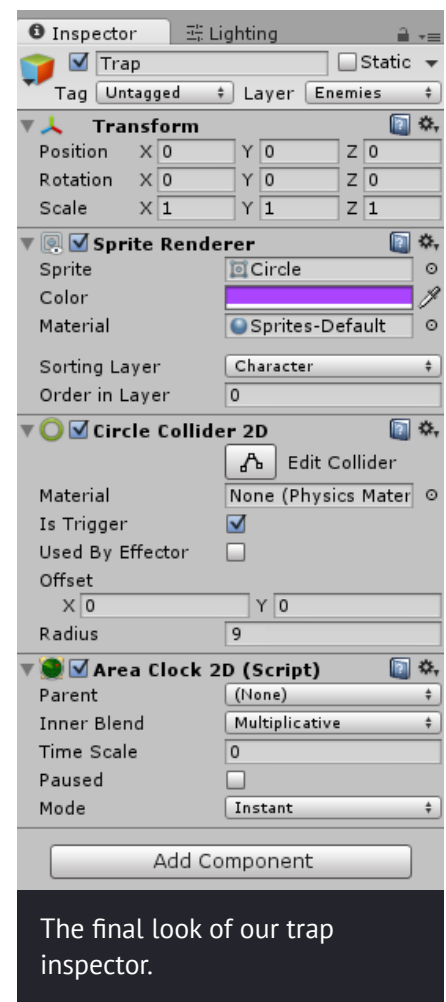
Let's then add our 3 components.

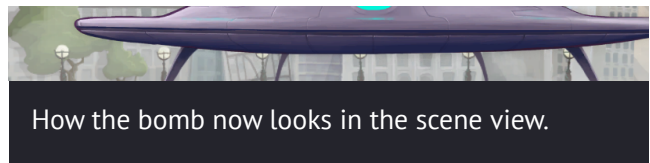
First, the renderer from **Rendering > Sprite Renderer**. Choose the **Sprites/_FX/Circle** sprite, and give it a translucent color that you like. Set its sorting layer to **Character** so that it renders above the background.

Second, the collider from **Physics 2D > Circle Collider 2D**. Toggle the **Is Trigger** checkbox and give it a radius of 9 so that it matches the size of the rendered circle.

Third, the area clock from **Time > Area Clock 2D**. You can either set its time scale to 0 or toggle the "Paused" checkbox – both will have the same effect. For now, leave the mode at **Instant**.

Select the bomb instance in the scene again and click **Apply** from the prefab toolbox on top of the inspector to save your changes.



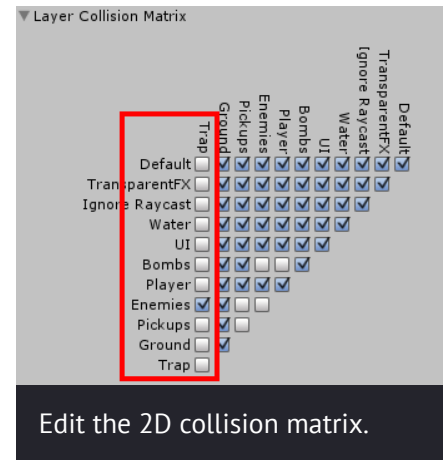
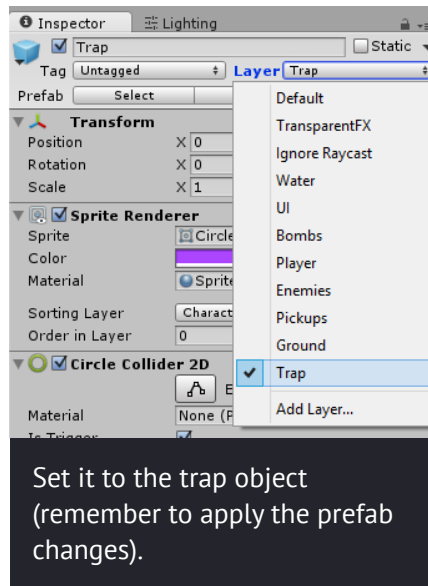
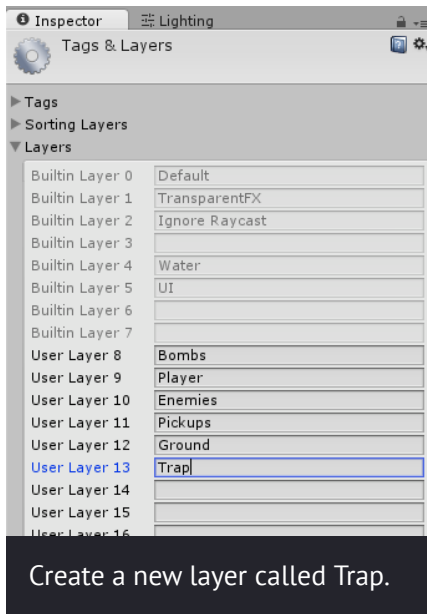


Try playing your game now. You'll notice the trap doesn't work! That's because we haven't set up our collisions properly.

Area clocks use Unity's built-in collisions to work. That means:

1. The area clock needs a trigger collider for `OnTriggerEnter` and `OnTriggerExit`
2. The object that enters the area clock must have a rigidbody for the collisions to be registered;
3. The layer of the area clock is used to select what objects it affects.

We're good for conditions 1 and 2, but not for 3. First, create a new layer called "Trap" and apply it to our trap object. Then, open up `Edit > Project Settings > Physics 2D` and configure the collision matrix so the Trap layer only collides with the Enemies layer. These steps are illustrated below:



Make sure you're not editing the 3D Physics settings! They're located in a similar but different menu: `Edit > Project Settings > Physics`.

Try playing the game now. The traps now sort of work – they stop the enemies, that is – but they stay stuck at the very edge of the trap, which doesn't look very realistic. Let's fix that.



Tired of waiting to get bombs? Change the Bomb Count value under the `Lay Bombs` component of the `hero` object in the scene. It's not cheating if we're just testing the game... right?

This unwanted behaviour happens because we left our area clock's mode at Instant, meaning that as soon as a collision is registered between a clock and the enemy, the clock's time scale, zero, is applied. This is rarely good for real-world scenarios, as it looks rough and unrealistic. Let's change our clock's mode to **Point to Edge** to change that.

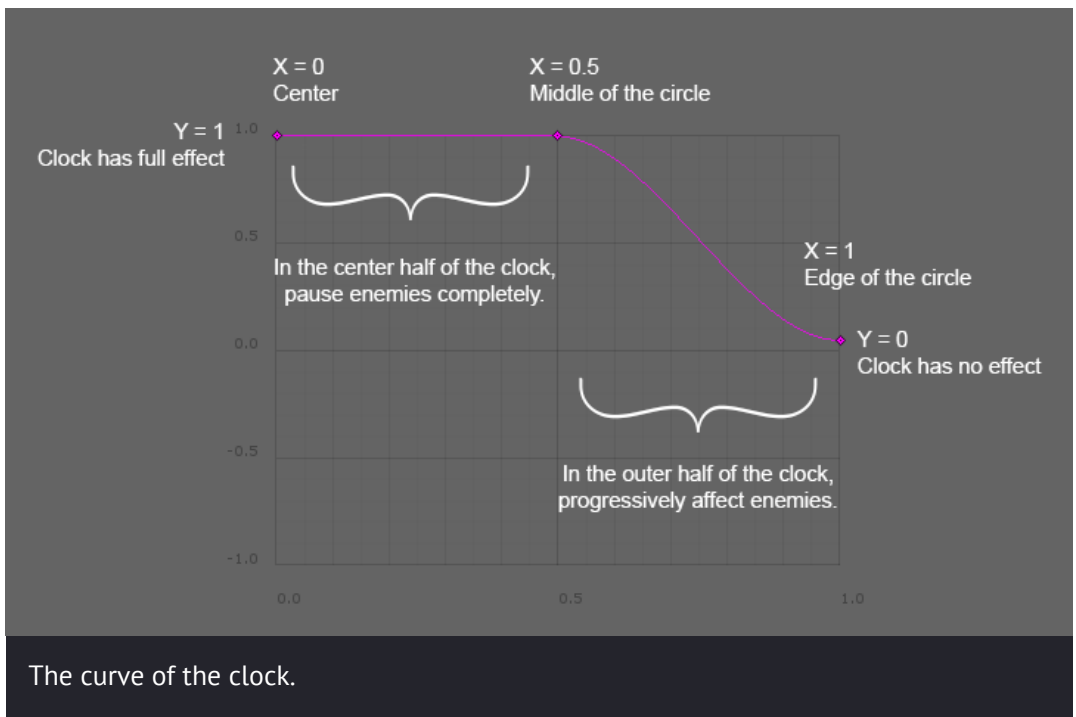
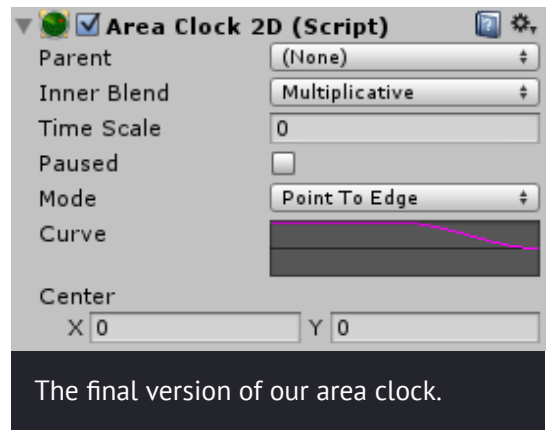
This brings up a curve in the inspector that allows us to specify how the time scale will progress along the area clock.

The **Y axis** is a multiplier of the effect of the clock. For example, $Y = 1$ means our enemies will be completely stopped, while $Y = 0$ means that they'd move at normal speed. For clocks that have a non-zero time scale, you can even set $Y < 0$ to reverse the time scale.

The **X axis** is an indication of progress within the clock. For the Point to Edge mode, $X = 0$ is the center of the clock, and $X = 1$ is the edge.

The **Center** parameter allows you to offset what point in the clock is considered to be the center for the curve at $X = 0$. Since we have a circle clock, it makes sense to leave it at $(0, 0)$, the center of the circle. If you change that parameter, you should see the area clock gizmo move in the scene view as a visual helper.

We want to make the enemies that enter our trap progressively affected by its pause. In other words, we want our curve at $X = 0$ to have full effect ($Y = 1$), and slowly dampen this effect from the middle of the clock ($X = 0.5$) to its edge ($X = 1$) until it reaches $Y = 0$. The picture below explains the curve we apply:





We could have obtained the same effect by using the `Distance from Entry` mode, with slightly different settings. See the [documentation](#) about area clock modes and curves to learn more.

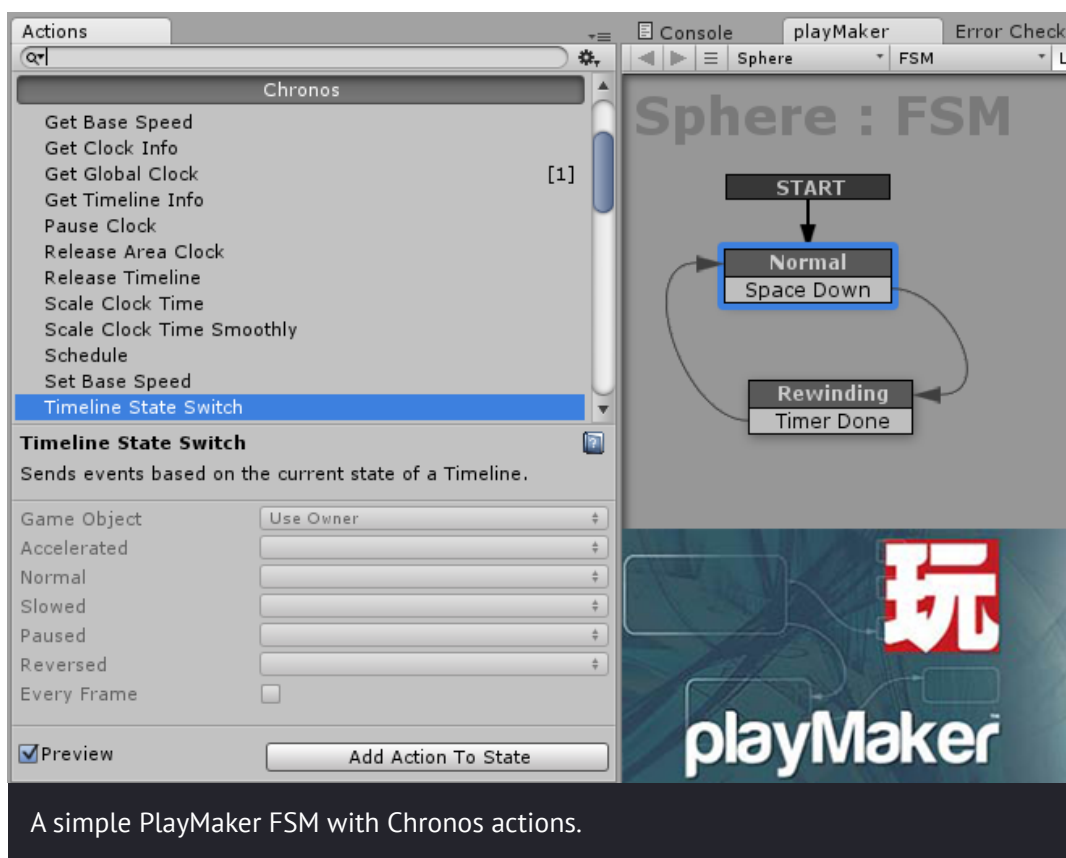
Now, apply your prefab changes and try playing your game. Enemies are properly affected by the trap – our bombs just became much more deadly!



PlayMaker

Chronos integrates with [PlayMaker](#), the popular visual scripting plugin for Unity. The action scripts and example scene are under the `Chronos/PlayMaker/` folder. If you don't need them, simply uncheck that folder when importing the package. The actions will be categorized under `Chronos` for general methods, and `Physics [2d] (Chronos)` for the [equivalents](#) of rigidbody methods.

There is no specific tutorial or documentation for PlayMaker actions as they follow the same naming conventions as regular Chronos methods. The help icon on each action is linked to the documentation page for each method.





Conclusion

You are now done with this introductory Chronos tutorial. Congratulations! You turned a simple platformer game into a time-controlled masterpiece (*kind of...*). Note that we won't cover how to add the time warp sound and icons shown in the end result, nor how to change controls, because these features are not Chronos specific.

Here are some suggestions to keep going:

- Tinker with the values of the tutorial to make the game your own
- To practice using timelines, try making the player's rockets affected by your time effects
- To practice using occurrences, try making some more events in the game rewindable: e.g. the player taking damage, the enemies flipping when they hit a collider, etc.
- Explore the [Documentation](#) to get a feel of all the possibilities Chronos has to offer.

We have only scratched the surface of what Chronos can help you create. Here at Ludiq, we look forward to seeing the games you will come up with. Above all, we hope that you will dare to tinker and explore Chronos and push it to its limits.

As a famous mind at Aperture Science once said: *now you're thinking with time control*. Or at least something like that. We're paraphrasing.



Did you spot any error in the tutorial?
If so, please report it in the [forum](#)!