DOCUMENTATION

Timekeeper

A global singleton tasked with keeping track of global clocks in the scene. One and only one Timekeeper is required per scene.

> The Timekeeper class follows the singleton design pattern. That's a complicated way of saying it's an object that can and must exist only once per scene. This object, called the instance, can be accessed from anywhere via `Timekeeper.instance`.

PROPERTIES

```
float debug { get; set; }
```

Determines whether Chronos should display debug messages and gizmos in the editor.

METHODS

```
bool HasClock(string key)
```

Indicates whether the timekeeper has a global clock with the specified key.

```
GlobalClock Clock(string key)
```

Returns the global clock with the specified key.

```
GlobalClock AddClock(string key)
```

Adds a global clock with the specified key and returns it.

```
GlobalClock RemoveClock(string key)
```

Removes the global clock with the specified key.

EXAMPLES

Speed up the "Monsters" global clock if it exists:

```
if (Timekeeper.instance.HasClock("Monsters"))
{
    Timekeeper.instance.Clock("Monsters").localTimeScale = 2;
}
```

Create a new clock for enemies and apply it to all Enemy GameObjects:

```
Timekeeper.instance.AddClock("Enemies");
Enemy[] enemies = GameObject.FindObjectsOfType<Enemy>();

foreach (Enemy enemy in enemies)
{
    // This assumes every enemy already has a Timeline component
    Timeline timeline = enemy.GetComponent<Timeline>();
    timeline.mode = TimelineMode.Global;
    timeline.globalClockKey = "Enemies";
}
```

Clock

An abstract base component that provides common timing functionality to all types of clocks. This means all the properties and methods of `Clock` are available on `GlobalClock`, `LocalClock` and `AreaClock`.

> ✓ You will almost never use the measurements (`time`, `deltaTime`, etc.) provided by clocks directly. You should instead use the same measurements available on the `Timeline` class, since these combine those of all relevant clocks for each GameObject.

PROPERTIES

float localTimeScale { get; set; } = 1

The scale at which the time is passing for the clock. This can be used for slow motion, acceleration, pause or even rewind effects.

float timeScale { get; }

The computed time scale of the clock. This value takes into consideration all of the clock's parameters (`parent`, `paused`, etc.) and multiplies their effect accordingly.

> ❗ Chronos is never affected by `Time.timeScale`. You should refrain from using that property, as it may even cause unexpected behaviour with physics. To control time globally, create a root `GlobalClock` instead and make all of your other clocks its descendants.

float time { get; }

The time in seconds since the creation of the clock, affected by the time scale. Returns the same value if called multiple times in a single frame.

> ⚠ Unlike `Time.time`, this value will not return `Time.fixedTime` when called inside MonoBehaviour's FixedUpdate.

float unscaledTime { get; }

The time in seconds since the creation of the clock regardless of the time scale. Returns the same value if called multiple times in a single frame.

float deltaTime { get; }

The time in seconds it took to complete the last frame, multiplied by the time scale. Returns the same value if called multiple times in a single frame.

> ⚠ Unlike `Time.deltaTime`, this value will not return `Time.fixedDeltaTime` when called inside MonoBehaviour's FixedUpdate.

float fixedDeltaTime { get; }

The interval in seconds at which physics and other fixed frame rate updates, multiplied by the time scale.

float startTime { get; }

The unscaled time in seconds between the start of the game and the creation of the clock.

bool paused { get; set; }

Determines whether the clock is paused. This toggle is especially useful if you want to pause a clock without having to worry about storing its previous time scale to restore it afterwards.

string parentKey { get; set; }

The key of the parent global clock, or `null` for none.

GlobalClock parent { get; set; }

The parent global clock. The parent clock will multiply its time scale with all of its children, allowing for cascading time effects.

ClockBlend parentBlend { get; set; } = Multiplicative

Determines how the clock combines its time scale with that of its parent.

Value	Description
`Multiplicative`	The clock's time scale is multiplied with that of its parent.
`Additive`	The clock's time scale is added to that of its parent.

✓ In most cases, multiplicative blend will yield the expected results. However, additive blend becomes extremely useful when you have a parent clock with a time scale of 0.

TimeState state { get; }

Indicates the state of the clock.

Value	Description
`Accelerated`	Time is accelerated (time scale > 1).
`Normal`	Time is in real-time (time scale = 1).
`Slowed`	Time is slowed (0 < time scale < 1).
`Paused`	Time is paused (time scale = 0).
`Reversed`	Time is reversed (time scale < 0).

METHODS

void LerpTimeScale(float timeScale, float duration, bool steady = false)

Changes the local time scale smoothly over the given duration in seconds.

> This method is not affected by any time scale.

> If you enable the `steady` parameter, `duration` will apply to a time scale variation of 1. For example, if you call `LerpTimeScale(3, 2, true)`, and the current time scale is 1, the duration of the lerp will be `(3 - 1) * 2 = 4` seconds instead of 2.

EXAMPLES

Toggle a pause on the "World" clock when pressing P:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.P))
        {
            GlobalClock worldClock = Timekeeper.instance.Clock("World");

            worldClock.paused = !worldClock.paused;
        }
    }
}
```

Same example, this time with smoothing:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    bool paused = false;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.P))
        {
            GlobalClock worldClock = Timekeeper.instance.Clock("World");

            if (!paused)
            {
                worldClock.LerpTimeScale(0, 1); // Change time scale to 0 over 1
                paused = true;
            }
            else
            {
```

GlobalClock

A `Clock` that affects all `Timeline` and other global clocks configured as its children.

PROPERTIES

`float key { get; }`

The unique key of the global clock.

LocalClock

A `Clock` that only affects a `Timeline` attached to the same GameObject.

METHODS

`void CacheComponents()`

The components used by the local clock are cached for performance optimization. If you add or remove the `Timeline` on the GameObject, you need to call this method to update the local clock accordingly.

AreaClock

A `Clock` that affects every `Timeline` within its collider by multiplying its time scale with that of their observed clock.

> ✓ Area clocks can be moved, scaled and rotated at runtime. They can even stack and combine their effects!

PROPERTIES

ClockBlend innerBlend { get; set; } = Multiplicative

Determines how the clock combines its time scale with that of the timelines within.

Value	Description
`Multiplicative`	The area clock's time scale is multiplied with that of the timelines.
`Additive`	The area clock's time scale is added to that of the timelines.

> In most cases, multiplicative blend will yield the expected results. However, additive blend becomes extremely useful to affect your timelines when they have a time scale of 0.

AreaClockMode mode { get; set; }

Determines how objects should behave when progressing within the area clock.

Value	Description
`Instant`	Objects that enter the clock are instantly affected by its full time scale, without any smoothing.
`PointToEdge`	Objects that enter the clock are progressively affected by its time scale, depending on a A / B ratio where: ▪ A is the distance between `center` and the object ▪ B is the distance between `center` and the collider's edge in the object's direction
	Objects that enter the clock are progressively affected by its time scale, depending on a A / B ratio where:

`DistanceFromEntry`	▪ A is the distance between the object's entry point and its current position ▪ B is the value of `padding`

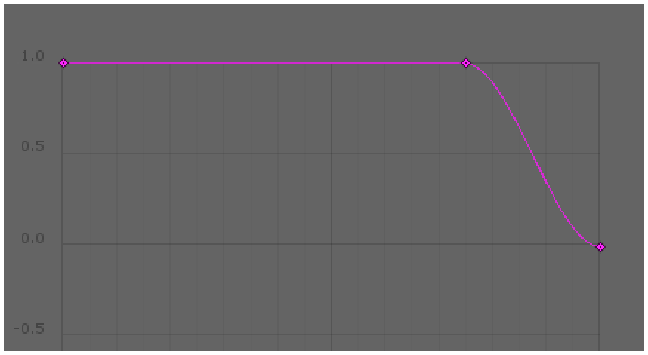
See the Notes section below for diagrams and more explanations for these modes.

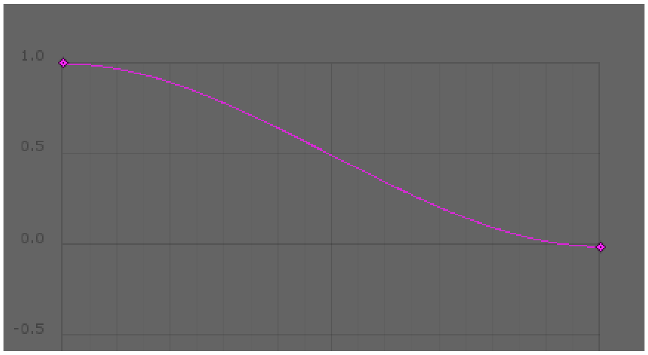
AnimationCurve curve { get; set; }

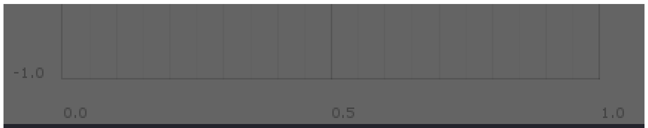
The curve of the area clock. This value is only used for the `PointToEdge` and `DistanceFromEntry` modes.

	Mode	
Axis	`PointToEdge`	`DistanceFromEntry`
Y	Indicates a multiplier of the clock's time scale, from 1 to -1.	
X	0 is at the `center`; 1 is at the collider's edge.	1 is at entry; 0 is at a distance of `padding` or more from entry.

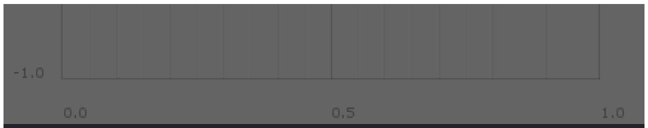
A good use for this curve is to dampen an area clock's effect to make it seem more natural. Think of the curve's left being the centermost part of the clock, and right being the edge. Common damping curves for both modes are illustrated below.







A common damping curve for the PointToEdge mode.



A common damping curve for the DistanceFromEntry mode.

Vector center { get; set; }

The center of the area clock. This value is only used for the `PointToEdge` mode.

float padding { get; set; }

The padding of the area clock. This value is only used for the `DistanceFromEntry` mode.

METHODS

void Release(Timeline timeline)

Releases the specified timeline from the clock's effects.

void ReleaseAll()

Releases all timelines within the clock.

> If the timeline enters the clock after having been released, it will be captured again. To configure which area clocks capture which timelines, use Unity's collision matrix.

void CacheComponents()

The components used by the area clock are cached for performance optimization. If you add or remove the `Collider` on the GameObject, you need to call this method to update the area clock accordingly.

NOTES

This section contains diagrams that help understand how the `PointToEdge` and `DistanceFromEntry` modes function. For simplicity, 2D area clocks are represented, however the same concepts and calculations apply to 3D area clocks.

POINT TO EDGE

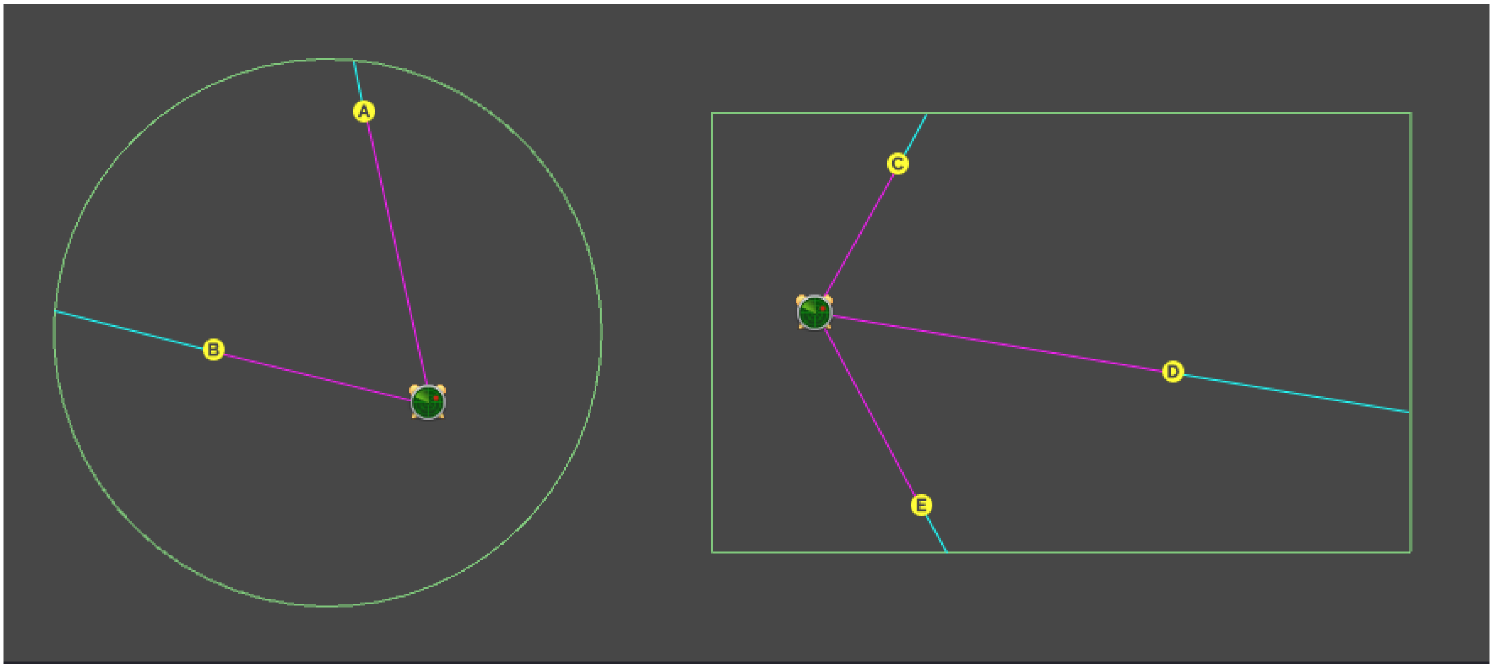


Diagram for objects within a `PointToEdge` area clock.

- Area clock icon: Clock's `center`
- Green edge: Clock's collider
- Yellow dot: Object's position
- Magenta line: Distance between center and object
- Cyan line: Distance between object and edge

In the point to edge mode, the X value of the `curve` property is calculated by the distance of the object from `center` to the edge of the area clock.

In the above diagram, this represents the following ratio:

```
x = Magenta line / ( Cyan line + Magenta line )
```

For example, A and E would have an x value of about 0.8, while B and D would have an x value of about 0.6. In the unlikely event that an object was placed at the exact center of the clock, its x value would be 0.

DISTANCE FROM ENTRY

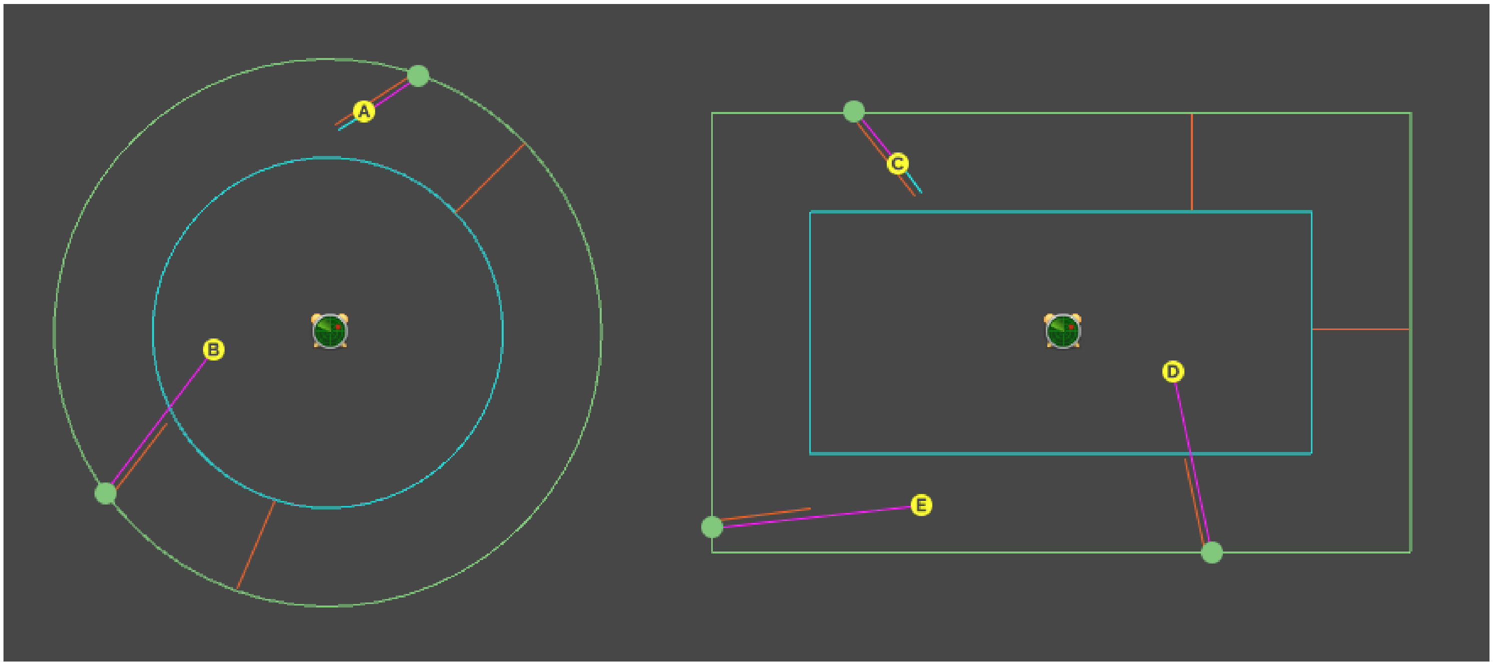


Diagram for objects within a `DistanceFromEntry` area clock.

- Area clock icon: Clock's local position
- Green edge: Clock's collider
- Green dot: Object's entry point
- Yellow dot: Object's position
- Orange line: Length of `padding`
- Cyan edge: Collider inset by padding (for reference only)
- Magenta line: Distance between entry and object
- Cyan line: Distance remaining before end of padding

In the distance from entry mode, the X value of the `curve` property is calculated by the distance of the object from its entry point relative to the value of `padding`.

In the above diagram, this represents the following ratio:

```
x = Cyan line / Orange line
```

For example, A and C would have an x value of about 0.25, while B, D and E would have an x value of 0, because they went over the maximum distance.

> ⚠ The inset cyan collider has no real impact on the calculation. It is merely a helper that lets you visualize how big the value of `padding` looks while in the editor. Indeed, unless objects enter perfectly ortogonally to the collider's edge, their max distance will not be at the edge of the inset cyan collider.

✓ You can display the cyan and magenta lines for both modes in scene view by enabling `Timekeeper.debug`.

Timeline

A component that combines timing measurements from an observed `LocalClock` or `GlobalClock` and any `AreaClock` within which it is.

ℹ This component should be attached to any GameObject that should be affected by Chronos.

PROPERTIES

TimelineMode mode { get; set; }

Determines what type of clock the timeline observes.

Value	Description
`Local`	The timeline observes a `LocalClock` attached to the same GameObject.
`Global`	The timeline observes a `GlobalClock` referenced by `globalClockKey`.

string globalClockKey { get; set; }

The key of the `GlobalClock` that is observed by the timeline. This value is only used for the `Global` mode.

Clock clock { get; }

The clock observed by the timeline.

float timeScale { get; }

The time scale of the timeline, computed from all observed clocks. For more information, see `Clock.timeScale`.

float deltaTime { get; }

The delta time of the timeline, computed from all observed clocks. For more information, see `Clock.deltaTime`.

float smoothDeltaTime { get; }

A smoothed out delta time. Use this value if you need to avoid spikes and fluctuations in delta times. The amount of frames over which this value is smoothed can be adjusted via `smoothingDeltas`.

static int smoothingDeltas { get; set; } = 5

The amount of frames over which `smoothDeltaTime` is smoothed.

float fixedDeltaTime { get; }

The fixed delta time of the timeline, computed from all observed clocks. For more information, see `Clock.fixedDeltaTime`.

float time { get; }

The time in seconds since the creation of this timeline, computed from all observed clocks. For more information, see `Clock.time`.

float unscaledTime { get; }

The unscaled time in seconds since the creation of this timeline. For more information, see

`Clock.unscaledTime`.

TimeState state { get; }

Indicates the state of the timeline.

Value	Description
`Accelerated`	Time is accelerated (time scale > 1).
`Normal`	Time is in real-time (time scale = 1).
`Slowed`	Time is slowed (0 < time scale < 1).
`Paused`	Time is paused (time scale = 0).
`Reversed`	Time is reversed (time scale < 0).

SPEEDS

Chronos manipulates the built-in Unity components at runtime to adjust their speeds. This allows an effortless setup in almost all cases. However, it also means that if you edit their speeds directly, Chronos will overwrite them or behave unexpectedly. To remedy this situation, you should use the properties below instead.

> For more information about the script changes needed to migrate to Chronos, see the Migration page.

float animatorSpeed { get; set; }

The speed that is applied to animations before time effects. Use this property instead of `Animator.speed`, which will be overwritten by the timeline at runtime.

float animationSpeed { get; set; }

The speed that is applied to animations before time effects. Use this property instead of `AnimationState.speed`, which will be overwritten by the timeline at runtime.

float particleSpeed { get; set; }

The speed that is applied to particles before time effects. Use this property instead of `ParticleSystem.playbackSpeed`, which will be overwritten by the timeline at runtime.

> ❗ At extremely low speeds or time scales (< 0.25), particle systems will appear to stutter. This is due to a bug in Unity's particle simulation method. A bug report has been submitted here: ParticleSystem.Simulate truncates first parameter to 2 decimals.

float audioSpeed { get; set; }

The speed that is applied to audio before time effects. Use this property instead of `AudioSource.pitch`, which will be overwritten by the timeline at runtime.

float navigationSpeed { get; set; }

The speed that is applied to navigation before time effects. Use this property instead of `NavMeshAgent.speed`, which will be overwritten by the timeline at runtime.

float navigationAngularSpeed { get; set; }

The angular speed that is applied to navigation before time effects. Use this property instead of `NavMeshAgent.angularSpeed`, which will be overwritten by the timeline at runtime.

WindZoneSpeeds windZoneSpeeds { get; set; }

The speeds that are applied to the wind zone before time effects. Use this property instead of `WindZone.wind*`, which will be overwritten by the timeline at runtime.

METHODS

void ReleaseFrom(AreaClock areaClock)

Releases the timeline from the specified area clock's effects.

void ReleaseFromAll()

Releases the timeline from the effects of all the area clocks within which it is.

Coroutine WaitForSeconds(float seconds)

Suspends the coroutine execution for the given amount of seconds. This method should only be used with a yield statement in coroutines.

> ❗ There is currently no built-in support for rewindable coroutines due to limitations in the way .NET enumerators work. This feature is being considered for a future release if a workaround can be found. For now, if time is going backward, `WaitForSeconds` will simply never finish.

void CacheComponents()

The components used by the timeline are cached for performance optimization. If you add or remove the `Animator`, `Animation`, `ParticleSystem`, `NavMeshAgent`, `AudioSource` or `WindZone` on the GameObject, you need to call this method to update the timeline accordingly.

OCCURRENCES

> ℹ To keep this documentation organized, the timeline methods to trigger occurrences can be found in the occurrence triggers section below.

EVENTS

void OnStartPause()

Sent to every behaviour on the GameObject when the timeline starts a pause.

void OnStopPause()

Sent to every behaviour on the GameObject when the timeline stops a pause.

void OnStartRewind()

Sent to every behaviour on the GameObject when the timeline starts a rewind.

void OnStopRewind()

Sent to every behaviour on the GameObject when the timeline stops a rewind.

void OnStartSlowDown()

Sent to every behaviour on the GameObject when the timeline starts a slow-down.

void OnStopSlowDown()

Sent to every behaviour on the GameObject when the timeline stops a slow-down.

void OnStartFastForward()

Sent to every behaviour on the GameObject when the timeline starts a fast-forward.

void OnStopFastForward()

Sent to every behaviour on the GameObject when the timeline stops a fast-forward.

EXAMPLES

Make the GameObject rotate in a framerate-independant manner by the scale of all affected clocks:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Update()
    {
        Timeline time = GetComponent<Timeline>();

        transform.Rotate(time.deltaTime * Vector3.one * 20);
    }
}
```

Make the GameObject cyan when paused:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    Color oldColor;

    void OnStartPause()
    {
        Renderer renderer = GetComponent<Renderer>();
        oldColor = renderer.material.color;
        renderer.material.color = Color.cyan;
    }

    void OnStopPause()
    {
        Renderer renderer = GetComponent<Renderer>();
        renderer.material.color = oldColor;
    }
}
```

Create a timeline component procedurally if it doesn't exist:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Awake()
    {
        Timeline time = GetComponent<Timeline>();

        if (time == null)
        {
            time = gameObject.AddComponent<Timeline>();
            time.mode = TimelineMode.Global;
            time.globalClockKey = "Monsters";
        }
    }
}
```

> It is usually much simpler to add a timeline component directly from the editor — this is purely in case you need to create your GameObjects procedurally.

Changing the GameObject's color randomly every 5 seconds. This delay will take in consideration pauses, fast-forwards and slow-downs.

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    Timeline time;

    void Awake()
    {
        time = GetComponent<Timeline>();

        StartCoroutine(ChangeColor());
    }

    IEnumerator ChangeColor()
    {
        while (true)
        {
            // Use Timeline.waitForSeconds()
            // instead of new WaitForSeconds()
```

⚠ The previous example is not rewindable.

Create a base behaviour that will let you access the timeline component easily:

```
using UnityEngine;
using Chronos;

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

// ... In other scripts, simply inherit from
// BaseBehaviour instead of MonoBehaviour

class MyBehaviour : BaseBehaviour
{
    void Update()
```

✓ Using a base behaviour is a good Unity design pattern, and it is certainly not limited to Chronos! If you have any methods or properties that you use very often, feel free to add them to `BaseBehaviour`. They'll be accessible from any script that extends it.

Occurrence

An event anchored at a specified moment in time composed of two actions: one when time goes forward, and another when time goes backward. The latter is most often used to revert the former.

PROPERTIES

```
float time { get; }
```

The time in seconds on the parent timeline at which the occurrence will happen.

```
bool repeatable { get; }
```

Indicates whether this occurrence can happen more than once; that is, whether it will stay on the timeline once it has been rewound.

METHODS

```
abstract void Forward()
```

The action that is executed when time goes forward.

```
abstract void Backward()
```

The action that is executed when time goes backward.

TRIGGERS

> The following methods are all called from a `Timeline` instance.
> They are placed in this section for the sake of organization only.

```
Occurrence Schedule(float time, bool repeatable, Occurrence occurrence)
Occurrence Schedule(float time, bool repeatable, ForwardAction forward, BackwardAction
Occurrence Schedule(float time, ForwardOnlyAction forward)
```

Schedules an occurrence at a specified absolute time in seconds on the timeline.

```
Occurrence Do(bool repeatable, Occurrence occurrence)
Occurrence Do(bool repeatable, ForwardAction forward, BackwardAction backward)
```

Executes an occurrence now and places it on the schedule for rewinding.

Occurrence Plan(float delay, bool repeatable, Occurrence occurrence)
Occurrence Plan(float delay, bool repeatable, ForwardAction forward, BackwardAction bac
Occurrence Plan(float delay, ForwardOnlyAction forward)

Plans an occurrence to be executed in the specified delay in seconds.

Occurrence Memory(float delay, bool repeatable, Occurrence occurrence)
Occurrence Memory(float delay, bool repeatable, ForwardAction forward, BackwardAction
Occurrence Memory(float delay, ForwardOnlyAction forward)

Creates a "memory" of an occurrence at a specified "past-delay" in seconds. This means that the occurrence will only be executed if time is rewound, and that it will be executed backward first.

> If `repeatable` is set to `false`, the occurrence will be cancelled after it has been rewound. This is useful for code that occurs as a result of object interaction and would therefore reoccur by itself after a rewind.

> All the trigger methods return the created occurrence as a result, in case you need to cancel or reschedule it via the methods below.

void Cancel(Occurrence occurrence)

Removes the specified occurrence from the timeline.

bool TryCancel(Occurrence occurrence)

Removes the specified occurrence from the timeline and returns `true` if it is found. Otherwise, returns `false`.

void Reschedule(Occurrence occurrence, float time)

Changes the absolute time in seconds of the specified occurrence on the timeline.

void Postpone(Occurrence occurrence, float delay)

Moves the specified occurrence forward on the timeline by the specified delay in seconds.

void Prepone(Occurrence occurrence, float delay)

Moves the specified occurrence backward on the timeline by the specified delay in seconds.

EXAMPLES

Changing the color of the GameObject to blue 5 seconds after Space is pressed. This delay will take into consideration pauses, fast-forwards and slow-downs. However, this occurrence is not (yet) rewindable: the object will not go back to its original color on rewind.

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            GetComponent<Timeline>().Plan(5, ChangeColor);

            // Notice the absence of () after the method name.
            // This is because we are refering to the method itself,
            // not calling it right now.
        }
    }

    void ChangeColor()
    {
        GetComponent<Renderer>.material.color = Color.blue;
```

Same example as before, but this time, we pass a color parameter to our method:

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            GetComponent<Timeline>().Plan(5, delegate { ChanceColor(Color.red);

            // Here, we create a delegate (an inline method) to
            // be called in 5 seconds. In turn, our delegate calls
            // the ChangeColor with the color red as a parameter.
        }
    }

    void ChangeColor(Color newColor)
    {
        GetComponent<Renderer>.material.color = newColor;
```

If you are unfamiliar with delegates, it is recommended that you take a look at the MSDN tutorial on delegates. Note, however, that you won't have to create your own delegate types to use Chronos — you will only need a basic understanding of what they are and how they can be created.

Now, let's make our color change a rewindable occurrence. When time is rewound up to the moment the GameObject became blue, it should automatically revert back to its original color.

```
using UnityEngine;
using Chronos;

class MyBehaviour : MonoBehaviour
{
    void Start()
    {
        GetComponent<Timeline>().Plan
        (
            5, // In 5 seconds...

            true, // ... create a repeatable event...

            delegate // ... that sets the color to blue and saves the previous o
            {
                Renderer renderer = GetComponent<Renderer>();
                Color previousColor = renderer.material.color;
                renderer.material.color = Color.blue;
                return previousColor; // This will be passed to "transfer" below
            },
```

✓ Try experimenting with the `repeatable` parameter from that example to get an understanding of what it does. For example, try setting it to `false`, then rewinding time after the object turned blue until it goes back to its original color, then letting time go forward normally. You'll realize it doesn't turn blue again! That's because the occurrence was removed from the timeline after rewinding.

That previous example works, but it's a bit tedious to set up. Imagine that we often wanted to have rewindable color-change occurrences like this one — it would be quite annoying to type that code every time! Fortunately, we don't have to.

In the following example, we'll create our own `Occurrence` class and transform our previously lengthy code into a reusable one-liner.

```
using UnityEngine;
using Chronos;

// Inherit Occurrence and implement Forward() and Backward()
public class ChangeColorOccurrence : Occurrence
{
    Material material;
    Color newColor;
    Color previousColor;

    public ChangeColorOccurrence(Material material, Color newColor)
    {
        this.material = material;
        this.newColor = newColor;
    }

    public override void Forward()
    {
        previousColor = material.color;
        material.color = newColor;
```

You now have the tools to make any kind of custom code work with Chronos; whether time flows normally, slower, faster or even backwards!

> There is one last *gotcha*. When making rewindable occurrences, it is crucial to remember that if your code occurs from the interaction of two game objects (e.g. collisions), it should — in almost all cases — **not** be set to `repeatable`.
>
> For example, say you change the color of two objects to red when they collide (in `OnCollisionEnter`). If that occurrence is repeatable and you rewind, then let time run normally, not only will the objects change to red from the existing occurrence, but they'll collide *again*, creating a new occurrence. This can quickly lead to unexpected results, so be careful!

Recorder

An abstract base component that saves snapshots of any kind at regular intervals to enable rewinding.

The recorder class is just a shell to properly record snapshots and rewind by interpolating between them. By itself, it doesn't do anything; you can't add it to a GameObject. However, it can be extended (subclassed) to record any type of information in snapshots.

There are 3 Chronos components that are recorders: `TransformRecorder`, `AnimatorRecorder` and `PhysicsTimer`. These are provided out of the box because they are commonly used. However, you can create your own! See the examples section below for more information.

PROPERTIES

float recordingDuration { get; }

The maximum duration in seconds during which snapshots will be recorded. Higher values offer more rewind time but require more memory.

float recordingInterval { get; }

The interval in seconds at which snapshots will be recorded. Lower values offer more rewind precision but require more memory.

bool exhaustedRewind { get; }

Indicates whether the recorder has exhausted its rewind capacity.

METHODS

void SetRecording(float duration, float inverval)

Sets the recording duration and interval in seconds.

⚠ This will reset the saved snapshots.

void Reset()

Resets the saved snapshots.

int EstimateMemoryUsage()

Estimate the memory usage in bytes from the storage of snapshots for the current recording duration and interval.

EVENTS

void OnExhaustRewind()

Sent to all behaviours on the GameObject when the timer exhausts its rewind capacity.

EXAMPLES

Let's say we want to record the health and color of our player (presuming its color changes over time). We will create a `PlayerRecorder` script that inherits `Recorder`, and implement the 3 mandatory abstract methods:

- `CopySnapshot`: Records the current state of the object and returns a snapshot
- `ApplySnapshot`: Takes a snapshot and applies it to the object
- `LerpSnapshot`: Interpolates between two snapshots and returns the result

```
using UnityEngine;
using Chronos;

// Make your class inherit Recorder.
// The generic parameter points to the type of snapshot.
public class PlayerRecorder : Recorder<PlayerRecorder.Snapshot>
{
    // The struct that contains each of our snapshot's data.
    // In our case, a health value and a color.
    public struct Snapshot
    {
        public float health;
        public Color color;
    }

    // Record the current health and color in a snapshot
    protected override Snapshot CopySnapshot()
    {
        return new Snapshot()
        {
```

Finally, we need to create a custom editor for our recorder. Fortunately, it is very easy to do. Create a `PlayerRecorderEditor` script and place it in an `Editor` folder in your assets directory, with the following code:

```
using UnityEditor;
using Chronos;

// Use attributes to tell unity that this is an editor for PlayerRecorder
// and that it can edit multiple objects at the same time.
// Inherit RecorderEditor with the type parameter set to our custom recorder.
[CustomEditor(typeof(PlayerRecorder)), CanEditMultipleObjects]
public class PlayerRecorderEditor : RecorderEditor<PlayerRecorder>
{
    // That's it! No need to add any code here.
    // Chronos will take care of the proper inspector display.
}
```

You can then attach your `PlayerRecorder` component to your player GameObject and configure its recording settings. Its health and color should now be rewindable.

If you'd like another example of the recorder class, look at the script behind `TransformRecorder`. It's actually just as simple as the one we created!

TransformRecorder

A component that enables rewinding the transform (position, rotation and scale) of the GameObject via recorded snapshots.

> No scripting required! Just attach this component to a GameObject you'd like to rewind (along with a timeline, of course). If the object is a non-kinematic rigidbody, use a `PhysicsTimer` instead.

AnimatorRecorder

A component that enables rewinding the state of the Animator on the GameObject via recorded snapshots of its parameters.

> This recorder takes snapshots of each of the animator controller's user-defined parameters (e.g. "Speed", "Left", etc.), whether they are floats, integers or bool. This makes your animators easy to rewind without having to use occurrences. For more information about animation parameters, see the Unity manual.

> ⚠ Trigger parameters are not yet supported.

> ! Until the next version of Unity is released, Chronos cannot access enough parameters of mecanim states to properly handle rewinding. Therefore, this component may be buggy in its current state; it will be updated as soon as Unity 5.1 is released.

PhysicsTimer

A component that enables time effects on the non-kinematic rigidbody attached to the same GameObject.

Chronos manipulates rigidbodies at runtime to adjust their physical properties. This way, you don't have to worry about the complex calculations used to make time effects physically accurate. However, it also means that if you edit their properties directly, Chronos will overwrite them or behave unexpectedly. To remedy this situation, you should use the properties and methods below instead.

> For more information about the script changes needed to migrate to Chronos, see the Migration page.

PROPERTIES

bool isKinematic { get; set; }

Determines whether the rigidbody is kinematic before time effects. Use this property instead of `Rigidbody.isKinematic`, which will be overwritten by the physics timer at runtime.

bool useGravity { get; set; } // 3D

Determines whether the rigidbody uses gravity. Use this property instead of `Rigidbody.useGravity`, which will be overwritten by the physics timer at runtime.

float gravityScale { get; set; } // 2D

The gravity scale of the rigidbody. Use this property instead of `Rigidbody2D.gravityScale`, which will be overwritten by the physics timer at runtime.

float mass { get; set; }

The mass of the rigidbody before time effects. Use this property instead of `Rigidbody.mass`, which will be overwritten by the physics timer at runtime.

```
Vector3 velocity { get; set; } // 3D
Vector2 velocity { get; set; } // 2D
```

The velocity of the rigidbody before time effects. Use this property instead of `Rigidbody.velocity`, which will be overwritten by the physics timer at runtime.

```
Vector3 angularVelocity { get; set; } // 3D
float angularVelocity { get; set; } // 2D
```

The angular velocity of the rigidbody before time effects. Use this property instead of `Rigidbody.angularVelocity`, which will be overwritten by the physics timer at runtime.

```
float drag { get; set; }
```

The drag of the rigidbody before time effects. Use this property instead of `Rigidbody.drag`, which will be overwritten by the physics timer at runtime.

```
float angularDrag { get; set; }
```

The angular drag of the rigidbody before time effects. Use this property instead of `Rigidbody.angularDrag`, which will be overwritten by the physics timer at runtime.

METHODS

```
void AddForce(Vector3 force, ForceMode mode = ForceMode.Force) // 3D
void AddForce(Vector2 force, ForceMode2D mode = ForceMode2D.Force) // 2D
```

The equivalent of `Rigidbody.AddForce` adjusted for time effects.

```
void AddRelativeForce(Vector3 force, ForceMode mode = ForceMode.Force) // 3D
void AddRelativeForce(Vector2 force, ForceMode2D mode = ForceMode2D.Force) // 2D
```

The equivalent of `Rigidbody.AddRelativeForce` adjusted for time effects.

```
void AddForceAtPosition(Vector3 force, Vector3 position, ForceMode mode = ForceMode.F
void AddForceAtPosition(Vector2 force, Vector2 position, ForceMode2D mode = ForceMod
```

The equivalent of `Rigidbody.AddForceAtPosition` adjusted for time effects.

```
void AddExplosionForce(float explosionForce, Vector3 explosionPosition, float explosionF
```

The equivalent of `Rigidbody.AddExplosionForce` adjusted for time effects.

```
void AddTorque(Vector3 force, ForceMode mode = ForceMode.Force) // 3D
void AddTorque(Vector2 force, ForceMode2D mode = ForceMode2D.Force) // 2D
```

The equivalent of `Rigidbody.AddTorque` adjusted for time effects.

```
void AddRelativeTorque(Vector3 force, ForceMode mode = ForceMode.Force) // 3D
void AddRelativeTorque(Vector2 force, ForceMode2D mode = ForceMode2D.Force) // 2D
```

The equivalent of `Rigidbody.AddRelativeTorque` adjusted for time effects.

```
void CacheComponents()
```

The components used by the physics timer are cached for performance optimization. If you add or remove the `Timeline` or `Rigidbody` on the GameObject, you need to call this method to update the physics timer accordingly.

EXAMPLES

Changing the velocity of the rigidbody based on horizontal input:

```
GetComponent<PhysicsTimer3D>.velocity = new Vector3(Input.GetAxis("Horizontal")

// Do *not* use GetComponent<Rigidbody>.velocity!
```

Adding a relative impulse force to the rigidbody:

```
GetComponent<PhysicsTimer3D>.AddRelativeForce(Vector3.forward * 20, ForceMode.Im

// Do *not* use GetComponent<Rigidbody>.AddForce!
```

NOTES

SLOWING & ACCELERATING

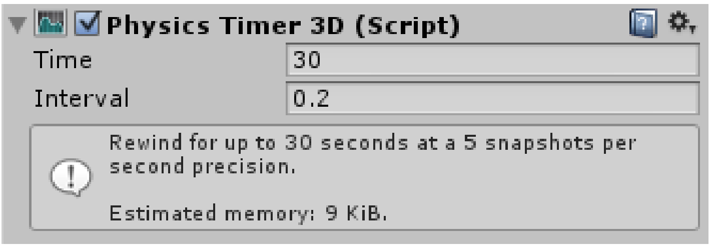
On physics timers, slowing or accelerating changes a lot of the properties of the rigidbody in order for physics to stay accurate. You must use the properties of the physics timer instead of those of the rigidbody if you need to adjust the velocities, drags or mass on the object at runtime, otherwise the physics simulation will behave incorrectly.

PAUSING

On physics timers, pausing works by setting the rigidbody to `isKinematic`. Note that if you use kinematic-only rigidbodies, there's no need to use a physics timer!

REWINDING

On physics timers, rewinding works by interpolating between snapshots of the rigidbody's state. Unlike rewinding normal GameObjects, it is not unlimited. Still, the operation is highly optimized for both memory and processing, so you can almost certainly provide enough rewind time and precision for the purpose of your game. The physics timer inspector will provide you with an estimate of how much memory (RAM) is used by the snapshots storage of each GameObject. If you select multiple GameObjects with physics timers, the estimate will sum all of them.



☑ Physics Timer 3D (Script)

Time: 30

Interval: 0.2

Rewind for up to 30 seconds at a 5 snapshots per second precision.

Estimated memory: 9 KiB.

The memory estimate is displayed live in the inspector window.

While rewinding, the rigidbody is set to kinematic and its transform is modified at each frame. Therefore, it will not behave like a physical object. Note that you cannot change the properties of the rigidbody while the physics timer is rewinding or paused.

Did you spot any error in the documentation?
If so, please report it in the forum!