Intro:

Hello and welcome to the setup guide for the physics prediction system.
This asset will allow you to predict physics interactions for Prefabs and GameObjects, it supports both 2D & 3D physics.

Initial Setup:

The initial setup for the asset is very minimal or basically non existent, just import into the project and run one of the examples (Projectile Shooter 2D & Projectile Shooter 3D) to see it in action.

Usage:

The system mainly consists of two scripts, the PredictionObject component, and the PredictionSystem static class.
The PredictionObject component is to be added to any GameObject that you want to influence predictions such as walls, roofs, floors, … etc, the component has a few restriction which are:

- No nested PredictionObjects, only one PredictionObject is to be applied within a GameObject tree, preferably the root.
- No nested physics (2D & 3D) on the same GameObject with the PredictionObject component, each PredictionObject must define either a 3D or 2D object.

Every PredictedObject's GameObject is cloned to an appropriate scene for individual simulation where every "non essential component" on the cloned object is destroyed immediately, there are a few gotchas that come with this cloning system, they will explained in the Gotchas section.

The second part of the system is the PredictionSystem class, you don't need to call any methods to configure this class, you can use it right away.
The PredictionSystem's class main purpose is to maintain the prediction simulation, you can predict the current physical world's next steps using the Simulate method:

```
void PredictionSystem.Simulate(iterations);
```

Where iterations define how many times to simulate the physical world, every iteration is a simulation of the physical world using the Fixed Delta Time (0.02 by default), so 50 iterations will give you a 1 second gaze into the future.

This process will simulate the physical world, but to retrieve any useful data from the simulation we will need to record it, the recording methods of the system are located within the Record nested class of the PredictionSystem:

```
PredictionSystem.Record... ;
```

You may record either a Prefab or an Object (PredictionObject) using the Add method, which will return a PredictionTimeline object:

```
PredictionTimeline PredictionSystem.Record.Prefabs.Add(prefab, action);
PredictionTimeline PredictionSystem.Record.Objects.Add(target);
```

The action parameter of the Prefab Add method is a delegate to be assigned as the action to perform to that prefab instance, such as shooting it by applying force to its rigidbody component. to ensure an accurate prediction; be sure to apply the same action to both your predicted projectiles and real projectiles, look into the ProjectileShooter3D's or ProjectileShooter2D's Launch methods for an example.
The returned timeline object will be populated with all the data (position & rotation) from the following simulations, where this data can be used to visualize trajectory, you will want to keep a reference of the timeline object.
To stop recording an object you will need to call the appropriate Remove method passing in the timeline returned from the Add method:

```
bool PredictionSystem.Record.Prefab.Remove(timeline);
bool PredictionSystem.Record.Objects.Remove(timeline);
```

Gotchas:

The first thing you need to keep in mind when using this system is the cloning system, as noted above, all PredictionObjects are cloned to a separate scene to be simulated accordingly, this may cause issues in the following way:

All components non essential to the simulation are destroyed immediately, by default only transforms, rigidbodies, colliders and components implementing the `IPredictionPersistantObject` interface are preserved on the clone object.
You may extend that destruction mechanism by either adding a type to the PersistentComponents hashset:

```
void PredictionSystem.Clone.PersistentComponents.Add(type);
```

For a more complex method to override the destruction evaluation you may also assign a method to evaluate on a per component basis using the EvaluateComponentPersistence property such as:

```
PredictionSystem.Clone.EvaluateComponentPersistence =
EvaluateComponentPersistence;

bool EvaluateComponentPersistence(GameObject gameObject, Component
component)
{
      if (component.tag == "Persistent") return true;

            return false;
}
```

The purpose of preserving components for simulation is to ensure accuracy for any component necessary for an accurate physical prediction, just as you need a rigidbody to simulate physics; you may have a BouncePad component that will exert force on physical objects that collide with it, and preserving that BouncePad component in the clone will ensure an accurate simulation. You may also want to define different behaviour for the BouncePad depending on whether if it's a clone or the original object, for example, if you have a BouncePad that reports statistics on how many times it was used, you wouldn't want the clone BouncePad to report all the predictions as actual usages, to differentiate between the original and clone you many assign an IsClone bool as a flag using the PredictionSystem.Clone.Flag property such as is used within the PredictionObject component

```
public bool IsClone { get; protected set; }

void Awake()
{
    IsClone = PredictionSystem.Clone.Flag;
}
```

From which you can alter the BouncePad's behaviour depending on whether it's a clone or not. Also be sure to assign the IsClone flag within the Awake method as that method will be called as soon as the Object is instantiated where the PredictionSystem.Clone.Flag will be set as true.

The second gotcha is also related to the cloning procedure as well, as the source GameObject is cloned using Object.Instantiate, Unity callbacks specifically Awake, OnEnable & OnDestroy will be invoked on every component, even the components that will be destroyed from the cloned object, this might cause problems as well since having those callbacks executed on a clone might cause issues, the way to handle this gotcha is also by using the PredictionSystem.Clone.Flag to control the code execution, for an example you may look in the TrajectoryPredictionDrawer component's source:

```
public bool IsClone { get; protected set; }

void Awake()
{
    IsClone = PredictionSystem.Clone.Flag;
    if (IsClone) return;

    //Rest of Awake Code ...
}

void OnDestroy()
{
    if (IsClone) return;

    //Rest of OnDestory Code ...
}
```

The third and final gotcha is very simple, performance! As you might've expected, simulating physics multiple times every frame is going to be a bit expensive! The performance loss so far actually seems acceptable, in the 3D Projectile example I'm simulating the physics 50 times every 0.33's of a second (30 times a second) and the performance loss is decent (about 1ms) which drops my PC's FPS from ~1000 to ~820, and it ran on my OnePlus3 without any noticeable slowdowns.
Yet, it's important to keep performance in mind, the more iterations the less performance, the more PredictionObjects the less performance.

End:

Finally, thank you for downloading this asset, hopefully you can make good use of it, please report any bugs or issues to moe4baker@gmail.com, and please consider rating this asset :)
Cheers.