ProLibrary

Thank you for purchasing ProLibrary!

ProLibrary is a collection of scripts including Extensions, common Data Structures, Pathfinding algorithms, Managers and Utilities that were designed to make your work easier and faster.

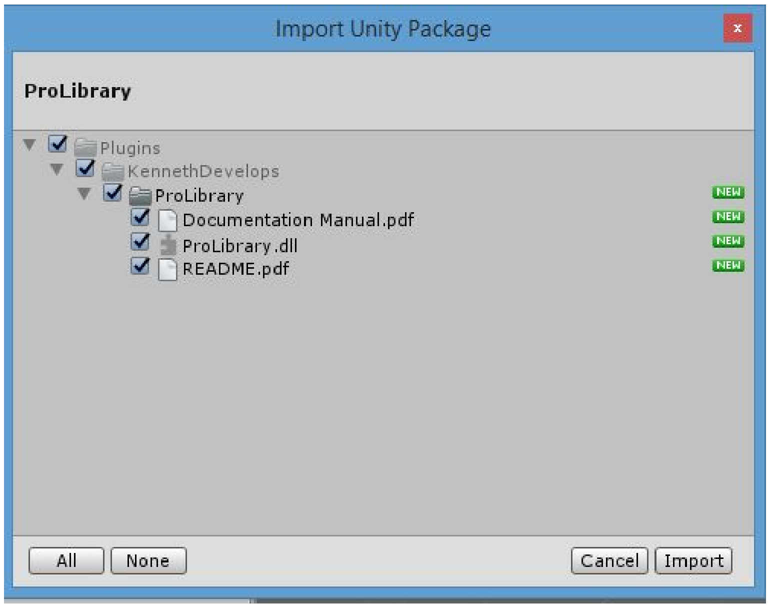
This document intends to serve as a guide to use and master this Asset Package. If possible, please avoid using this offline document as it may not be updated as often as the online version.

To go to the online version please click here

Table of Contents

Getting Started

Just after purchasing ProLibrary in the Asset Store you should be prompted to download and import the Asset and then the following dialog will appear:



As seen above, it is no more than a single file, so as long as you import that it is ok. I recommend you to not change the location of the library for precaution.

After the import is done, you'll see the Unity Editor loading the library and after a few seconds you'll be ready to begin using ProLibrary by just importing the namespace of the script you want to use, like this:

```
using KennethDevelops.ProLibrary.DataStructures;
```

Check the other pages in this document to see examples about how to use the library properly.

Support

If you have encountered any issues with ProLibrary please contact me at kennethdevelops+support@gmail.com

Please be kind enough to explain the issue as best as you can. Images, gifs and/or video will also be appreciated if you think it could help to understand the problem.

For any other comment on this wiki or ProLibrary itself, feedback or any other kind of suggestions you may have, please contact me at kennethdevelops+contact@gmail.com

Data Structures

A Data Structure is a way of organizing or managing data to enable a more efficient way to access and modify this data.

ProLibrary provides common Data Structures to manage data in a better way and spare you from writing this code yourself every time a new project comes along.

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's Data Structures:

```
using KennethDevelops.ProLibrary.DataStructures;
```

If you want to also use the Pool, you will have to import its namespace too:

```
using KennethDevelops.ProLibrary.DataStructures.Pool;
```

IPoolObject

IPoolObject is an interface that every class has to implement in order to become a Pool Object managed by a PoolManager

Methods

- *OnAcquire()* - Method that will be called just after the IPoolObject is Acquired from the pool.
- *OnDispose()* - Method that will be called just before the IPoolObject is Disposed from the pool.

Getting Started

To implement this interface, you first need to import the namespace of ProLibrary's IPoolObject:

```
using KennethDevelops.ProLibrary.DataStructures.Pool;
```

To know how to use the IPoolObject interface, please read further into PoolManager's Getting Started section

LookUpTable

A LookUp Table is a Data Structure containing an array that maps input keys to output values.

It is used to replace runtime computation with a simpler array indexing operation. In other words, it is used to execute a process or calculation only once for the same input.

Constructors

- *LookUpTable<TKey, TValue>(Func<TKey, TValue> process)* - Initializes a new instance of the LookUpTable class with the function that will be used to calculate the output value for every input key.

Properties

- *this[TKey key]* - Returns the requested value.

Methods

- *GetValue(TKey key)* - Returns the requested value.

Getting Started

First of all, you need to import the namespace of ProLibrary's Data Structures:

```
using KennethDevelops.ProLibrary.DataStructures;
```

Then you can instantiate it in your script wherever you want by passing as a parameter the method you want outputs to be calculated with:

```
void Start()
{
    var lookUpTable = new LookUpTable<int, int>(MultiplyByTwo);
}
```

```
public int MultiplyByTwo(int a)
{
    return a * 2;
}
```

In the example above we can see that we are initializing a LookUpTable by passing it the method "MultiplyByTwo" as the only parameter.

Now, to get a value we simply type:

```
var value = lookUpTable[key];
```

Or...

```
var value = lookUpTable.GetValue(key);
```

The *key* being the input of the process. Once you get a value with a key it won't be necessary to run it any further for that same key, as it is now stored in its memory.

PriorityQueue

A PriorityQueue is a Queue Data Structure with the main difference being that when Dequeue() is called, the element with the lighter weight is returned instead of the first element to Enqueue.

This Data Structure is used in Pathfinding algorithms like Dijkstra, AStar and ThetaStar.

Constructors

- *PriorityQueue<T>()* - Creates a new PriorityQueue instance, T being the type of the value of each WeightedNode element.

Properties

- *IsEmpty* - Returns true if the Queue does not contain any elements.

Methods

- *Enqueue(WeightedNode<T> element)* - Adds an element to the Queue.
- *Dequeue()* - Returns and removes from the Queue the element with minimum weight in the Queue.

Getting Started

The only thing you need to do is import the namespace of ProLibrary's Data Structures:

```
using KennethDevelops.ProLibrary.DataStructures;
```

Tree

A Tree is a Data Structure that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent TreeNode, represented as a set of linked nodes.

Constructors

- *Tree<T>(T element)* - Creates a Tree instance with an element passed as the root node.
- *Tree<T>(TreeNode<T> root)* - Creates a Tree instance with a TreeNode<T> passed as the root node.

Properties

- *root* - The root TreeNode<T> of the Tree.

TreeNode

A TreeNode is an element of a Tree data structure.

Constructors

- *TreeNode<T>(T element, TreeNode<T> parent)* - Creates a TreeNode instance with an element passed as its value and the corresponding parent of the node.

Properties

- *element* - The element value contained into the TreeNode.
- *parent* - The parent of the current node.
- *leaves* - The children of the current node.

Methods

- *AddLeaf(TreeNode<T> leaf)* - Adds a child.
- *AddLeaf(T leaf)* - Adds a child.
- *RemoveLeaf(TreeNode<T> leaf)* - Removes a child.
- *IsRoot()* - Returns true if this TreeNode is the root of the Tree by checking whether its parent is null or not.

WeightedNode

A WeightedNode is a Data Structure that pairs an element of type T to a weight.

Constructors

- *WeightedNode<T>(T element, float weight)* - Creates a WeightedNode instance.

Properties

- *Element* - The element value contained into the WeightedNode.
- *Weight* - The weight paired to the value of the WeightedNode.

Managers

Getting Started

The only thing you need to do is import the namespace of ProLibrary's Managers:

```
using KennethDevelops.ProLibrary.Managers;
```

PoolManager

An object pool is a design pattern that stores a set of objects that are ready to use. A client of the pool will request an object from the pool (AcquireObject) and perform operations on the returned object. When the client has finished, it returns the object to the pool (Dispose) rather than destroying it.

The PoolManager facilitates the creation and configuration of an object pool.

Properties

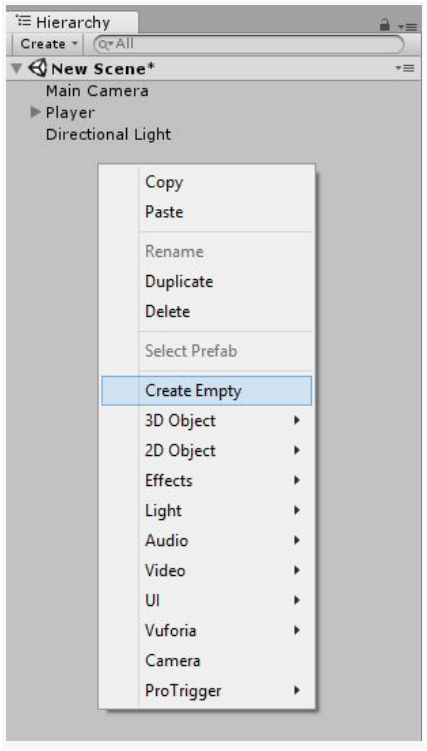
- *destroyOnLoad* - Whether or not this Pool is destroyed when a Scene is loaded.
- *initialQuantity* - The amount of GameObjects that will initially be created when the game starts.
- *prefab* - The prefab of the GameObject that will be instantiated.

Methods

- *GetInstance(string poolName)* - Returns a PoolManager by its GameObject's name.
- *AcquireObject(Vector3 position, Quaternion rotation)* - Acquires an object from the pool, while asigning it its position and rotation.
- *Dispose(IPoolObject poolObject)* - Returns an object to the pool.

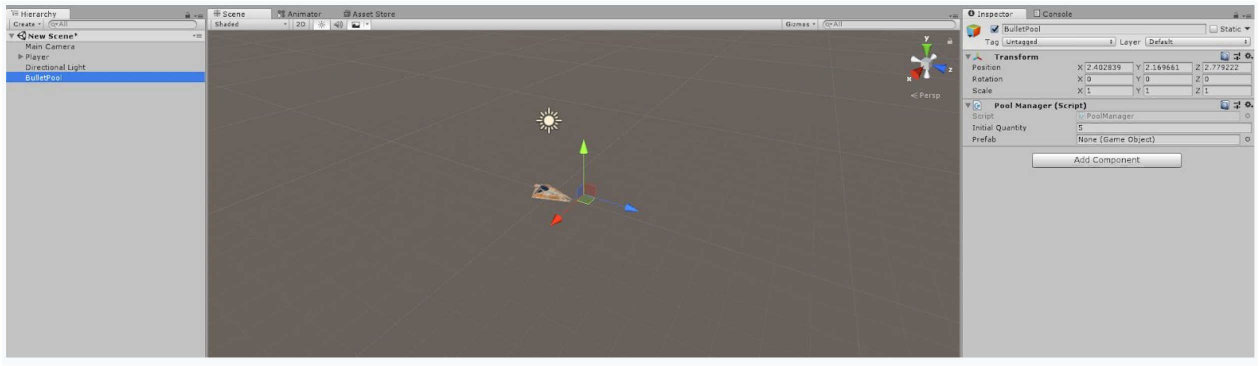
Getting Started

First of all, we need to create an empty GameObject



Now rename it to whatever you want, in this example we're making a BulletPool, so that'll be its name. Remember this name, it is important to access this pool later.

Then we add the component "PoolManager" to it



```
BulletPool                          Static
Tag  Untagged           Layer  Default
Transform
Position   X 2.402839   Y 2.169661   Z 2.779222
Rotation   X 0          Y 0          Z 0
Scale      X 1          Y 1          Z 1
Pool Manager (Script)
Script              PoolManager
Initial Quantity    5
Prefab              None (Game Object)
```

As we can see, PoolManager has 2 parameters, *initialQuantity* (the amount of GameObjects that will initially be created when the game starts) and *prefab* (the prefab of the GameObject that will be instantiated). For this example, we'll let initialQuantity in its default value(5) and we'll be focusing on the prefab.

Now, we can't just put any prefab in there, it has to be a GameObject containing a script that inherits from both MonoBehaviour and IPoolObject. So, we create one.

First of all, we make sure we are importing the Pool and the Managers namespace:

```
using KennethDevelops.ProLibrary.Managers;
using KennethDevelops.ProLibrary.DataStructures.Pool;
```

Then we can inherit from MonoBehaviour and implement the interface IPoolObject.

```
public class Bullet : MonoBehaviour, IPoolObject{

    public void OnAcquire()
    {

    }

    public void OnDispose()
    {

    }

}
```

We can see that when implementing IPoolObject we are implementing the methods *OnAcquire()*and *OnDispose()*. The first will be called every time the GameObject is retrieved from the pool and the second will be called every time it is about to be disabled and return to the pool.

But before focusing on that, we'll add some code to make the bullet move:

```
public class Bullet : MonoBehaviour, IPoolObject{

    public float speed;

    void Update()
    {
        transform.position += transform.forward * speed * Time.deltaTime;
    }

    public void OnAcquire()
    {

    }

    public void OnDispose()
    {

    }

}
```

Ok, so that'll make our bullet move successfully... But we don't want it to move to infinity, we want it to be disposed after the bullet is out of sight. Now, there's multiple ways to approach this but for now we're making a Coroutine that after some time will call the PoolManager and dispose itself from the pool. And we have the perfect tool for that in ProLibrary: WaitForSecondsAndDo

First, we import the WaitForSecondsAndDo's namespace:

```
using KennethDevelops.ProLibrary.Util;
```

And then we can focus on the Bullet's script:

```
public class Bullet : MonoBehaviour, IPoolObject{

    public float speed;
    public float secondsToDestroy = 2;

    private Coroutine _disposeCoroutine;


    void Update()
    {
        transform.position += transform.forward * speed * Time.deltaTime;
    }

    public void OnAcquire()
    {
        _disposeCoroutine = StartCoroutine(new WaitForSecondsAndDo(secondsToDestroy,
DisposeBullet));
    }

    private void DisposeBullet()
    {
        PoolManager.GetInstance("BulletPool").Dispose(this);
    }

    public void OnDispose()
    {
        if (_disposeCoroutine != null) StopCoroutine(_disposeCoroutine);
    }

}
```

To go a little slower, first we added a public variable to define how much time to wait till the bullet is disposed:

```
    public float secondsToDestroy = 2;
```

Then, we want to start the coroutine every time the Bullet is spawned:

```
    public void OnAcquire()
     {
         _disposeCoroutine = StartCoroutine(new WaitForSecondsAndDo(secondsToDestroy,
DisposeBullet));
     }
```

We use the WaitForSecondsAndDo utility and we pass it the time we want it to wait (secondsToDestroy) and the function that will dispose the bullet (DisposeBullet).

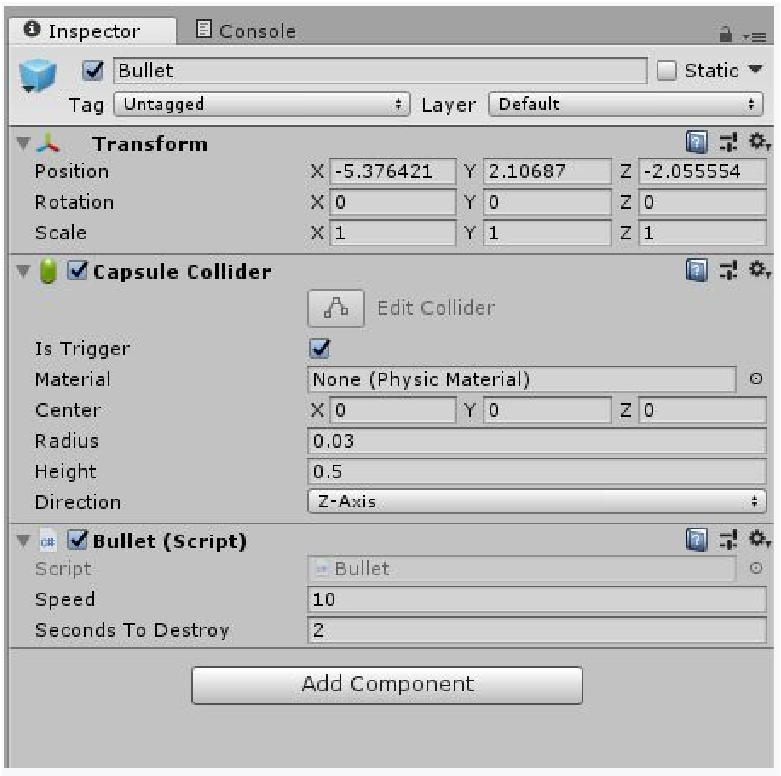
To dispose the bullet we get the PoolManager with the name of its GameObject, earlier we named it "BulletPool". Then we execute the method Dispose and pass it the IPoolObject we want to return to the pool, in this case, the same instance of Bullet that's executing the function: *this*.

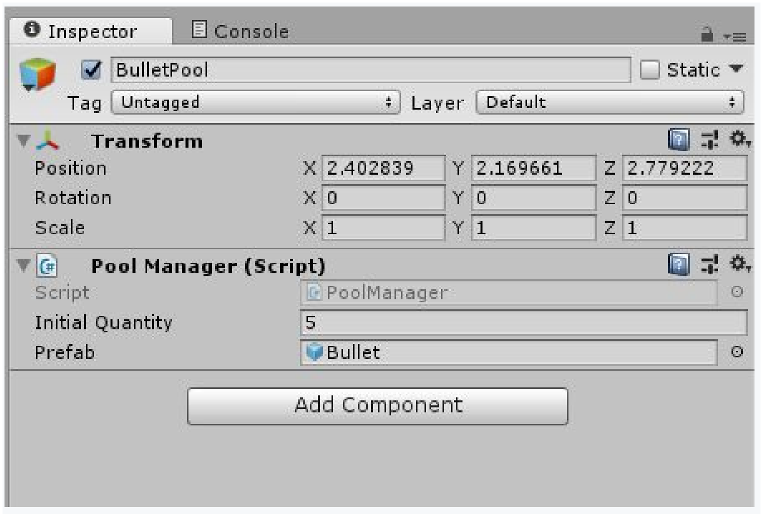
```
    private void DisposeBullet()
     {
         PoolManager.GetInstance("BulletPool").Dispose(this);
     }
```

And when we dispose the bullet, we want to stop the coroutine in case it's still running. This is in case the bullet may be disposed for other reasons, like it being destroyed because it collided with something. Right now we're not going to focus on damage and collisions, but it's a good practice to prevent these scenarios from an early stage.

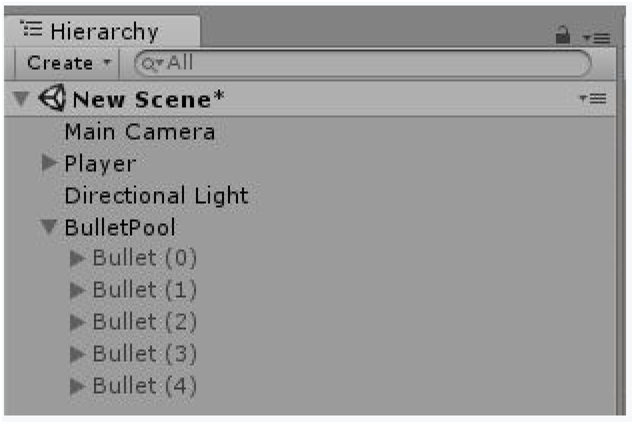
```
    public void OnDispose()
     {
         if (_disposeCoroutine != null) StopCoroutine(_disposeCoroutine);
     }
```

Now that we finally finished making our Bullet script, we attach it to a prefab and put that prefab in the PoolManager:


```
Inspector   Console
Bullet                                   Static
Tag Untagged                Layer Default
Transform
Position            X -5.376421  Y 2.10687  Z -2.055554
Rotation            X 0          Y 0        Z 0
Scale               X 1          Y 1        Z 1
Capsule Collider
                    Edit Collider
Is Trigger          ✓
Material            None (Physic Material)
Center              X 0          Y 0        Z 0
Radius              0.03
Height              0.5
Direction           Z-Axis
Bullet (Script)
Script              Bullet
Speed               10
Seconds To Destroy  2
Add Component
```


```
Inspector   Console
BulletPool                               Static
Tag Untagged                Layer Default
Transform
Position            X 2.402839   Y 2.169661  Z 2.779222
Rotation            X 0          Y 0         Z 0
Scale               X 1          Y 1         Z 1
Pool Manager (Script)
Script              PoolManager
Initial Quantity    5
Prefab              Bullet
Add Component
```

And if we hit "Play" we'll see that the Pool successfully instantiates the amount of Bullets we told it to at the start of the game:



```
New Scene*
    Main Camera
  ▶ Player
    Directional Light
  ▼ BulletPool
      ▶ Bullet (0)
      ▶ Bullet (1)
      ▶ Bullet (2)
      ▶ Bullet (3)
      ▶ Bullet (4)
```

Now, what kind of game is it if we can't even fire those bullets, right? So we'll go into the main character's script and make it able to shoot through the PoolManager.

We have our simple Player script here, which is only able to rotate. We also have the position from which we'll spawn bullets (bulletSpawn).

```
public class Player : MonoBehaviour{

    public float rotateAngle = 1;
    public Transform bulletSpawn;

    void Update()
    {
        Rotate(Input.GetAxisRaw("Horizontal"));
    }

    private void Rotate(float axis)
    {
        var rotation = Quaternion.AngleAxis(axis * rotateAngle, Vector3.up);
        transform.forward = rotation * transform.forward;
    }

}
```

Now, to shoot Bullets we first need to access to the PoolManager, our "BulletPool". For this we import PoolManager's namespace:

```
using KennethDevelops.ProLibrary.Managers;
```

And then we can access the PoolManager:

```
var pool = PoolManager.GetInstance("BulletPool");
```

Then we can tell it to give us one object from the pool and specify the position and rotation it will have when it's spawned:

```
pool.AcquireObject<Bullet>(bulletSpawn.position, transform.rotation);
```

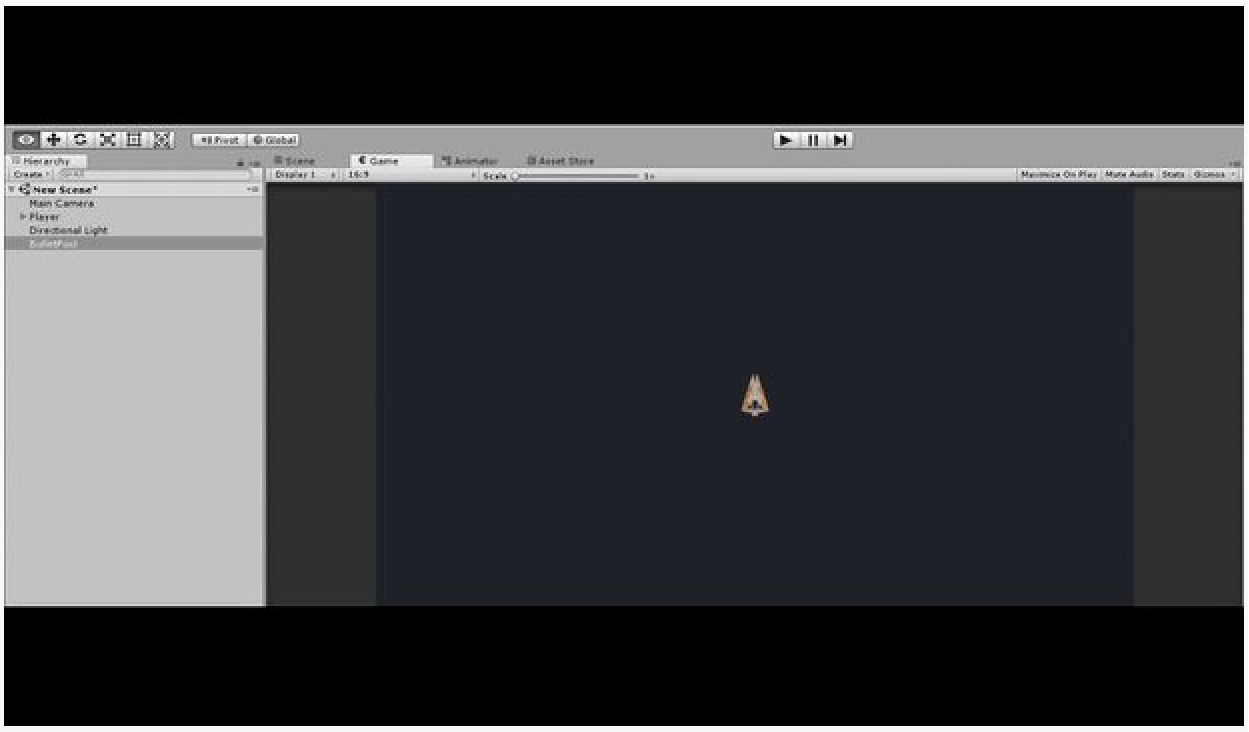
The method "AcquireObject()" also returns the Bullet instance so we can perform many more operations to the returned IPoolObject, but that won't be necessary for this guide.

Now we just tell it to shoot when we press the space key and there we go:

```
public class Player : MonoBehaviour{

    public float rotateAngle = 1;
    public Transform bulletSpawn;

    void Update()
    {
        Rotate(Input.GetAxisRaw("Horizontal"));

        if (Input.GetKeyDown(KeyCode.Space))
        {
            var pool = PoolManager.GetInstance("BulletPool");
            pool.AcquireObject<Bullet>(bulletSpawn.position, transform.rotation);
        }
    }

    private void Rotate(float axis)
    {
        var rotation = Quaternion.AngleAxis(axis * rotateAngle, Vector3.up);
        transform.forward = rotation * transform.forward;
    }

}
```

And here we have it, a totally functional BulletPool working with ProLibrary's PoolManager. You can see that after 2 seconds have passed the Bullets dispose themselves.

Also, you can clearly see that when the Player asks for more Bullets than the Pool has, the PoolManager instantiates even more.



ResourcesManager

The ResourcesManager allows us to Load any Asset from the Resources folder in almost the same way as UnityEngine's Resources class. The main difference being that ResourcesManager uses a LookUpTable to save any Asset previously loaded on memory, to increase performance and not reload it multiple times in game.

Methods

- *Load<T>(string url)* - Loads an asset stored at path in a Resources folder. When using the empty string (i.e., ""), the function will load the entire contents of the Resources folder.

Pathfinding

Getting Started

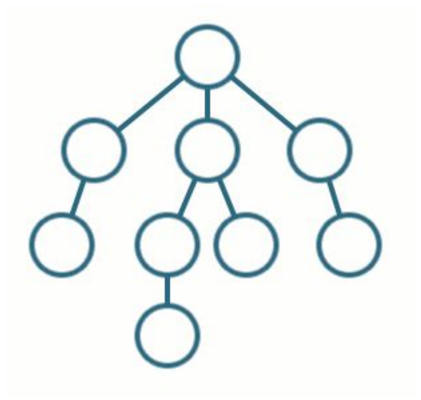
The only thing you need to do is import the namespace of ProLibrary's Pathfinding:

```
using KennethDevelops.ProLibrary.Algorithms.Pathfinding;
```

DFS

DFS (Depth First Search) is an algorithm for searching Tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

This algorithm could also be of use in bigger procedural algorithms.



A gif representing how a DFS algorithm works. We could not find the original poster of this gif.

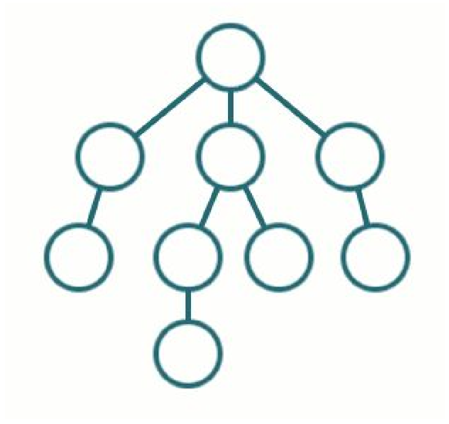
Methods

- *CalculatePath* - Calculates a path using Depth First Search. Returns the path from start node to goal.
    - *T start* - The node where it starts calculating the path
    - *Func<T, bool> isGoal* - A method that, given a node, tells us whether we reached our goal or not
    - *Func<T, IEnumerable<T>> explode* - A method that returns all near neighbours of a given node.

BFS

BFS (Breadth First Search) is an algorithm for searching Tree or graph data structures. The algorithm starts at the root node and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

This algorithm could also be of use in bigger procedural algorithms.



A gif representing how a BFS algorithm works. We could not find the original poster of this gif.

Methods

- *CalculatePath* - Calculates a path using Breadth First Search. Returns the path from start node to goal.
    - *T start* - The node where it starts calculating the path
    - *Func<T, bool> isGoal* - A method that, given a node, tells us whether we reached our goal or not
    - *Func<T, IEnumerable<T>> explode* - A method that returns all near neighbours of a given node.

Dijkstra

Dijkstra is an algorithm for finding the shortest path between nodes in a graph.

Methods

- *CalculatePath* - Calculates a path using Dijkstra. Returns a path from start node to goal. Returns null if a path could not be found.
    - *T start* - The node where it starts calculating the path
    - *Func<T, bool> isGoal* - A method that, given a node, tells us whether we reached our goal or not
    - *Func<T, IEnumerable<WeightedNode<T>>> explode* - A method that returns all near neighbours of a given node.

AStar

AStar is an algorithm for finding the shortest path between nodes in a graph.

Methods

- *CalculatePath* - Calculates a path using AStar. Returns a path from start node to goal. Returns null if a path could not be found.
    - *T start* - The node where it starts calculating the path
    - *Func<T, bool> isGoal* - A method that, given a node, tells us whether we reached our goal or not
    - *Func<T, IEnumerable<WeightedNode<T>>> explode* - A method that returns all near neighbours of a given node.
    - *Func<T, float> getHeuristic* - A method that returns the heuristic of the given node.

ThetaStar

ThetaStar is an algorithm for finding the shortest path between nodes in a graph.

Methods

- *CalculatePath* - Calculates a path using AStar. Returns a path from start node to goal. Returns null if a path could not be found.
    - *T start* - The node where it starts calculating the path
    - *Func<T, bool> isGoal* - A method that, given a node, tells us whether we reached our goal or not
    - *Func<T, IEnumerable<WeightedNode<T>>> explode* - A method that returns all near neighbours of a given node.
    - *Func<T, float> getHeuristic* - A method that returns the heuristic of the given node.
    - *Func<T, float> getCost* - A method that returns the cost to transition from one node to another.
    - *Func<T, float> lineOfSight* - A method that returns whether the path from one to another can be direct or not (due to obstacles in the way).

Utilities

Getting Started

The only thing you need to do is import the namespace of ProLibrary's Utilities:

```
using KennethDevelops.ProLibrary.Util;
```

WaitForSecondsAndDo

WaitForSecondsAndDo is a Utility class, similar to UnityEngine's WaitForSeconds with the difference that after the time has passed it performs an operation.

Constructors

- *WaitForSecondsAndDo(float seconds, Action action)* - Creates a new WaitForSecondsAndDo instance, with *seconds* being the amount of seconds to wait before performing an action, and *action* the method to invoke.

Methods

- *StopAndDo()* - Stops the coroutine and performs the *action*.

Getting Started

To start using WaitForSecondsAndDo you first need to import its namespace:

```
using KennethDevelops.ProLibrary.Util;
```

And then start the coroutine:

```
StartCoroutine(new WaitForSecondsAndDo(2f, HelloWorld));
```

With HelloWorld being a method that receives and returns nothing:

```
private void PerformOperation()
{
    Debug.Log("Hello World");
}
```

You can use lambdas too:

```
StartCoroutine(new WaitForSecondsAndDo(2f, () => Debug.Log("Hello World")));
```

WaitForConditionAndDo

WaitForConditionAndDo is a Utility class, similar to WaitForSecondsAndDo with the difference that it only executes the action when the condition is true.

Constructors

- *WaitForConditionAndDo(Func<bool> condition, Action action)* - Creates a new WaitForConditionAndDo instance, with *condition* being the condition that will be expected to be true before performing an action, and *action* the method to invoke.

Methods

- *StopAndDo()* - Stops the coroutine and performs the *action*.

Getting Started

To start using WaitForConditionAndDo you first need to import its namespace:

```
using KennethDevelops.ProLibrary.Util;
```

And then start the coroutine:

```
StartCoroutine(new WaitForConditionAndDo(CanShowValue, ShowValue));
```

With CanShowValue being a method that receives nothing and returns a bool:

```
private bool CanShowValue()
{
    return value > 40;
}
```

And ShowValue being a method that receives and returns nothing:

```
private void ShowValue()

{

    Debug.Log(value);

}
```

You can use lambdas too:

```
StartCoroutine(new WaitForSecondsAndDo(() => value > 40, () => Debug.Log(value)));
```

DoNotDestroyOnLoad

The DoNotDestroyOnLoad component serves the purpose of not having to manually modify every script tha you suddenly don't want to mark as "DontDestroyOnLoad". You can simple tell it to mark it on Awake(), Start() or after X amount of seconds (Custom).

Properties

- *removeDuplicates* - Whether or not to remove duplicates on the new loaded scene.
- *executeOn* - When to mark it as DontDestroyOnLoad. Can be Awake(), Start() or Custom() (where you choose how many seconds after the game starts)
- *customTimeToExecute* - If executeOn is "Custom", it represents the amount of seconds it will wait after the game starts to mark the gameobject as "DontDestroyOnLoad".

Getting Started

To make use of this component, you can simply select your gameObject and in the inspector window, click on the "Add Component" button and type "DoNotDestroyOnLoad" then select it and there you go!

Extensions

In C#, Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

ProLibrary provides Extensions to solve some common problems most programmers have and reduce the amount of lines of code needed to achieve your goal.

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

ArrayExtensions

Methods

- *GetRandomItem()* - Returns a random item of the List/Array
- *Randomize()* - Returns a new randomized(items changes places randomly) List/Array.

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

CameraExtensions

Methods

- *IsInFov(GameObject gameObject)* - Returns whether or not the gameObject is visible to the camera.

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

ColorExtensions

Methods

- *Invert() - Returns a new Color with an inverted color value.*
- *Alpha(float alpha) - Returns a new Color with a modified alpha value.*
- *Red(float red) - Returns a new Color with a modified red value.*
- *Green(float green) - Returns a new Color with a modified green value.*
- *Blue(float blue) - Returns a new Color with a modified blue value.*
- *ToGrayscale() - Returns the grayscale version of the Color.*
- *HtmlToColor(this string hex, Color originalColor = new Color()) - Tries to create a new Color with the prompted hex color value. If it fails it will return the value of originalColor*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

FloatExtensions

Methods

- *IsInRangeInclusive(float from, float to) - Returns if the value is in the defined range of values, including from and to.*
- *IsInRangeExclusive(float from, float to) - Returns if the value is in the defined range of values, excluding from and to.*
- *ClampAngle(float min, float max) - Clamps a value between a minimum and a maximum angle value (in degrees).*
- *NormalizeAngle() - Normalizes and angle value (in degrees), making it be between the value ranges of 0 and 360. This means that the value 365 will be changed to 5, and the value -5 will be changed to 355*
- *DegreesToRadians() - Converts an angle in degrees to radians.*
- *RadiansToDegrees() - Converts an angle in radians to degrees.*
- *Pow(float exponent) - Returns the value raised to power exponent.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

GenericExtensions

Methods

- *IsIn(param T[] values) - Returns true if an object is equal to any of the values.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

IntExtensions

Methods

- *IsInRangeInclusive(int from, int to) - Returns if the value is in the defined range of values, including from and to.*
- *IsInRangeExclusive(int from, int to) - Returns if the value is in the defined range of values, excluding from and to.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

LayerMaskExtensions

Methods

- *IncludesLayer(int layer) - Returns if the LayerMask includes the layer value.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

RectExtensions

Methods

- *GetCenter() - Returns the position (Vector2) of the Rect's center.*
- *AddX(float x) - Adds a x value to the current Rect.*
- *AddY(float y) - Adds a y value to the current Rect.*
- *AddWith(float width) - Adds a width value to the current Rect.*
- *AddHeight(float height) - Adds a height value to the current Rect.*
- *SetX(float x) - Sets the x value to the current Rect.*
- *SetY(float y) - Sets the y value to the current Rect.*
- *SetWith(float width) - Sets the width value to the current Rect.*
- *SetHeight(float height) - Sets the height value to the current Rect.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

TransformExtensions

Methods

- *GetDirectChildren() - Returns the direct childs of the Transform instance, excluding grandchildren, great-grand children and so on.*
- *GetComponentsInDirectChildren<T>() - Returns the direct children's components of the Transform instance, excluding grandchildren, great-grand children and so on.*
- *GetComponentsInDirectChildren<T>() - Returns the direct children's components of the GameObject instance, excluding grandchildren, great-grand children and so on.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```

Vector3Extensions

Methods

- *SetX(float x) - Returns a new Vector3 with a modified x value.*
- *SetY(float y) - Returns a new Vector3 with a modified y value.*
- *SetZ(float z) - Returns a new Vector3 with a modified z value.*

- *SqrDistance(Vector3 other) - Returns the squared distance between two vectors*
- *ToVector2XY(Vector3 other) - Returns a new Vector2 using the given vector's x as x and the y as y.*
- *ToVector2XZ(Vector3 other) - Returns a new Vector2 using the given vector's x as x and the z as y.*

- *GetPerpendicularDirector() - Returns the perpendicular director of the current Vector3.*

- *IsRight(Vector3 right) - Returns if current vector's direction is relatively Right, right being the relative Right direction used for comparison.*
- *IsLeft(Vector3 right) - Returns if current vector's direction is relatively Left, right being the relative Right direction used for comparison.*
- *IsLateral(Vector3 right) - Returns if current vector's direction is either right or left relative to a custom right direction.*
- *IsForward(Vector3 forward) - Returns if current vector's direction is relatively Forward, forwardbeing the relative forward used for comparison.*
- *IsBackward(Vector3 forward) - Returns if current vector's direction is relatively Backward, forward being the relative Forward direction used for comparison.*
- *IsFrontal(Vector3 forward) - Returns if current vector's direction is either forward or backward relative to a custom Forward direction.*

Getting Started

The only thing you need to do is to import the namespace of ProLibrary's extensions.

```
using KennethDevelops.ProLibrary.Extensions;
```