Quadtree Spatial Partitioning for Unity

Daniel Dolejška
dolejskad@gmail.com
https://github.com/dolejska-daniel/unity-quadtree

December 18, 2020

Contents

1 Introduction

This package provides generic implementation of the quadtree[1] spatial partitioning algorithm. The aim of this package is to provide an out of the box working solution for simple 2D spatial partitioning of Unity's `GameObjects` but at the same time allowing the implementation to be easily extended, modified and used with *any* items that you would want it to. The tree can be used as a component of a `GameObject` or it can just as easily be used only in the scripts without it being attached to any `GameObject`.

As mentioned before this package provides out of the box solution for `GameObjects` — more on that topic in section 3.1. Programmatic usage of the quadtree implementation is further described in section 3.2. Section 4 goes into some useful details of the implemented quadtree structures. Finally section 5 describes ways to extend the current implementation.

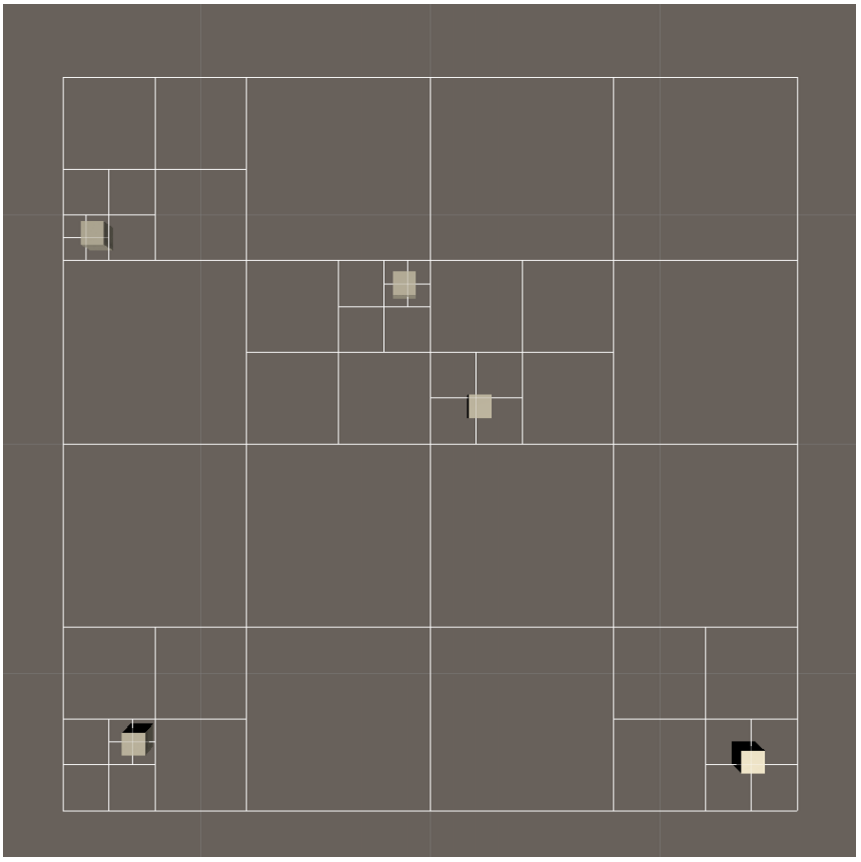

Figure 1: Visualization of the quadtree structure for 5 `GameObjects`

A view of the gizmos generated by the quadtree implementation for Unity's `GameObjects` from the editor.

2 Package Contents

`Scenes/` directory contains example Unity scene(s) demonstrating usage of the package.

`Documentation/` directory contains offline documentation.

`Scripts/` directory contains all package's script source codes.

[1] https://en.wikipedia.org/wiki/Quadtree

3 Usage

This section describes the usage of the scripts provided by this package. Simple editor usage for `GameObject`s with `Renderer` components is described in 3.1. Section 3.2 then describes how to use the structures in the scripts.

3.1 From Editor (Out of the Box)

Section 3.1.2 explains how are objects stored in the tree structure and how it is set-up. Then section 3.1.3 gives an example how to search the structure and find previously stored objects. Finally section 3.1.4 describes how the item removal from the tree works and is used.

3.1.1 Setting Up the Tree

Before we can play with the items in the tree, we first need to add a root of our quadtree to the scene. The initial setup of the root of the quadtree is extremely simple:

- Create empty `GameObject`.

  This can be done by right-clicking in the *Hierarchy* window and selecting "Create Empty". `GameObject` you've now created will be used as a root for the tree. Keep your newly created empty `GameObject` instance selected.

- From window dropdown menu select: *Component* > *Spatial partitioning* > *Quadtree* > *Rootnode (for GameObjects)*.

  This command will add `GameObjectQuadtreeRoot` script component to the previously created `GameObject`. At this point you should be able to see white rectangle on the screen (make sure your gizmos are enabled and showing up in the editor).

3.1.2 Storing Objects

Now with the tree root set-up

- Add `GameObject`s you wish to be in the tree as children of the tree root `GameObject`.

  The item must have any `Renderer` component — `MeshRenderer`, `SpriteRenderer`, ... For example create any `GameObject` with `Renderer` component — do that by right-clicking on the tree root `GameObject` and selecting anything from submenu *3D Object* or add your own! You've now added object supported by the quadtree, keep it selected in the *hierarchy* window.

- From window dropdown menu select: *Component* > *Spatial partitioning* > *Quadtree* > *Items* > *Renderer-based Item*.

  This command will add `RendererItem` script component to the previously added `GameObject`. At this point you should be able to see that the previous simple rectangle has somewhat changed. Try moving the item around, you should see the quadtree structure changing based on the current position of the item.

3.1.3 Looking up Objects

Looking up objects from editor does not make much sense, since you will probably want to look the items up in a code which will then work with them. How to look up the objects in the tree is described in section 3.2.2.

3.1.4 Removing Objects

Removing the items stored in the tree with `Renderer`-based item components is just removing either the `RendererItem` script component from the `GameObject` or removing the `GameObject` itself. Simple as that.

3.2 From Code

This section describes how can you control the tree structure from the code.

3.2.1 Storing Objects

The stored items do not necessarily need to be children of the root node, but it allows the tree to fill completely automatically. You can add items to the tree using scripts too, thought the item type must match with the item type of the tree root. To add item to the tree follow the code shown in Listing 1.

```
// get the tree root component of the tree you want to insert to
var root = GetComponent<GameObjectQuadtreeRoot>();
// locate the GameObject to be added into the tree
var itemGO = GameObject.Find("Game Object Name");

// locate the tree item component of the GameObject
var item = itemGO.GetComponent<RendererItem>();

// insert the item to the tree using the Insert method
// the tree itself will find the smallest node to store it in
root.Insert(item);
```

Listing 1: Inserting Item to the Tree

This code snippet first locates the root of the quadtree, the `GameObject` to be inserted into the tree and its quadtree item component. Algorithm then inserts the item component into the tree via its root.

3.2.2 Looking up Objects

To look up the stored objects you will need to use Unity's `Bounds`[2] object. This object describes an area in which the algorithm will look for the items. Follow code shown in Listing 2.

```
// get the tree root component of the tree you want to search in
var root = GetComponent<GameObjectQuadtreeRoot>();

// locate or define the Bounds to be used in the search
// bounds defined below will find all items with bounds
// intersecting between (-5, 0, -5) and (5, 0, 5)
var bounds = new Bounds(Vector3.zero, new Vector3(10f, 0f, 10f));

// pass the bounds to the Find method of the root and
// the tree itself will find all the items with intersecting bounds
var items = root.Find(bounds);

// the Find method returns List of all found items
// continue as you will
foreach (var item in items)
{
    // process found items ...
}
```

Listing 2: Looking Up Stored Objects

This code snippet first locates the root of the quadtree and creates the `Bounds` object which defines the search area. Algorithm then locates all items intersecting with the provided `Bounds` via tree root.

[2] https://docs.unity3d.com/ScriptReference/Bounds-ctor.html

3.2.3 Removing Objects

Removing the items from the tree is as easy as adding them in. There are actually two ways to delete the item from the tree. The location of the `GameObject` and its tree item component is the same for both possibilities.

The first removal option is to use the reference to the parent node, by which is the item currently contained, from the item object instance. Since the item is actually contained by that node, the algorithm won't have to traverse the tree and look for the node which contains the item. With the variables defined in the snippet above:

```
// locate the GameObject to be removed from the tree
var itemGO = GameObject.Find("Game Object Name");
// locate the tree item component of the GameObject
var item = itemGO.GetComponent<RendererItem>();

// task the parent node with the removal of the item from the tree
item.ParentNode.Remove(item);
```

Listing 3: Removing the Item Via Its Parent Node

This code snippet removes the `GameObject`'s quadtree item from the tree via its own reference to the node in which it is currently contained.

The second option is to pass the item to the tree root itself to remove the item. The algorithm will have to traverse the tree to find correct node to remove the item from hence the former removal option is faster than the latter. With the variables defined in the snippet above:

```
// locate the GameObject to be removed from the tree
var itemGO = GameObject.Find("Game Object Name");
// locate the tree item component of the GameObject
var item = itemGO.GetComponent<RendererItem>();

// get the tree root component of the tree you want to remove from
var root = GetComponent<GameObjectQuadtreeRoot>();

// task the tree root with the removal of the item
root.Remove(item);
```

Listing 4: Removing the Item via Tree Root

This code snippet removes the `GameObject`'s quadtree item from the tree via root of the tree.

4 Implementation Details

This section aims to explain some important thoughts behind implementation of this library. It should help you understand how is this library supposed to generally work. The goal of this library is to allow maximum flexibility for you — the data structures are designed to allow *any* items to be stored in the tree — the only requirement is implementation of predefined interface of the items.

As part of this package, there is already prepared tree item implementation supporting any `GameObject` instances with `Renderer` components. This implementation works out of the box.

4.1 Tree Root

Tree root represents the tree itself and provides an entry point for all the algorithms operating on the tree structure. There always is only single tree root for a single tree instance. Tree root class is the main interface of the quadtree — any search, insertion or deletion starts in the root of the tree.

```
public interface IQuadtreeRoot<TItem, TNode>
    where TItem : IItem<TItem, TNode>
    where TNode : INode<TItem, TNode>
```

This is the interface that any tree root implementation should have. Thought if you wish to implement your own tree root class from scratch nothing is stopping you! Of course extending either `QuadtreeRoot` or `QuadtreeMonoRoot` is a more sensible thing to do. More on that topic in section 5.3.

```
public class QuadtreeRoot<TItem, TNode> : IQuadtreeRoot<TItem, TNode>
    where TItem : IItem<TItem, TNode>
    where TNode : INode<TItem, TNode>, new()
```

This is the actual used class implementing all that is necessary for a tree root. It allows anyone to further specify `TItem` and `TNode` which are classes of the items, respectively the nodes of the tree. The only constraint for item classes is an implementation of interface `TItem`. Tree node classes must implement `TNode` interface and have a parameterless constructor.

The previously mentioned class `QuadtreeMonoRoot` does actually not derive from `QuadtreeRoot`, because C# does not allow multiple inheritance and to be able to use the class as a `GameObject` component in Unity it must derive from `MonoBehaviour`. Hence `QuadtreeMonoRoot` only implements the interface `IQuadtreeRoot` as a proxy of `QuadtreeRoot`. The class `GameObjectQuadtreeRoot` then derives from `QuadtreeMonoRoot` and specifies its type arguments as shown below. Fully specifying types for generic classes is the last required step for the class to be used as a `GameObject` component.

```
public class GameObjectQuadtreeRoot : QuadtreeMonoRoot<GameObjectItem, Node<GameObjectItem>>
```

4.2 Tree Node

The `Node` class represents hierarchy of the tree. It contains items stored in the tree and nodes which are lower in the hierarchy but still within its own boundaries. There is usually no need to work with the nodes themselves — all actions should be done through root of the tree. As mentioned before every node implementation must implement the `INode` interface.

```
public interface INode<TItem, TNode>
    where TItem : IItem<TItem, TNode>
    where TNode : INode<TItem, TNode>
```

If it does and also has a public parameterless constructor, then you can use it instead of original `Node` implementation. More about implementing custom node classes in section 5.2.

The original implementation of the tree `Node`s derives from `NodeBase` which actually implements all the necessary functionality and, of course, the `INode` interface. What `Node` does is just type specification of the `TNode` type parameter of the `NodeBase` class.

```
public class Node<TItem> : NodeBase<TItem, Node<TItem>>
        where TItem : IItem<TItem, Node<TItem>> {}
```

4.3 Tree Item

Tree item classes represent the items in the tree. All the classes must implement the `IItem` interface so the nodes know how to operate with the items. More about implementing custom item classes in section 5.1.

```
public interface IItem<TItem, TNode>
    where TItem : IItem<TItem, TNode>
    where TNode : INode<TItem, TNode>
```

The implementation for `GameObjects` uses a number of classes but almost all the necessary logic is contained in the `GameObjectItemBase` class. This class implements the `IItem` interface and all the necessarry logic for subclasses to work *almost* automatically. The `GameObjectItemBase` is implemented as `abstract` and requires only `GetBounds()` *(and `This()`)* method to be implemented by its subclasses. It also derives from `MonoBehaviour` so it can be used as a `GameObject` component.

```
public abstract class GameObjectItemBase<TItem, TNode>
    : MonoBehaviour, IItem<TItem, TNode>
    where TItem : IItem<TItem, TNode>
    where TNode : INode<TItem, TNode>
```

The `GameObjectItem` class then derives from `GameObjectItemBase`. It, again, only specifies the type parameters of its superclass and extremely simple implementation of the `This()` method necessary for correct type checking.

```
public abstract class GameObjectItem
    : GameObjectItemBase<GameObjectItem, Node<GameObjectItem>>
{
    protected override GameObjectItem This() => this;
}
```

Finally the `RendererItem` class comes into light. It further derives from `GameObjectItem` and implements the required `GetBounds()` method so the `GameObjectItemBase` can work properly. This class specifically extracts the bounds from the `Renderer` component of the `GameObject` it is currently attached to.

5 Extending Implementation

Implementing custom structures and modifying existing ones should be pretty straightforward. Just subclass the structure you wish to modify, override the implemented methods, et voilà! It should be that simple.

5.1 Custom Items

If you implement custom item classes you will also need to create a custom tree root implementation which supports usage of those items (exception to this are subclasses of `GameObjectItem` using `GameObjectQuadtreeRoot`). But this is very simple too, as shown below.

```
public class CustomItem : IItem<CustomItem, Node<CustomItem>>
{
    // TODO: Implement required interface of IItem here
}

public class CustomQuadtreeRoot : QuadtreeRoot<CustomItem, Node<CustomItem>> {}
```

5.2 Custom Nodes

It's the same thing with the nodes as it is with the items — ff you implement custom node classes you will also need to create a custom tree root implementation which supports usage of those items.

```
public class CustomNode<TItem> : NodeBase<TItem, CustomNode<TItem>>
    where TItem : IItem<TItem, CustomNode<TItem>>
{
    // TODO: Override the NodeBase methods here
}

public class CustomQuadtreeRoot : QuadtreeRoot<CustomItem, CustomNode<CustomItem>> {}
```

5.3 Custom Roots

As shown a number of times in the previous sections – creating a custom tree root implementation is quite simple. No class body has been provided in the previous section since modification of the item's or node's type has been enough, but you can override any method of your choosing of course.

```
public class CustomQuadtreeRoot : QuadtreeRoot<CustomItem, CustomNode<CustomItem>>
{
    // TODO: Override the QuadtreeRoot methods here
    // TODO: Implement custom methods here
}
```