Optimizing garbage collection in Unity games

Checked with version: 5.5
Difficulty: Intermediate

Introduction

When our game runs, it uses memory to store data. When this data is no longer needed, the memory that stored that data is freed up so that it can be reused. *Garbage* is the term for memory that has been set aside to store data but is no longer in use. *Garbage collection* is the name of the process that makes that memory available again for reuse.

Unity uses garbage collection as part of how it manages memory. Our game may perform poorly if garbage collection happens too often or has too much work to do, which means that garbage collection is a common cause of performance problems.

In this article, we'll learn how garbage collection works, when garbage collection happens and how to use memory efficiently so that we minimize the impact of garbage collection on our game.

Diagnosing problems with garbage collection

Performance problems caused by garbage collection can manifest as low frame rates, jerky performance or intermittent freezes. However, other problems can cause similar symptoms. If our game has performance problems like this, the first thing we should do is to use Unity's Profiler window to establish whether the problems we are seeing are actually due to garbage collection.

To learn how to use the Profiler window to find the cause of your performance problems, please follow this tutorial (/learn/tutorials/topics/performance-optimization/diagnosing-performance-problems-using-profiler?playlist=44069).

A brief introduction to memory management in Unity

To understand how garbage collection works and when it happens, we must first understand how memory usage works in Unity. Firstly, we must understand that Unity uses different

approaches when running its own core engine code and when running the code that we write in our scripts.

The way Unity manages memory when running its own core Unity Engine code is called *manual memory management (https://en.wikipedia.org/wiki/Manual_memory_management)*. This means that the core engine code must explicitly state how memory is used. Manual memory management does not use garbage collection and won't be covered further in this article.

The way that Unity manages memory when running our code is called *automatic memory management (https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html)*. This means that our code doesn't need to explicitly tell Unity how to manage memory in a detailed way. Unity takes care of this for us.

At its most basic level, automatic memory management in Unity works like this:

- Unity has access to two pools of memory: the *stack* and the *heap* (also known as the *managed heap*. The stack is used for short term storage of small pieces of data, and the heap is used for longer term storage and larger pieces of data.
- When a variable is created, Unity requests a block of memory from either the stack or the heap.
- As long as the variable is in scope (/learn/tutorials/topics/scripting/scope-and-access-modifiers) (still accessible by our code), the memory assigned to it remains in use. We say that this memory has been *allocated*. We describe a variable held in stack memory as an *object on the stack* and a variable held in heap memory as an *object on the heap*.
- When the variable goes out of scope, the memory is no longer needed and can be returned to the pool that it came from. When memory is returned to its pool, we say that the memory has been *deallocated*. Memory from the stack is deallocated as soon as the variable it refers to goes out of scope. Memory from the heap, however, is not deallocated

at this point and remains in an allocated state even though the variable it refers to is out of scope.

- The *garbage collector* identifies and deallocates unused heap memory. The garbage collector is run periodically to clean up the heap.

Now that we understand the flow of events, let's take a closer look at how stack allocations and deallocations differ from heap allocations and deallocations.

What happens during stack allocation and deallocation?

Stack allocations and deallocations are quick and simple. This is because the stack is only used to store small data for short amounts of time. Allocations and deallocations always happen in a predictable order and are of a predictable size.

The stack works like a stack data type (https://en.wikipedia.org/wiki/Stack_(abstract_data_type)): it is a simple collection of elements, in this case blocks of memory, where elements can only be added and removed in a strict order. This simplicity and strictness is what makes it so quick: when a variable is stored on the stack, memory for it is simply allocated from the "end" of the stack. When a stack variable goes out of scope, the memory used to store that variable is immediately returned to the stack for reuse.

What happens during a heap allocation?

A heap allocation is much more complex than a stack allocation. This is because the heap can be used to store both long term and short term data, and data of many different types and sizes. Allocations and deallocations don't always

happen in a predictable order and may require very different sized blocks of memory.

When a heap variable is created, the following steps take place:

- First, Unity must check if there is enough free memory in the heap. If there is enough free memory in the heap, the memory for the variable is allocated.
- If there is not enough free memory in the heap, Unity triggers the garbage collector in an attempt to free up unused heap memory. This can be a slow operation. If there is now enough free memory in the heap, the memory for the variable is allocated.
- If there isn't enough free memory in the heap after garbage collection, Unity increases the amount of memory in the heap. This can be a slow operation. The memory for the variable is then allocated.

Heap allocations can be slow, especially if the garbage collector must run and the heap must be expanded.

What happens during garbage collection?

When a heap variable goes out of scope, the memory used to store it is not immediately deallocated. Unused heap memory is only deallocated when the garbage collector runs.

Every time the garbage collector runs, the following steps occur:

- The garbage collector examines every object on the heap.
- The garbage collector searches all current object references to determine if the objects on the heap are still in scope.
- Any object which is no longer in scope is flagged for deletion.
- Flagged objects are deleted and the memory that was allocated to them is returned to the heap.

Garbage collection can be an expensive operation. The more objects on the heap, the more work it must do and the more object references in our code, the more work it must do.

When does garbage collection happen?

Three things can cause the garbage collector to run:

- The garbage collector runs whenever a heap allocation is requested that cannot be fulfilled using free memory from the heap.
- The garbage collector runs automatically from time to time (although the frequency varies by platform).
- The garbage collector can be forced to run manually.

Garbage collection can be a frequent operation. The garbage collector is triggered whenever a heap allocation cannot be fulfilled from available heap memory, which means that frequent heap allocations and deallocations can lead to frequent garbage collection.

Problems with garbage collection

Now that we understand the role that garbage collection plays in memory management in Unity, we can consider the types of problems that might occur.

The most obvious problem is that the garbage collector can take a considerable amount of time to run. If the garbage collector has a lot of objects on the heap and/or a lot of object references to examine, the process of examining all of these objects can be slow. This can cause our game to stutter or run slowly.

Another problem is that the garbage collector may run at inconvenient times. If the CPU is already working hard in a performance-critical part of our game, even a small amount of additional overhead from garbage collection can cause our frame rate to drop and performance to noticeably change.

Another problem that is less obvious is *heap fragmentation*. When memory is allocated from the heap it is taken from the free space in blocks of different sizes depending on the size of data that must be stored. When these blocks of memory are returned to the heap, the heap can get split up into lots of small free blocks separated by allocated blocks. This means that although the total amount of free memory may be high, we are unable to allocate large blocks of memory without running the garbage collector and/or expanding the heap because none of the existing blocks are large enough.

There are two consequences to a fragmented heap. The first is that our game's memory usage will be higher than it needs to be and the second is that the garbage collector will run more frequently. For a more detailed discussion of heap fragmenation, see this Unity best practice guide on performance (https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html).

Finding heap allocations

If we know that garbage collection is causing problems in our game, we need to know is which parts of our code are generating garbage. Garbage is generated when heap variables go out of scope, so first we need to know what causes a variable to be allocated on the heap.

What is allocated on the stack and the heap?

In Unity, value-typed local variables are allocated on the stack and everything else is allocated on the heap. If you're unsure of

the difference between value and reference types in Unity, see this tutorial (/learn/tutorials/topics/scripting/data-types).

The following code is an example of a stack allocation, as the variable *localInt* is both local and value-typed. The memory allocated for this variable will be deallocated from the stack immediately after this function has finished running.

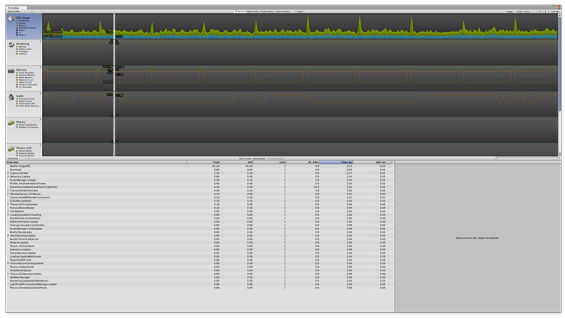
```
void ExampleFunction()
{
    int localInt = 5;
}
```

The following code is an example of a heap allocation, as the variable *localList* is local but reference-typed. The memory allocated for this variable will be deallocated when the garbage collector runs.

```
void ExampleFunction()
{
    List localList = new List();
}
```

Using the Profiler window to find heap allocations

We can see where our code is creating heap allocations with the Profiler window.



With the CPU usage profiler selected, we can select any frame to see CPU usage data about that frame in the bottom part of the Profiler window. One of the columns of data is called **GC alloc**. This column shows heap allocations that are being made in that frame. If we select

the column header we can sort the data by this statistic, making it easy to see which functions in our game are causing the most heap allocations. Once we know which function causes the heap allocations, we can examine that function.

Once we know what code *within* the function is causing garbage to be generated, we can decide how to solve this problem and minimize the amount of garbage generated.

Reducing the impact of garbage collection

Broadly speaking, we can reduce the impact of garbage collection on our game in three ways:

- We can reduce the time that the garbage collector takes to run.
- We can reduce the frequency with which the garbage collector runs.
- We can deliberately trigger the garbage collector so that it runs at times that are not performance-critical, for example during a loading screen.

With that in mind, there are three strategies that will help us here:

- We can organise our game so we have fewer heap allocations and fewer object references. Fewer objects on the heap and fewer references to examine means that when garbage collection is triggered, it takes less time to run.
- We can reduce the frequency of heap allocations and deallocations, particularly at performance-critical times. Fewer allocations and deallocations means fewer occasions that trigger garbage collection. This also reduces risk of heap fragmentation.
- We can attempt to time garbage collection and heap expansion so that they happen at predictable and convenient times. This is a more difficult and less reliable

approach, but when used as part of an overall memory management strategy can reduce the impact of garbage collection.

Reducing the amount of garbage created

Let's examine a few techniques that will help us to reduce the amount of garbage generated by our code.

Caching

If our code repeatedly calls functions that lead to heap allocations and then discards the results, this creates unnecessary garbage. Instead, we should store references to these objects and reuse them. This technique is known as *caching*.

In the following example, the code causes a heap allocation each time it is called. This is because a new array is created.

```
void OnTriggerEnter(Collider other)
{
    Renderer[] allRenderers = FindObjectsOfType<Renderer
    ExampleFunction(allRenderers);
}
```

The following code causes only one heap allocation, as the array is created and populated once and then cached. The cached array can be reused again and again without generating more garbage.

```
private Renderer[] allRenderers;

void Start()
```

```
{
    allRenderers = FindObjectsOfType<Renderer>();
}


void OnTriggerEnter(Collider other)
{
    ExampleFunction(allRenderers);
}
```

Don’t allocate in functions that are called frequently

If we have to allocate heap memory in a MonoBehaviour, the worst place we can do it is in functions that run frequently. *Update()* and *LateUpdate()*, for example, are called once per frame, so if our code is generating garbage here it will quickly add up. We should consider caching references to objects in *Start()* or *Awake()* where possible, or ensuring that code that causes allocations only runs when it needs to.

Let’s look at a very simple example of moving code so that it only runs when things change. In the following code, a function that causes an allocation is called every time *Update()* is called, creating garbage frequently:

```
void Update()
{
    ExampleGarbageGeneratingFunction(transform.position
}
```

With a simple change, we now ensure that the allocating function is called only when the value of *transform.position.x* has changed. We are now only making heap allocations when necessary rather than in every single frame.

```
private float previousTransformPositionX;
```

```
void Update()
{
    float transformPositionX = transform.position.x;
    if (transformPositionX != previousTransformPosition)
    {
        ExampleGarbageGeneratingFunction(transformPosit
        previousTransformPositionX = transformPositionX
    }
}
```

Another technique for reducing garbage generated in *Update()* is to use a timer. This is suitable for when we have code that generates garbage that must run regularly, but not necessarily every frame.

In the following example code, the function that generates garbage runs once per frame:

```
void Update()

{

    ExampleGarbageGeneratingFunction();

}
```

In the following code, we use a timer to ensure that the function that generates garbage runs once per second.

```
private float timeSinceLastCalled;

private float delay = 1f;

void Update()
{
    timeSinceLastCalled += Time.deltaTime;
    if (timeSinceLastCalled > delay)
    {
```

```
            ExampleGarbageGeneratingFunction();
            timeSinceLastCalled = 0f;
        }
    }
```

Small changes like this, when made to code that runs frequently, can greatly reduce the amount of garbage generated.

Clearing collections

Creating new collections causes allocations on the heap. If we find that we're creating new collections more than once in our code, we should cache the reference to the collection and use *Clear()* to empty its contents instead of calling *new* repeatedly.

In the following example, a new heap allocation occurs every time *new *is used.

```
void Update()
{
    List myList = new List();
    PopulateList(myList);
}
```

In the following example, an allocation occurs only when the collection is created or when the collection must be resized behind the scenes. This greatly reduces the amount of garbage generated.

```
private List myList = new List();
void Update()
{
    myList.Clear();
    PopulateList(myList);
}
```

Object pooling

Even if we reduce allocations within our scripts, we may still have garbage collection problems if we create and destroy a lot of objects at runtime. *Object pooling (https://en.wikipedia.org/wiki/Object_pool_pattern)* is a technique that can reduce allocations and deallocations by reusing objects rather than repeatedly creating and destroying them. Object pooling is used widely in games and is most suitable for situations where we frequently spawn and destroy similar objects; for example, when shooting bullets from a gun.

A full guide to object pooling is beyond the scope of this article, but it is a really useful technique and one worth learning. This tutorial on object pooling on the Unity Learn site (/learn/tutorials/topics/scripting/object-pooling) is a great guide to implementing an object pooling system in Unity.

Common causes of unnecessary heap allocations

We understand that local, value-typed variables are allocated on the stack and that everything else is allocated on the heap. However, there are lots of situations where heap allocations may take us by surprise. Let's take a look at a few common causes of unnecessary heap allocations and consider how best to reduce these.

Strings

In C#, strings are reference types (https://msdn.microsoft.com/en-us/library/362314fe.aspx) not value types, even though they seem to hold the "value" of a string. This means that creating and discarding strings creates garbage. As strings are commonly used in a lot of code, this garbage can really add up.

Strings in C# are also *immutable*, which means that their value can't be changed after they are first created. Every time we manipulate a string (for example, by using the + operator to concatenate two strings), Unity creates a new string with the updated value and discards the old string. This creates garbage.

We can follow a few simple rules to keep garbage from strings to a minimum. Let's consider these rules, then look at an example of how to apply them.

- We should cut down on unnecessary string creation. If we are using the same string value more than once, we should create the string once and cache the value.
- We should cut down on unnecessary string manipulations. For example, if we have a Text component that is updated frequently and contains a concatenated string we could consider separating it into two Text components.
- If we have to build strings at runtime, we should use the StringBuilder class (https://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.110).aspx). The StringBuilder class is designed for building strings without allocations and will save on the amount of garbage we produce when concatenating complex strings.
- We should remove calls to *Debug.Log()* as soon as they are no longer needed for debugging purposes. Calls to *Debug.Log()* still execute in all builds of our game, even if they do not output to anything. A call to *Debug.Log()* creates and disposes of at least one string, so if our game contains many of these calls, the garbage can add up.

Let's examine an example of code that generates unnecessary garbage through inefficient use of strings. In the following code, we create a string for a score display in *Update()* by combining the string "TIME:" with the value of the float *timer*. This creates unnecessary garbage.

```
public Text timerText;
```

```
private float timer;

void Update()
{
    timer += Time.deltaTime;
    timerText.text = "TIME:" + timer.ToString();
}
```

In the following example, we have improved things considerably. We put the word "TIME:" in a separate Text component, and set its value in *Start()*. This means that in *Update()*, we no longer have to combine strings. This reduces the amount of garbage generated considerably.

```
public Text timerHeaderText;
public Text timerValueText;
private float timer;

void Start()
{
    timerHeaderText.text = "TIME:";
}

void Update()
{
    timerValueText.text = timer.toString();
}
```

Unity function calls

It's important to be aware that whenever we call code that we didn't write ourselves, whether that's in Unity itself or in a plugin, we could be generating garbage. Some Unity function calls create heap allocations, and so should be used with care to avoid generating unnecessary garbage.

There is no list of functions that we should avoid. Every function can be useful in some situations and less useful in others. As ever, it's best to profile our game carefully, identify

where garbage is being created and think carefully about how to handle it. In some cases, it may be wise to cache the results of the function; in other cases, it may be wise to call the function less frequently; in other cases, it may be best to refactor our code to use a different function. Having said that, let's look at a couple of common examples of Unity functions that cause heap allocations and consider how best to handle them.

Every time we access a Unity function that returns an array, a new array is created and passed to us as the return value. This behaviour isn't always obvious or expected, especially when the function is an accessor (https://msdn.microsoft.com/en-us/library/ms228503.aspx) (for example, *Mesh.normals (https://docs.unity3d.com/ScriptReference/Mesh-normals.html)*).

In the following code, a new array is created for each iteration of the loop.

```
void ExampleFunction()
{
    for (int i = 0; i < myMesh.normals.Length; i++)
    {
        Vector3 normal = myMesh.normals[i];
    }
}
```

It's easy to reduce allocations in cases like this: we can simply cache a reference to the array. When we do this, only one array is created and the amount of garbage created is reduced accordingly.

The following code demonstrates this. In this case, we call *Mesh.normals* before the loop runs and cache the reference so that only one array is created.

```
void ExampleFunction()
{
    Vector3[] meshNormals = myMesh.normals;
```

```
        for (int i = 0; i < meshNormals.Length; i++)
        {
            Vector3 normal = meshNormals[i];
        }
    }
```

Another unexpected cause of heap allocations can be found in the functions *GameObject.name* or *GameObject.tag*. Both of these are accessors that return new strings, which means that calling these functions will generate garbage. Caching the value may be useful, but in this case there is a related Unity function that we can use instead. To check a GameObject's tag against a value without generating garbage, we can use *GameObject.CompareTag() (https://docs.unity3d.com/ScriptReference/GameObject.CompareTag.html)*.

In the following example code, garbage is created by the call to *GameObject.tag*:

```
private string playerTag = "Player";

void OnTriggerEnter(Collider other)
{
    bool isPlayer = other.gameObject.tag == playerTag;
}
```

If we use *GameObject.CompareTag()*, this function no longer generates any garbage:

```
private string playerTag = "Player";

void OnTriggerEnter(Collider other)
{
    bool isPlayer = other.gameObject.CompareTag(playerTa
}
```

GameObject.CompareTag isn't unique; many Unity function calls have alternative versions that cause no heap allocations. For example, we could use *Input.GetTouch() (https://docs.unity3d.com/ScriptReference/Input.GetTouch.html)* and *Input.touchCount (https://docs.unity3d.com/ScriptReference/Input-touchCount.html)* in place of *Input.touches (https://docs.unity3d.com/ScriptReference/Input-touches.html)*, or *Physics.SphereCastNonAlloc() (https://docs.unity3d.com/ScriptReference/Physics.SphereCastNonAlloc.html)* in place of *Physics.SphereCastAll() (https://docs.unity3d.com/ScriptReference/Physics.SphereCastAll.html)*.

Boxing

Boxing (https://msdn.microsoft.com/en-GB/library/yz2be5wk.aspx) is the term for what happens when a value-typed variable is used in place of a reference-typed variable. Boxing usually occurs when we pass value-typed variables, such as ints or floats, to a function with object parameters such as *Object.Equals()*.

For example, the function *String.Format()* takes a string and an object parameter. When we pass it a string and an int, the int must be boxed. Therefore the following code contains an example of boxing:

```
void ExampleFunction()
{
    int cost = 5;
    string displayString = String.Format("Price: {0} go
}
```

Boxing creates garbage because of what happens behind the scenes. When a value-typed variable is boxed, Unity creates a temporary System.Object on the heap to wrap the value-typed

variable. A System.Object is a reference-typed variable, so when this temporary object is disposed of this creates garbage.

Boxing is an extremely common cause of unnecessary heap allocations. Even if we don't box variables directly in our code, we may be using plugins that cause boxing or it may be happening behind the scenes of other functions. It's best practice to avoid boxing wherever possible and to remove any function calls that lead to boxing.

Coroutines

Calling *StartCoroutine()* creates a small amount of garbage, because of the classes that Unity must create instances of to manage the coroutine. With that in mind, calls to *StartCoroutine()* should be limited while our game is interactive and performance is a concern. To reduce garbage created in this way, any coroutines that must run at performance-critical times should be started in advance and we should be particularly careful when using nested coroutines that may contain delayed calls to *StartCoroutine()*.

yield statements within coroutines do not create heap allocations in their own right; however, the values we pass with our *yield* statement could create unnecessary heap allocations. For example, the following code creates garbage:

```
yield return 0;
```

This code creates garbage because the int with a value of 0 is boxed. In this case, if we wish to simply wait for a frame without causing any heap allocations, the best way to do so is with this code:

```
yield return null;
```

Another common mistake with coroutines is to use *new* when yielding with the same value more than once. For example, the following code will create and then dispose of a WaitForSeconds object each time the loop iterates:

```
while (!isComplete)
{
    yield return new WaitForSeconds(1f);
}
```

If we cache and reuse the WaitForSeconds object, much less garbage is created. The following code shows this as an example:

```
WaitForSeconds delay = new WaitForSeconds(1f);

while (!isComplete)
{
    yield return delay;
}
```

If our code generates a lot of garbage due to coroutines, we may wish to consider refactoring our code to use something other than coroutines. Refactoring code is a complex subject and every project is unique, but there are a couple of common alternatives to coroutines that we may wish to bear in mind. For example, if we are using coroutines mainly to manage time, we may wish to simply keep track of time in an *Update()* function. If we are using coroutines mainly to control the order in which things happen in our game, we may wish to create some sort of messaging system to allow objects to communicate. There is no one size fits all approach to this, but it is useful to remember that there is often more than one way to achieve the same thing in code.

foreach loops

In versions of Unity prior to 5.5, a *foreach* loop iterating over anything other than an array generates garbage each time the loop terminates. This is due to boxing that happens behind the scenes. A System.Object is allocated on the heap when the loop begins and disposed of when the loop terminates. This problem was fixed in Unity 5.5.

For example, in versions of Unity prior to 5.5, the loop in the following code generates garbage:

```
void ExampleFunction(List listOfInts)
{
    foreach (int currentInt in listOfInts)
    {
            DoSomething(currentInt);
    }
}
```

If we are unable to upgrade our version of Unity, there is a simple solution to this problem. *for* and *while* loops do not cause boxing behind the scenes and therefore do not generate any garbage. We should favour their use when iterating over collections that are not arrays.

The loop in the following code will not generate garbage:

```
void ExampleFunction(List listOfInts)
{
    for (int i = 0; i < listOfInts.Count; i ++)
    {
        int currentInt = listOfInts[i];
        DoSomething(currentInt);
    }
}
```

Function references

References to functions, whether they refer to anonymous methods (https://msdn.microsoft.com/en-

us/library/0yw3tz5k.aspx) or named methods, are reference-typed variables in Unity. They will cause heap allocations. Converting an anonymous method to a closure (https://en.wikipedia.org/wiki/Closure_(computer_programming)) (where the anonymous method has access to the variables in scope at the time of its creation) significantly increases the memory usage and the number of heap allocations.

The precise details of how function references and closures allocate memory vary depending on platform and compiler settings, but if garbage collection is a concern then it's best to minimize the use of function references and closures during gameplay. This Unity best practice guide on performance (https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html) goes into greater technical detail on this topic.

LINQ and Regular Expressions

Both LINQ and Regular Expressions generate garbage due to boxing that occurs behind the scenes. It is best practice to avoid using these altogether where performance is a concern. Again, this Unity best practice guide on performance (https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity4-1.html) provides greater technical detail about this subject.

Structuring our code to minimize the impact of garbage collection

The way that our code is structured can impact garbage collection. Even if our code does not create heap allocations, it can add to the garbage collector's workload.

One way that our code can unnecessarily add to the garbage collector's workload is by requiring it to examine things that it

should not have to examine. Structs are value-typed variables, but if we have a struct that contains contains a reference-typed variable then the garbage collector must examine the whole struct. If we have a large array of these structs, then this can create a lot of additional work for the garbage collector.

In this example, the struct contains a string, which is reference-typed. The whole array of structs must now be examined by the garbage collector when it runs.

```
public struct ItemData
{
    public string name;
    public int cost;
    public Vector3 position;
}



private ItemData[] itemData;
```

In this example, we store the data in separate arrays. When the garbage collector runs, it need only examine the array of strings and can ignore the other arrays. This reduces the work that the garbage collector must do.

```
private string[] itemNames;
private int[] itemCosts;
private Vector3[] itemPositions;
```

Another way that our code can unnecessarily add to the garbage collector's workload is by having unnecessary object references. When the garbage collector searches for references to objects on the heap, it must examine every current object reference in our code. Having fewer object references in our code means that it has less work to do, even if we don't reduce the total number of objects on the heap.

In this example, we have a class that populates a dialog box. When the user has viewed the dialog, another dialog box is displayed. Our code contains a reference to the next instance of DialogData that should be displayed, meaning that the garbage collector must examine this reference as part of its operation:

```
public class DialogData
{
    private DialogData nextDialog;

    public DialogData GetNextDialog()
    {
        return nextDialog;
    }
}
```

Here, we have restructured the code so that it returns an identifier that is used to look up the next instance of DialogData, instead of the instance itself. This is not an object reference, so it does not add to the time taken by the garbage collector.

```
public class DialogData
{
    private int nextDialogID;

    public int GetNextDialogID()
    {
        return nextDialogID;
    }
}
```

On its own, this example is fairly trivial. However, if our game contains a great many objects that hold references to other objects, we can considerably reduce the complexity of the heap by restructuring our code in this fashion.

Timing garbage collection

Manually forcing garbage collection

Finally, we may wish to trigger garbage collection ourselves. If we know that heap memory has been allocated but is no longer used (for example, if our code has generated garbage when loading assets) and we know that a garbage collection freeze won't affect the player (for example, while the loading screen is still showing), we can request garbage collection using the following code:

```
System.GC.Collect();
```

This will force the garbage collector to run, freeing up the unused memory at a time that is convenient for us.

Conclusion

We've learned how garbage collection works in Unity, why it can cause performance problems and how to minimize its impact on our game. Using this knowledge and our profiling tools, we can fix performance problems related to garbage collection and structure our games so that they manage memory efficiently.

The links below provide further information on the topics covered in this article.

Further reading

Memory management and garbage collection in Unity

Unity Manual: Understanding Optimization in Unity (https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity.html)

Unity Manual: Understanding Automatic Memory Management (https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html)

Gamasutra: C# Memory Management for Unity Developers by Wendelin Reich (http://www.gamasutra.com/blogs/WendelinReich/20131109/203841/C_Memory_Management_for_Unity_Developers_part_1_of_3.php)

Gamasutra: C# memory and performance tips for Unity by Robert Zubek (http://www.gamasutra.com/blogs/RobertZubek/20150504/242572/C_memory_and_performance_tips_for_Unity.php)

Gamasutra: Reducing memory allocations to avoid Garbage Collection on Unity by Grhyll JDD (http://www.gamasutra.com/blogs/GrhyllJDD/20160119/263849/Reducing_memory_allocations_to_avoid_Garbage_Collection_on_Unity.php)

Gamasutra: Unity Garbage Collection Tips and Tricks by Megan Hughes (http://www.gamasutra.com/blogs/MeganHughes/20150727/249375/Unity_Garbage_Collection_Tips_and_Tricks.php)

MSDN: Boxing and Unboxing (C# Programming Guide) (https://msdn.microsoft.com/en-GB/library/yz2be5wk.aspx)

Unity Learn: Object Pooling Tutorial (https://unity3d.com/learn/tutorials/topics/scripting/object-pooling)

Wikipedia: Object Pool Pattern (https://en.wikipedia.org/wiki/Object_pool_pattern)

Best Practices for Using Strings in the .NET Framework (https://msdn.microsoft.com/en-us/library/dd465121(v=vs.110).aspx)

Partners

ot
oy
/o
ct
an
er
en
de
r)