Performance

Introduction

Godot follows a balanced performance philosophy. In the performance world, there are always tradeoffs, which consist of trading speed for usability and flexibility. Some practical examples of this are:

- Rendering large amounts of objects efficiently is easy, but when a large scene must be rendered, it can become inefficient. To solve this, visibility computation must be added to the rendering. This makes rendering less efficient, but at the same time, fewer objects are rendered. Therefore, the overall rendering efficiency is improved.
- Configuring the properties of every material for every object that needs to be rendered is also slow. To solve this, objects are sorted by material to reduce the costs. At the same time, sorting has a cost.
- In 3D physics, a similar situation happens. The best algorithms to handle large amounts of physics objects (such as SAP) are slow at insertion/removal of objects and raycasting. Algorithms that allow faster insertion and removal, as well as raycasting, will not be able to handle as many active objects.

And there are many more examples of this! Game engines strive to be general-purpose in nature. Balanced algorithms are always favored over algorithms that might be fast in some situations and slow in others, or algorithms that are fast but are more difficult to use.

Godot is not an exception to this. While it is designed to have backends swappable for different algorithms, the default backends prioritize balance and flexibility over performance.

With this clear, the aim of this tutorial section is to explain how to get the maximum performance out of Godot. While the tutorials can be read in any order, it is a good idea to start from General optimization tips.

Common

CPU

GPU

3D

Threads

General optimization tips

Introduction

In an ideal world, computers would run at infinite speed. The only limit to what we could achieve would be our imagination. However, in the real world, it's all too easy to produce software that will bring even the fastest computer to its knees.

Thus, designing games and other software is a compromise between what we would like to be possible, and what we can realistically achieve while maintaining good performance.

To achieve the best results, we have two approaches:

- Work faster.
- Work smarter.

And preferably, we will use a blend of the two.

Smoke and mirrors

Part of working smarter is recognizing that, in games, we can often get the player to believe they're in a world that is far more complex, interactive, and graphically exciting than it really is. A good programmer is a magician, and should strive to learn the tricks of the trade while trying to invent new ones.

The nature of slowness

To the outside observer, performance problems are often lumped together. But in reality, there are several different kinds of performance problems:

- A slow process that occurs every frame, leading to a continuously low frame rate.
- An intermittent process that causes "spikes" of slowness, leading to stalls.
- A slow process that occurs outside of normal gameplay, for instance, when loading a level.

Each of these are annoying to the user, but in different ways.

Measuring performance

Probably the most important tool for optimization is the ability to measure performance - to identify where bottlenecks are, and to measure the success of our attempts to speed them up.

There are several methods of measuring performance, including:

- Putting a start/stop timer around code of interest.
- Using the Godot profiler.
- Using external CPU profilers.
- Using external GPU profilers/debuggers such as NVIDIA Nsight Graphics [https://developer.nvidia.com/nsight-graphics], Radeon GPU Profiler [https://gpuopen.com/rgp/] or Intel Graphics Performance Analyzers [https://www.intel.com/content/www/us/en/developer/tools/graphics-performance-analyzers/overview.html].
- Checking the frame rate (with V-Sync disabled). Third-party utilities such as RivaTuner Statistics Server [https://www.guru3d.com/files-details/rtss-rivatuner-statistics-server-download.html] (Windows) or MangoHud [https://github.com/flightlessmango/MangoHud] (Linux) can also be useful here.
- Using an unofficial debug menu add-on [https://github.com/godot-extended-libraries/godot-debug-menu].

Be very aware that the relative performance of different areas can vary on different hardware. It's often a good idea to measure timings on more than one device. This is especially the case if you're targeting mobile devices.

Limitations

CPU profilers are often the go-to method for measuring performance. However, they don't always tell the whole story.

- Bottlenecks are often on the GPU, "as a result" of instructions given by the CPU.
- Spikes can occur in the operating system processes (outside of Godot) "as a result" of instructions used in Godot (for example, dynamic memory allocation).
- You may not always be able to profile specific devices like a mobile phone due to the initial setup required.
- You may have to solve performance problems that occur on hardware you don't have access to.

As a result of these limitations, you often need to use detective work to find out where bottlenecks are.

Detective work

Detective work is a crucial skill for developers (both in terms of performance, and also in terms of bug fixing). This can include hypothesis testing, and binary search.

Hypothesis testing

Say, for example, that you believe sprites are slowing down your game. You can test this hypothesis by:

- Measuring the performance when you add more sprites, or take some away.

This may lead to a further hypothesis: does the size of the sprite determine the performance drop?

- You can test this by keeping everything the same, but changing the sprite size, and measuring performance.

Binary search

If you know that frames are taking much longer than they should, but you're not sure where the bottleneck lies. You could begin by commenting out approximately half the routines that occur on a normal frame. Has the performance improved more or less than expected?

Once you know which of the two halves contains the bottleneck, you can repeat this process until you've pinned down the problematic area.

Profilers

Profilers allow you to time your program while running it. Profilers then provide results telling you what percentage of time was spent in different functions and areas, and how often functions were called.

This can be very useful both to identify bottlenecks and to measure the results of your improvements. Sometimes, attempts to improve performance can backfire and lead to slower performance. **Always use profiling and timing to guide your efforts.**

For more info about using Godot's built-in profiler, see The Profiler.

Principles

Donald Knuth [https://en.wikipedia.org/wiki/Donald_Knuth] said:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a*

> *strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

The messages are very important:

- Developer time is limited. Instead of blindly trying to speed up all aspects of a program, we should concentrate our efforts on the aspects that really matter.
- Efforts at optimization often end up with code that is harder to read and debug than non-optimized code. It is in our interests to limit this to areas that will really benefit.

Just because we *can* optimize a particular bit of code, it doesn't necessarily mean that we *should*. Knowing when and when not to optimize is a great skill to develop.

One misleading aspect of the quote is that people tend to focus on the subquote *"premature optimization is the root of all evil"*. While *premature* optimization is (by definition) undesirable, performant software is the result of performant design.

Performant design

The danger with encouraging people to ignore optimization until necessary, is that it conveniently ignores that the most important time to consider performance is at the design stage, before a key has even hit a keyboard. If the design or algorithms of a program are inefficient, then no amount of polishing the details later will make it run fast. It may run *faster*, but it will never run as fast as a program designed for performance.

This tends to be far more important in game or graphics programming than in general programming. A performant design, even without low-level optimization, will often run many times faster than a mediocre design with low-level optimization.

Incremental design

Of course, in practice, unless you have prior knowledge, you are unlikely to come up with the best design the first time. Instead, you'll often make a series of versions of a particular area of code, each taking a different approach to the problem, until you come to a satisfactory solution. It's important not to spend too much time on the details at this stage until you have finalized the overall design. Otherwise, much of your work will be thrown out.

It's difficult to give general guidelines for performant design because this is so dependent on the problem. One point worth mentioning though, on the CPU side, is that modern CPUs are nearly always limited by memory bandwidth. This has led to a resurgence in data-oriented design, which involves designing data structures and algorithms for *cache locality* of data and linear access, rather than jumping around in memory.

The optimization process

Assuming we have a reasonable design, and taking our lessons from Knuth, our first step in optimization should be to identify the biggest bottlenecks - the slowest functions, the low-hanging fruit.

Once we've successfully improved the speed of the slowest area, it may no longer be the bottleneck. So we should test/profile again and find the next bottleneck on which to focus.

The process is thus:

1. Profile / Identify bottleneck.
2. Optimize bottleneck.
3. Return to step 1.

Optimizing bottlenecks

Some profilers will even tell you which part of a function (which data accesses, calculations) are slowing things down.

As with design, you should concentrate your efforts first on making sure the algorithms and data structures are the best they can be. Data access should be local (to make best use of CPU cache), and it can often be better to use compact storage of data (again, always profile to test results). Often, you precalculate heavy computations ahead of time. This can be done by performing the computation when loading a level, by loading a file containing precalculated data or simply by storing the results of complex calculations into a script constant and reading its value.

Once algorithms and data are good, you can often make small changes in routines which improve performance. For instance, you can move some calculations outside of loops or transform nested `for` loops into non-nested loops. (This should be feasible if you know a 2D array's width or height in advance.)

Always retest your timing/bottlenecks after making each change. Some changes will increase speed, others may have a negative effect. Sometimes, a small positive effect will be outweighed by the negatives of more complex code, and you may choose to leave out that optimization.

Appendix

Bottleneck math

The proverb *"a chain is only as strong as its weakest link"* applies directly to performance optimization. If your project is spending 90% of the time in function `A`, then optimizing `A` can have a massive effect on performance.

```
A: 9 ms
Everything else: 1 ms
Total frame time: 10 ms
```

```
A: 1 ms
Everything else: 1ms
Total frame time: 2 ms
```

In this example, improving this bottleneck `A` by a factor of $9\times$ decreases overall frame time by $5\times$ while increasing frames per second by $5\times$.

However, if something else is running slowly and also bottlenecking your project, then the same improvement can lead to less dramatic gains:

```
A: 9 ms
Everything else: 50 ms
Total frame time: 59 ms
```

```
A: 1 ms
Everything else: 50 ms
Total frame time: 51 ms
```

In this example, even though we have hugely optimized function `A`, the actual gain in terms of frame rate is quite small.

In games, things become even more complicated because the CPU and GPU run independently of one another. Your total frame time is determined by the slower of the two.

```
CPU: 9 ms
GPU: 50 ms
Total frame time: 50 ms
```

```
CPU: 1 ms
GPU: 50 ms
Total frame time: 50 ms
```

In this example, we optimized the CPU hugely again, but the frame time didn't improve because we are GPU-bottlenecked.

Optimization using Servers

Engines like Godot provide increased ease of use thanks to their high level constructs and features. Most of them are accessed and used via the Scene System. Using nodes and resources simplifies project organization and asset management in complex games.

There are, of course, always drawbacks:

- There is an extra layer of complexity.
- Performance is lower than when using simple APIs directly.
- It is not possible to use multiple threads to control them.
- More memory is needed.

In many cases, this is not really a problem (Godot is very optimized, and most operations are handled with signals, so no polling is required). Still, sometimes it can be. For example, dealing with tens of thousands of instances for something that needs to be processed every frame can be a bottleneck.

This type of situation makes programmers regret they are using a game engine and wish they could go back to a more handcrafted, low level implementation of game code.

Still, Godot is designed to work around this problem.

See also

You can see how using low-level servers works in action using the Bullet Shower demo project [https://github.com/godotengine/godot-demo-projects/tree/master/2d/bullet_shower]

Servers

One of the most interesting design decisions for Godot is the fact that the whole scene system is *optional*. While it is not currently possible to compile it out, it can be completely bypassed.

At the core, Godot uses the concept of Servers. They are very low-level APIs to control rendering, physics, sound, etc. The scene system is built on top of them and uses them directly. The most common servers are:

- RenderingServer: handles everything related to graphics.
- PhysicsServer3D: handles everything related to 3D physics.
- PhysicsServer2D: handles everything related to 2D physics.
- AudioServer: handles everything related to audio.

Explore their APIs and you will realize that all the functions provided are low-level implementations of everything Godot allows you to do.

RIDs

The key to using servers is understanding Resource ID (RID) objects. These are opaque handles to the server implementation. They are allocated and freed manually. Almost every function in the servers requires RIDs to access the actual resource.

Most Godot nodes and resources contain these RIDs from the servers internally, and they can be obtained with different functions. In fact, anything that inherits Resource can be directly casted to an RID. Not all resources contain an RID, though: in such cases, the RID will be empty. The resource can then be passed to server APIs as an RID.

Warning

Resources are reference-counted (see RefCounted), and references to a resource's RID are *not* counted when determining whether the resource is still in use. Make sure to keep a reference to the resource outside the server, or else both it and its RID will be erased.

For nodes, there are many functions available:

- For CanvasItem, the CanvasItem.get_canvas_item() method will return the canvas item RID in the server.
- For CanvasLayer, the CanvasLayer.get_canvas() method will return the canvas RID in the server.
- For Viewport, the Viewport.get_viewport_rid() method will return the viewport RID in the server.
- For 3D, the World3D resource (obtainable in the Viewport and Node3D nodes) contains functions to get the *RenderingServer Scenario*, and the *PhysicsServer Space*. This allows creating 3D objects directly with the server API and using them.
- For 2D, the World2D resource (obtainable in the Viewport and CanvasItem nodes) contains functions to get the *RenderingServer Canvas*, and the *Physics2DServer Space*. This allows creating 2D objects directly with the server API and using them.
- The VisualInstance3D class, allows getting the scenario *instance* and *instance base* via the VisualInstance3D.get_instance() and VisualInstance3D.get_base() respectively.

Try exploring the nodes and resources you are familiar with and find the functions to obtain the server *RIDs*.

It is not advised to control RIDs from objects that already have a node associated. Instead, server functions should always be used for creating and controlling new ones and interacting with the existing ones.

Creating a sprite

This is an example of how to create a sprite from code and move it using the low-level CanvasItem API.

GDScript

```
extends Node2D


# RenderingServer expects references to be kept around.
var texture


func _ready():
    # Create a canvas item, child of this node.
    var ci_rid = RenderingServer.canvas_item_create()
    # Make this node the parent.
    RenderingServer.canvas_item_set_parent(ci_rid,
get_canvas_item())
    # Draw a texture on it.
    # Remember, keep this reference.
    texture = load("res://my_texture.png")
    # Add it, centered.
    RenderingServer.canvas_item_add_texture_rect(ci_rid,
Rect2(texture.get_size() / 2, texture.get_size()), texture)
    # Add the item, rotated 45 degrees and translated.
    var xform =
Transform2D().rotated(deg_to_rad(45)).translated(Vector2(20,
30))
    RenderingServer.canvas_item_set_transform(ci_rid, xform)
```

C#

```
public partial class MyNode2D : Node2D
{
    // RenderingServer expects references to be kept around.
    private Texture2D _texture;

    public override void _Ready()
    {
        // Create a canvas item, child of this node.
        Rid ciRid = RenderingServer.CanvasItemCreate();
        // Make this node the parent.
        RenderingServer.CanvasItemSetParent(ciRid,
GetCanvasItem());
        // Draw a texture on it.
        // Remember, keep this reference.
        _texture = ResourceLoader.Load<Texture2D>
("res://MyTexture.png");
        // Add it, centered.
        RenderingServer.CanvasItemAddTextureRect(ciRid, new
Rect2(_texture.GetSize() / 2, _texture.GetSize()),
_texture.GetRid());
        // Add the item, rotated 45 degrees and translated.
        Transform2D xform =
Transform2D.Identity.Rotated(Mathf.DegToRad(45)).Translated(n
ew Vector2(20, 30));
        RenderingServer.CanvasItemSetTransform(ciRid, xform);
    }
}
```

The Canvas Item API in the server allows you to add draw primitives to it. Once added, they can't be modified. The Item needs to be cleared and the primitives re-added (this is not the case for setting the transform, which can be done as many times as desired).

Primitives are cleared this way:

GDScript

```
RenderingServer.canvas_item_clear(ci_rid)
```

C#

```
RenderingServer.CanvasItemClear(ciRid);
```

Instantiating a Mesh into 3D space

The 3D APIs are different from the 2D ones, so the instantiation API must be used.

GDScript

```
extends Node3D


# RenderingServer expects references to be kept around.
var mesh


func _ready():
    # Create a visual instance (for 3D).
    var instance = RenderingServer.instance_create()
    # Set the scenario from the world, this ensures it
    # appears with the same objects as the scene.
    var scenario = get_world_3d().scenario
    RenderingServer.instance_set_scenario(instance, scenario)
    # Add a mesh to it.
    # Remember, keep the reference.
    mesh = load("res://mymesh.obj")
    RenderingServer.instance_set_base(instance, mesh)
    # Move the mesh around.
    var xform = Transform3D(Basis(), Vector3(20, 100, 0))
    RenderingServer.instance_set_transform(instance, xform)
```

C#

```
public partial class MyNode3D : Node3D
{
    // RenderingServer expects references to be kept around.
    private Mesh _mesh;

    public override void _Ready()
    {
        // Create a visual instance (for 3D).
        Rid instance = RenderingServer.InstanceCreate();
        // Set the scenario from the world, this ensures it
        // appears with the same objects as the scene.
        Rid scenario = GetWorld3D().Scenario;
        RenderingServer.InstanceSetScenario(instance,
scenario);
        // Add a mesh to it.
        // Remember, keep the reference.
        _mesh = ResourceLoader.Load<Mesh>
("res://MyMesh.obj");
        RenderingServer.InstanceSetBase(instance,
_mesh.GetRid());
        // Move the mesh around.
        Transform3D xform = new Transform3D(Basis.Identity,
new Vector3(20, 100, 0));
        RenderingServer.InstanceSetTransform(instance,
xform);
    }
}
```

Creating a 2D RigidBody and moving a sprite with it

This creates a RigidBody2D using the PhysicsServer2D API, and moves a CanvasItem when the body moves.

GDScript

```
# Physics2DServer expects references to be kept around.
var body
var shape


func _body_moved(state, index):
    # Created your own canvas item, use it here.
    RenderingServer.canvas_item_set_transform(canvas_item,
state.transform)


func _ready():
    # Create the body.
    body = Physics2DServer.body_create()
    Physics2DServer.body_set_mode(body,
Physics2DServer.BODY_MODE_RIGID)
    # Add a shape.
    shape = Physics2DServer.rectangle_shape_create()
    # Set rectangle extents.
    Physics2DServer.shape_set_data(shape, Vector2(10, 10))
    # Make sure to keep the shape reference!
    Physics2DServer.body_add_shape(body, shape)
    # Set space, so it collides in the same space as current
scene.
    Physics2DServer.body_set_space(body,
get_world_2d().space)
    # Move initial position.
    Physics2DServer.body_set_state(body,
Physics2DServer.BODY_STATE_TRANSFORM, Transform2D(0,
Vector2(10, 20)))
    # Add the transform callback, when body moves
    # The last parameter is optional, can be used as index
    # if you have many bodies and a single callback.
    Physics2DServer.body_set_force_integration_callback(body,
self, "_body_moved", 0)
```

The 3D version should be very similar, as 2D and 3D physics servers are identical (using RigidBody3D and PhysicsServer3D respectively).

Getting data from the servers

Try to **never** request any information from `RenderingServer`, `PhysicsServer2D` or `PhysicsServer3D` by calling functions unless you know what you are doing. These servers will often run asynchronously for performance and calling any function that returns a value will stall them and force them to process anything pending until the function is actually called. This will severely decrease performance if you call them every frame (and it won't be obvious why).

Because of this, most APIs in such servers are designed so it's not even possible to request information back, until it's actual data that can be saved.

CPU optimization

Measuring performance

We have to know where the "bottlenecks" are to know how to speed up our program. Bottlenecks are the slowest parts of the program that limit the rate that everything can progress. Focusing on bottlenecks allows us to concentrate our efforts on optimizing the areas which will give us the greatest speed improvement, instead of spending a lot of time optimizing functions that will lead to small performance improvements.

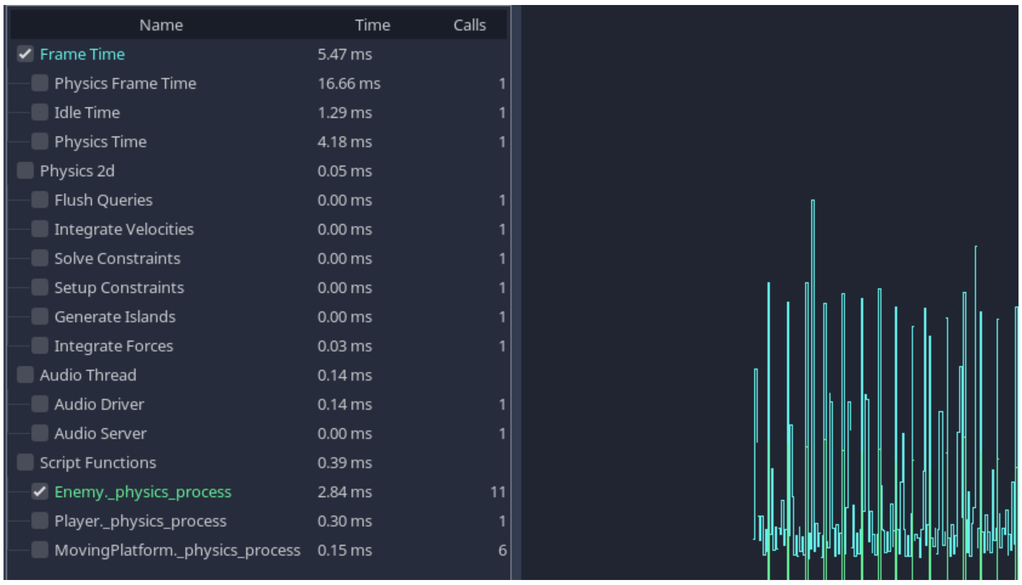
For the CPU, the easiest way to identify bottlenecks is to use a profiler.

CPU profilers

Profilers run alongside your program and take timing measurements to work out what proportion of time is spent in each function.

The Godot IDE conveniently has a built-in profiler. It does not run every time you start your project: it must be manually started and stopped. This is because, like most profilers, recording these timing measurements can slow down your project significantly.

After profiling, you can look back at the results for a frame.



Name	Time	Calls
✓ Frame Time	5.47 ms	
Physics Frame Time	16.66 ms	1
Idle Time	1.29 ms	1
Physics Time	4.18 ms	1
Physics 2d	0.05 ms	
Flush Queries	0.00 ms	1
Integrate Velocities	0.00 ms	1
Solve Constraints	0.00 ms	1
Setup Constraints	0.00 ms	1
Generate Islands	0.00 ms	1
Integrate Forces	0.03 ms	1
Audio Thread	0.14 ms	
Audio Driver	0.14 ms	1
Audio Server	0.00 ms	1
Script Functions	0.39 ms	
✓ Enemy._physics_process	2.84 ms	11
Player._physics_process	0.30 ms	1
MovingPlatform._physics_process	0.15 ms	6

Results of a profile of one of the demo projects.

Note

We can see the cost of built-in processes such as physics and audio, as well as seeing the cost of our own scripting functions at the bottom.

Time spent waiting for various built-in servers may not be counted in the profilers. This is a known bug.

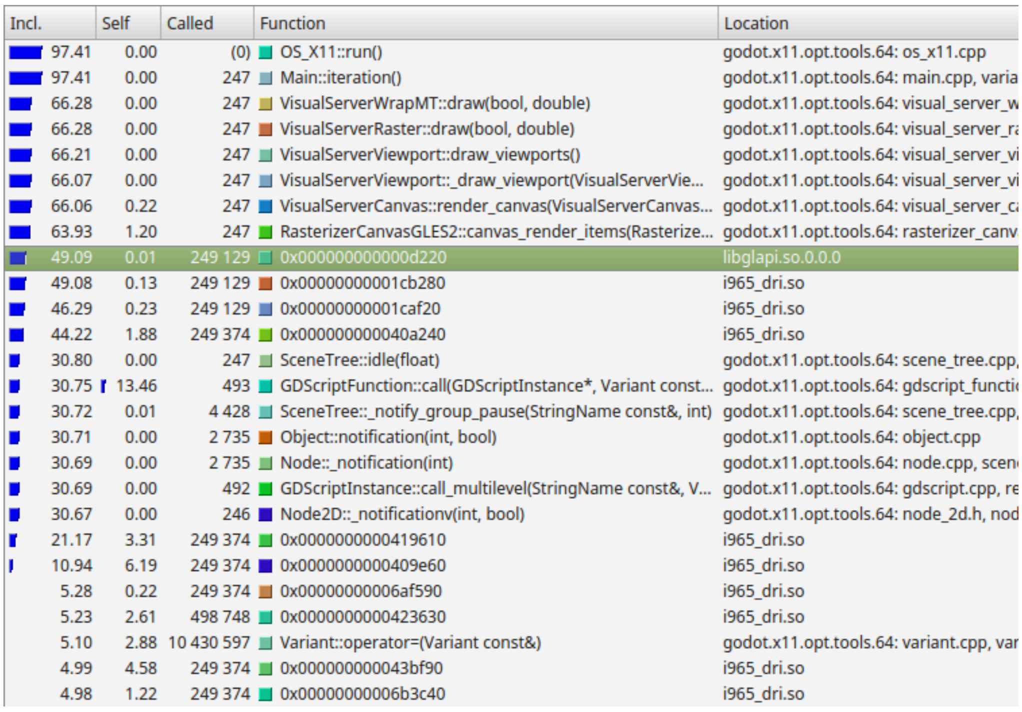
When a project is running slowly, you will often see an obvious function or process taking a lot more time than others. This is your primary bottleneck, and you can usually increase speed by optimizing this area.

For more info about using Godot's built-in profiler, see Debugger panel.

External profilers

Although the Godot IDE profiler is very convenient and useful, sometimes you need more power, and the ability to profile the Godot engine source code itself.

You can use a number of third-party C++ profilers to do this.



Incl.	Self	Called	Function	Location
97.41	0.00	(0)	OS_X11::run()	godot.x11.opt.tools.64: os_x11.cpp
97.41	0.00	247	Main::iteration()	godot.x11.opt.tools.64: main.cpp, varia
66.28	0.00	247	VisualServerWrapMT::draw(bool, double)	godot.x11.opt.tools.64: visual_server_w
66.28	0.00	247	VisualServerRaster::draw(bool, double)	godot.x11.opt.tools.64: visual_server_ra
66.21	0.00	247	VisualServerViewport::draw_viewports()	godot.x11.opt.tools.64: visual_server_vi
66.07	0.00	247	VisualServerViewport::_draw_viewport(VisualServerVie...	godot.x11.opt.tools.64: visual_server_vi
66.06	0.22	247	VisualServerCanvas::render_canvas(VisualServerCanvas...	godot.x11.opt.tools.64: visual_server_ca
63.93	1.20	247	RasterizerCanvasGLES2::canvas_render_items(Rasterize...	godot.x11.opt.tools.64: rasterizer_canv
49.09	0.01	249 129	0x000000000000d220	libglapi.so.0.0.0
49.08	0.13	249 129	0x00000000001cb280	i965_dri.so
46.29	0.23	249 129	0x00000000001caf20	i965_dri.so
44.22	1.88	249 374	0x000000000040a240	i965_dri.so
30.80	0.00	247	SceneTree::idle(float)	godot.x11.opt.tools.64: scene_tree.cpp,
30.75	13.46	493	GDScriptFunction::call(GDScriptInstance*, Variant const...	godot.x11.opt.tools.64: gdscript_functi
30.72	0.01	4 428	SceneTree::_notify_group_pause(StringName const&, int)	godot.x11.opt.tools.64: scene_tree.cpp,
30.71	0.00	2 735	Object::notification(int, bool)	godot.x11.opt.tools.64: object.cpp
30.69	0.00	2 735	Node::_notification(int)	godot.x11.opt.tools.64: node.cpp, scen
30.69	0.00	492	GDScriptInstance::call_multilevel(StringName const&, V...	godot.x11.opt.tools.64: gdscript.cpp, re
30.67	0.00	246	Node2D::_notificationv(int, bool)	godot.x11.opt.tools.64: node_2d.h, nod
21.17	3.31	249 374	0x0000000000419610	i965_dri.so
10.94	6.19	249 374	0x0000000000409e60	i965_dri.so
5.28	0.22	249 374	0x00000000006af590	i965_dri.so
5.23	2.61	498 748	0x0000000000423630	i965_dri.so
5.10	2.88	10 430 597	Variant::operator=(Variant const&)	godot.x11.opt.tools.64: variant.cpp, var
4.99	4.58	249 374	0x000000000043bf90	i965_dri.so
4.98	1.22	249 374	0x00000000006b3c40	i965_dri.so

Example results from Callgrind, which is part of Valgrind.

From the left, Callgrind is listing the percentage of time within a function and its children (Inclusive), the percentage of time spent within the function itself, excluding child functions (Self), the number of times the function is called, the function name, and the file or module.

In this example, we can see nearly all time is spent under the `Main::iteration()` function. This is the master function in the Godot source code that is called repeatedly. It causes frames to be drawn, physics ticks to be simulated, and nodes and scripts to be updated. A large proportion of the time is spent in the functions to render a canvas (66%), because this example uses a 2D benchmark. Below this, we see that almost 50% of the time is spent outside Godot code in `libglapi` and `i965_dri` (the graphics driver). This tells us the a large proportion of CPU time is being spent in the graphics driver.

This is actually an excellent example because, in an ideal world, only a very small proportion of time would be spent in the graphics driver. This is an indication that there is a problem with too much communication and work being done in the graphics API. This specific profiling led to the development of 2D batching, which greatly speeds up 2D rendering by reducing bottlenecks in this area.

Manually timing functions

Another handy technique, especially once you have identified the bottleneck using a profiler, is to manually time the function or area under test. The specifics vary depending on the language, but in GDScript, you would do the following:

```
var time_start = Time.get_ticks_usec()

# Your function you want to time
update_enemies()

var time_end = Time.get_ticks_usec()
print("update_enemies() took %d microseconds" % time_end - time_start)
```

When manually timing functions, it is usually a good idea to run the function many times (1,000 or more times), instead of just once (unless it is a very slow function). The reason for doing this is that timers often have limited accuracy. Moreover, CPUs will schedule processes in a haphazard manner. Therefore, an average over a series of runs is more accurate than a single measurement.

As you attempt to optimize functions, be sure to either repeatedly profile or time them as you go. This will give you crucial feedback as to whether the optimization is working (or not).

Caches

CPU caches are something else to be particularly aware of, especially when comparing timing results of two different versions of a function. The results can be highly dependent on whether the data is in the CPU cache or not. CPUs don't load data directly from the system RAM, even though it's huge in comparison to the CPU cache (several gigabytes instead of a few megabytes). This is because system RAM is very slow to access. Instead, CPUs load data from a smaller, faster bank of memory called cache. Loading data from cache is very fast, but every time you try and load a memory address that is not stored in cache, the cache must make a trip to main memory and slowly load in some data. This delay can result in the CPU sitting around idle for a long time, and is referred to as a "cache miss".

This means that the first time you run a function, it may run slowly because the data is not in the CPU cache. The second and later times, it may run much faster because the data is in the cache. Due to this, always use averages when timing, and be aware of the effects of cache.

Understanding caching is also crucial to CPU optimization. If you have an algorithm (routine) that loads small bits of data from randomly spread out areas of main memory, this can result in a lot of cache misses, a lot of the time, the CPU will be waiting around for data instead of doing any work. Instead, if you can make your data accesses localised, or even better, access memory in a linear fashion (like a continuous list), then the cache will work optimally and the CPU will be able to work as fast as possible.

Godot usually takes care of such low-level details for you. For example, the Server APIs make sure data is optimized for caching already for things like rendering and physics. Still, you should be especially aware of caching when writing GDExtensions.

Languages

Godot supports a number of different languages, and it is worth bearing in mind that there are trade-offs involved. Some languages are designed for ease of use at the cost of speed, and others are faster but more difficult to work with.

Built-in engine functions run at the same speed regardless of the scripting language you choose. If your project is making a lot of calculations in its own code, consider moving those calculations to a faster language.

GDScript

GDScript is designed to be easy to use and iterate, and is ideal for making many types of games. However, in this language, ease of use is considered more important than performance. If you need to make heavy calculations, consider moving some of your project to one of the other languages.

C#

C# is popular and has first-class support in Godot. It offers a good compromise between speed and ease of use. Beware of possible garbage collection pauses and leaks that can occur during gameplay, though. A common approach to workaround issues with garbage collection is to use *object pooling*, which is outside the scope of this guide.

Other languages

Third parties provide support for several other languages, including Rust [https://github.com/godot-rust/gdext].

C++

Godot is written in C++. Using C++ will usually result in the fastest code. However, on a practical level, it is the most difficult to deploy to end users' machines on different platforms. Options for using C++ include GDExtensions and custom modules.

Threads

Consider using threads when making a lot of calculations that can run in parallel to each other. Modern CPUs have multiple cores, each one capable of doing a limited amount of work. By spreading work over multiple threads, you can move further towards peak CPU efficiency.

The disadvantage of threads is that you have to be incredibly careful. As each CPU core operates independently, they can end up trying to access the same memory at the same time. One thread can be reading to a variable while another is writing: this is called a *race condition*. Before you use threads, make sure you understand the dangers and how to try and prevent these race conditions. Threads can make debugging considerably more difficult.

For more information on threads, see Using multiple threads.

SceneTree

Although Nodes are an incredibly powerful and versatile concept, be aware that every node has a cost. Built-in functions such as `_process()` and `_physics_process()` propagate through the tree. This housekeeping can reduce performance when you have a very large numbers of nodes (how many exactly depends on the target platform and can range from thousands to tens of thousands so ensure that you profile performance on all target platforms during development).

Each node is handled individually in the Godot renderer. Therefore, a smaller number of nodes with more in each can lead to better performance.

One quirk of the SceneTree is that you can sometimes get much better performance by removing nodes from the SceneTree, rather than by pausing or hiding them. You don't have to delete a detached node. You can for example, keep a reference to a node, detach it from the scene tree using Node.remove_child(node), then reattach it later using Node.add_child(node). This can be very useful for adding and removing areas from a game, for example.

You can avoid the SceneTree altogether by using Server APIs. For more information, see Optimization using Servers.

Physics

In some situations, physics can end up becoming a bottleneck. This is particularly the case with complex worlds and large numbers of physics objects.

Here are some techniques to speed up physics:

- Try using simplified versions of your rendered geometry for collision shapes. Often, this won't be noticeable for end users, but can greatly increase performance.
- Try removing objects from physics when they are out of view / outside the current area, or reusing physics objects (maybe you allow 8 monsters per area, for example, and reuse these).

Another crucial aspect to physics is the physics tick rate. In some games, you can greatly reduce the tick rate, and instead of for example, updating physics 60 times per second, you may update them only 30 or even 20 times per second. This can greatly reduce the CPU load.

The downside of changing physics tick rate is you can get jerky movement or jitter when the physics update rate does not match the frames per second rendered. Also, decreasing the physics tick rate will increase input lag. It's recommended to stick to the default physics tick rate (60 Hz) in most games that feature real-time player movement.

The solution to jitter is to use *fixed timestep interpolation*, which involves smoothing the rendered positions and rotations over multiple frames to match the physics. You can either implement this yourself or use a third-party addon [https://github.com/lawnjelly/smoothing-addon]. Performance-wise, interpolation is a very cheap operation compared to running a physics tick. It's orders of magnitude faster, so this can be a significant performance win while also reducing jitter.

GPU optimization

Introduction

The demand for new graphics features and progress almost guarantees that you will encounter graphics bottlenecks. Some of these can be on the CPU side, for instance in calculations inside the Godot engine to prepare objects for rendering. Bottlenecks can also occur on the CPU in the graphics driver, which sorts instructions to pass to the GPU, and in the transfer of these instructions. And finally, bottlenecks also occur on the GPU itself.

Where bottlenecks occur in rendering is highly hardware-specific. Mobile GPUs in particular may struggle with scenes that run easily on desktop.

Understanding and investigating GPU bottlenecks is slightly different to the situation on the CPU. This is because, often, you can only change performance indirectly by changing the instructions you give to the GPU. Also, it may be more difficult to take measurements. In many cases, the only way of measuring performance is by examining changes in the time spent rendering each frame.

Draw calls, state changes, and APIs

Note

The following section is not relevant to end-users, but is useful to provide background information that is relevant in later sections.

Godot sends instructions to the GPU via a graphics API (Vulkan, OpenGL, OpenGL ES or WebGL). The communication and driver activity involved can be quite costly, especially in OpenGL, OpenGL ES and WebGL. If we can provide these instructions in a way that is preferred by the driver and GPU, we can greatly increase performance.

Nearly every API command in OpenGL requires a certain amount of validation to make sure the GPU is in the correct state. Even seemingly simple commands can lead to a flurry of behind-the-scenes housekeeping. Therefore, the goal is to reduce these instructions to a bare minimum and group together similar objects as much as possible so they can be rendered together, or with the minimum number of these expensive state changes.

2D batching

In 2D, the costs of treating each item individually can be prohibitively high - there can easily be thousands of them on the screen. This is why 2D *batching* is used with OpenGL-based rendering methods. Multiple similar items are grouped together and rendered in a batch, via a single draw call, rather than making a separate draw call for each item. In addition, this means state changes, material and texture changes can be kept to a minimum.

Vulkan-based rendering methods do not use 2D batching yet. Since draw calls are much cheaper with Vulkan compared to OpenGL, there is less of a need to have 2D batching (although it can still be beneficial in some cases).

3D batching

In 3D, we still aim to minimize draw calls and state changes. However, it can be more difficult to batch together several objects into a single draw call. 3D meshes tend to comprise hundreds or thousands of triangles, and combining large meshes in real-time is prohibitively expensive. The costs of joining them quickly exceeds any benefits as the number of triangles grows per mesh. A much better alternative is to **join meshes ahead of time** (static meshes in relation to each other). This can be done by artists, or programmatically within Godot using an add-on.

There is also a cost to batching together objects in 3D. Several objects rendered as one cannot be individually culled. An entire city that is off-screen will still be rendered if it is joined to a single blade of grass that is on screen. Thus, you should always take objects' location and culling into account when attempting to batch 3D objects together. Despite this, the benefits of joining static objects often outweigh other considerations, especially for large numbers of distant or low-poly objects.

For more information on 3D specific optimizations, see Optimizing 3D performance.

Reuse shaders and materials

The Godot renderer is a little different to what is out there. It's designed to minimize GPU state changes as much as possible. StandardMaterial3D does a good job at reusing materials that need similar shaders. If custom shaders are used, make sure to reuse them as much as possible. Godot's priorities are:

- **Reusing Materials:** The fewer different materials in the scene, the faster the rendering will be. If a scene has a huge amount of objects (in the hundreds or thousands), try reusing the materials. In the worst case, use atlases to decrease the amount of texture changes.
- **Reusing Shaders:** If materials can't be reused, at least try to reuse shaders. Note: shaders are automatically reused between StandardMaterial3Ds that share the same configuration (features that are enabled or disabled with a check box) even if they have different parameters.

If a scene has, for example, 20,000 objects with 20,000 different materials each, rendering will be slow. If the same scene has 20,000 objects, but only uses 100 materials, rendering will be much faster.

Pixel cost versus vertex cost

You may have heard that the lower the number of polygons in a model, the faster it will be rendered. This is *really* relative and depends on many factors.

On a modern PC and console, vertex cost is low. GPUs originally only rendered triangles. This meant that every frame:

1. All vertices had to be transformed by the CPU (including clipping).
2. All vertices had to be sent to the GPU memory from the main RAM.

Nowadays, all this is handled inside the GPU, greatly increasing performance. 3D artists usually have the wrong feeling about polycount performance because 3D modeling software (such as Blender, 3ds Max, etc.) need to keep geometry in CPU memory for it to be edited, reducing actual performance. Game engines rely on the GPU more, so they can render many triangles much more efficiently.

On mobile devices, the story is different. PC and console GPUs are brute-force monsters that can pull as much electricity as they need from the power grid. Mobile GPUs are limited to a tiny battery, so they need to be a lot more power efficient.

To be more efficient, mobile GPUs attempt to avoid *overdraw*. Overdraw occurs when the same pixel on the screen is being rendered more than once. Imagine a town with several buildings. GPUs don't know what is visible and what is hidden until they draw it. For example, a house might be drawn and then another house in front of it (which means rendering happened twice for the same pixel). PC GPUs normally don't care much about this and just throw more pixel processors to the hardware to increase performance (which also increases power consumption).

Using more power is not an option on mobile so mobile devices use a technique called *tile-based rendering* which divides the screen into a grid. Each cell keeps the list of triangles drawn to it and sorts them by depth to minimize *overdraw*. This technique improves performance and reduces power consumption, but takes a toll on vertex performance. As a result, fewer vertices and triangles can be processed for drawing.

Additionally, tile-based rendering struggles when there are small objects with a lot of geometry within a small portion of the screen. This forces mobile GPUs to put a lot of strain on a single screen tile, which considerably decreases performance as all the other cells must wait for it to complete before displaying the frame.

To summarize, don't worry about vertex count on mobile, but **avoid concentration of vertices in small parts of the screen**. If a character, NPC, vehicle, etc. is far away (which means it looks tiny), use a smaller level of detail (LOD) model. Even on desktop GPUs, it's preferable to avoid having triangles smaller than the size of a pixel on screen.

Pay attention to the additional vertex processing required when using:

- Skinning (skeletal animation)
- Morphs (shape keys)

Pixel/fragment shaders and fill rate

In contrast to vertex processing, the costs of fragment (per-pixel) shading have increased dramatically over the years. Screen resolutions have increased: the area of a 4K screen is $8,294,400$ pixels, versus $307,200$ for an old 640×480 VGA screen. That is 27 times the area! Also, the complexity of fragment shaders has exploded. Physically-based rendering requires complex calculations for each fragment.

You can test whether a project is fill rate-limited quite easily. Turn off V-Sync to prevent capping the frames per second, then compare the frames per second when running with a large window, to running with a very small window. You may also benefit from similarly reducing your shadow map size if using shadows. Usually, you will find the FPS increases quite a bit using a small window, which indicates you are to some extent fill rate-limited. On the other hand, if there is little to no increase in FPS, then your bottleneck lies elsewhere.

You can increase performance in a fill rate-limited project by reducing the amount of work the GPU has to do. You can do this by simplifying the shader (perhaps turn off expensive options if you are using a StandardMaterial3D), or reducing the number and size of textures used. Also, when using non-unshaded particles, consider forcing vertex shading in their material to decrease the shading cost.

See also

On supported hardware, Variable rate shading can be used to reduce shading processing costs without impacting the sharpness of edges on the final image.

When targeting mobile devices, consider using the simplest possible shaders you can reasonably afford to use.

Reading textures

The other factor in fragment shaders is the cost of reading textures. Reading textures is an expensive operation, especially when reading from several textures in a single fragment shader. Also, consider that filtering may slow it down further (trilinear filtering between mipmaps, and averaging). Reading textures is also expensive in terms of power usage, which is a big issue on mobiles.

If you use third-party shaders or write your own shaders, try to use algorithms that require as few texture reads as possible.

Texture compression

By default, Godot compresses textures of 3D models when imported using video RAM (VRAM) compression. Video RAM compression isn't as efficient in size as PNG or JPG when stored, but increases performance enormously when drawing large enough textures.

This is because the main goal of texture compression is bandwidth reduction between memory and the GPU.

In 3D, the shapes of objects depend more on the geometry than the texture, so compression is generally not noticeable. In 2D, compression depends more on shapes inside the textures, so the artifacts resulting from 2D compression are more noticeable.

As a warning, most Android devices do not support texture compression of textures with transparency (only opaque), so keep this in mind.

Note

Even in 3D, "pixel art" textures should have VRAM compression disabled as it will negatively affect their appearance, without improving performance significantly due to their low resolution.

Post-processing and shadows

Post-processing effects and shadows can also be expensive in terms of fragment shading activity. Always test the impact of these on different hardware.

Reducing the size of shadowmaps can increase performance, both in terms of writing and reading the shadowmaps. On top of that, the best way to improve performance of shadows is to turn shadows off for as many lights and objects as possible. Smaller or distant OmniLights/SpotLights can often have their shadows disabled with only a small visual impact.

Transparency and blending

Transparent objects present particular problems for rendering efficiency. Opaque objects (especially in 3D) can be essentially rendered in any order and the Z-buffer will ensure that only the front most objects get shaded. Transparent or blended objects are different. In most cases, they cannot rely on the Z-buffer and must be rendered in "painter's order" (i.e. from back to front) to look correct.

Transparent objects are also particularly bad for fill rate, because every item has to be drawn even if other transparent objects will be drawn on top later on.

Opaque objects don't have to do this. They can usually take advantage of the Z-buffer by writing to the Z-buffer only first, then only performing the fragment shader on the "winning" fragment, the object that is at the front at a particular pixel.

Transparency is particularly expensive where multiple transparent objects overlap. It is usually better to use transparent areas as small as possible to minimize these fill rate requirements, especially on mobile, where fill rate is very expensive. Indeed, in many situations, rendering more complex opaque geometry can end up being faster than using transparency to "cheat".

Multi-platform advice

If you are aiming to release on multiple platforms, test *early* and test *often* on all your platforms, especially mobile. Developing a game on desktop but attempting to port it to mobile at the last minute is a recipe for disaster.

In general, you should design your game for the lowest common denominator, then add optional enhancements for more powerful platforms. For example, you may want to use the Compatibility rendering method for both desktop and mobile platforms where you target both.

Mobile/tiled renderers

As described above, GPUs on mobile devices work in dramatically different ways from GPUs on desktop. Most mobile devices use tile renderers. Tile renderers split up the screen into regular-sized tiles that fit into super fast cache memory, which reduces the number of read/write operations to the main memory.

There are some downsides though. Tiled rendering can make certain techniques much more complicated and expensive to perform. Tiles that rely on the results of rendering in different tiles or on the results of earlier operations being preserved can be very slow. Be very careful to test the performance of shaders, viewport textures and post processing.

Optimization using MultiMeshes

For large amount of instances (in the thousands), that need to be constantly processed (and certain amount of control needs to be retained), using servers directly is the recommended optimization.

When the amount of objects reach the hundreds of thousands or millions, none of these approaches are efficient anymore. Still, depending on the requirements, there is one more optimization possible.

MultiMeshes

A MultiMesh is a single draw primitive that can draw up to millions of objects in one go. It's extremely efficient because it uses the GPU hardware to do this (in OpenGL ES 2.0, it's less efficient because there is no hardware support for it, though).

The only drawback is that there is no *screen* or *frustum* culling possible for individual instances. This means, that millions of objects will be *always* or *never* drawn, depending on the visibility of the whole MultiMesh. It is possible to provide a custom visibility rect for them, but it will always be *all-or-none* visibility.

If the objects are simple enough (just a couple of vertices), this is generally not much of a problem as most modern GPUs are optimized for this use case. A workaround is to create several MultiMeshes for different areas of the world.

It is also possible to execute some logic inside the vertex shader (using the `INSTANCE_ID` or `INSTANCE_CUSTOM` built-in constants). For an example of animating thousands of objects in a MultiMesh, see the Animating thousands of fish tutorial. Information to the shader can be provided via textures (there are floating-point Image formats which are ideal for this).

Another alternative is to use a GDExtension and C++, which should be extremely efficient (it's possible to set the entire state for all objects using linear memory via the RenderingServer.multimesh_set_buffer() function). This way, the array can be created with multiple threads, then set in one call, providing high cache efficiency.

Finally, it's not required to have all MultiMesh instances visible. The amount of visible ones can be controlled with the MultiMesh.visible_instance_count property. The typical workflow is to allocate the maximum amount of instances that will be used, then change the amount visible depending on how many are currently needed.

Multimesh example

Here is an example of using a MultiMesh from code. Languages other than GDScript may be more efficient for millions of objects, but for a few thousands, GDScript should be fine.

GDScript

```
extends MultiMeshInstance3D


func _ready():
    # Create the multimesh.
    multimesh = MultiMesh.new()
    # Set the format first.
    multimesh.transform_format = MultiMesh.TRANSFORM_3D
    # Then resize (otherwise, changing the format is not
allowed).
    multimesh.instance_count = 10000
    # Maybe not all of them should be visible at first.
    multimesh.visible_instance_count = 1000

    # Set the transform of the instances.
    for i in multimesh.visible_instance_count:
        multimesh.set_instance_transform(i,
Transform3D(Basis(), Vector3(i * 20, 0, 0)))
```

C#

```
using Godot;

public partial class MyMultiMeshInstance3D :
MultiMeshInstance3D
{
    public override void _Ready()
    {
        // Create the multimesh.
        Multimesh = new MultiMesh();
        // Set the format first.
        Multimesh.TransformFormat =
MultiMesh.TransformFormatEnum.Transform3D;
        // Then resize (otherwise, changing the format is not
allowed)
        Multimesh.InstanceCount = 1000;
        // Maybe not all of them should be visible at first.
        Multimesh.VisibleInstanceCount = 1000;

        // Set the transform of the instances.
        for (int i = 0; i < Multimesh.VisibleInstanceCount;
i++)
        {
            Multimesh.SetInstanceTransform(i, new
Transform3D(Basis.Identity, new Vector3(i * 20, 0, 0)));
        }
    }
}
```

Optimizing 3D performance

Culling

Godot will automatically perform view frustum culling in order to prevent rendering objects that are outside the viewport. This works well for games that take place in a small area, however things can quickly become problematic in larger levels.

Occlusion culling

Walking around a town for example, you may only be able to see a few buildings in the street you are in, as well as the sky and a few birds flying overhead. As far as a naive renderer is concerned however, you can still see the entire town. It won't just render the buildings in front of you, it will render the street behind that, with the people on that street, the buildings behind that. You quickly end up in situations where you are attempting to render $10\times$ or $100\times$ more than what is visible.

Things aren't quite as bad as they seem, because the Z-buffer usually allows the GPU to only fully shade the objects that are at the front. This is called *depth prepass* and is enabled by default in Godot when using the Forward+ or Compatibility rendering methods. However, unneeded objects are still reducing performance.

One way we can potentially reduce the amount to be rendered is to **take advantage of occlusion**. Godot 4.0 and later offers a new approach to occlusion culling using occluder nodes. See Occlusion culling for instructions on setting up occlusion culling in your scene.

Note

In some cases, you may have to adapt your level design to add more occlusion opportunities. For example, you may have to add more walls to prevent the player from seeing too far away, which would decrease performance due to the lost opportunities for occlusion culling.

Transparent objects

Godot sorts objects by Material and Shader to improve performance. This, however, can not be done with transparent objects. Transparent objects are rendered from back to front to make blending with what is behind work. As a result, **try to use as few transparent objects as possible**. If an object has a small section with transparency, try to make that section a separate surface with its own material.

For more information, see the GPU optimizations doc.

Level of detail (LOD)

In some situations, particularly at a distance, it can be a good idea to **replace complex geometry with simpler versions**. The end user will probably not be able to see much difference. Consider looking at a large number of trees in the far distance. There are several strategies for replacing models at varying distance. You could use lower poly models, or use transparency to simulate more complex geometry.

Godot 4 offers several ways to control level of detail:

- An automatic approach on mesh import using Mesh level of detail (LOD).
- A manual approach configured in the 3D node using Visibility ranges (HLOD).
- Decals and lights can also benefit from level of detail using their respective **Distance Fade** properties.

While they can be used independently, these approaches are most effective when used together. For example, you can set up visibility ranges to hide particle effects that are too far away from the player to notice. At the same time, you can rely on mesh LOD to make the particle effect's meshes rendered with less detail at a distance.

Visibility ranges are also a good way to set up *impostors* for distant geometry (see below).

Billboards and imposters

The simplest version of using transparency to deal with LOD is billboards. For example, you can use a single transparent quad to represent a tree at distance. This can be very cheap to render, unless of course, there are many trees in front of each other. In this case, transparency may start eating into fill rate (for more information on fill rate, see GPU optimization).

An alternative is to render not just one tree, but a number of trees together as a group. This can be especially effective if you can see an area but cannot physically approach it in a game.

You can make imposters by pre-rendering views of an object at different angles. Or you can even go one step further, and periodically re-render a view of an object onto a texture to be used as an imposter. At a distance, you need to move the viewer a considerable distance for the angle of view to change significantly. This can be complex to get working, but may be worth it depending on the type of project you are making.

Use instancing (MultiMesh)

If several identical objects have to be drawn in the same place or nearby, try using MultiMesh instead. MultiMesh allows the drawing of many thousands of objects at very little performance cost, making it ideal for flocks, grass, particles, and anything else where you have thousands of identical objects.

See also the Using MultiMesh documentation.

Bake lighting

Lighting objects is one of the most costly rendering operations. Realtime lighting, shadows (especially multiple lights), and global illumination are especially expensive. They may simply be too much for lower power mobile devices to handle.

Consider using baked lighting, especially for mobile. This can look fantastic, but has the downside that it will not be dynamic. Sometimes, this is a tradeoff worth making.

See Using Lightmap global illumination for instructions on using baked lightmaps. For best performance, you should set lights' bake mode to **Static** as opposed to the default **Dynamic**, as this will skip real-time lighting on meshes that have baked lighting.

The downside of lights with the **Static** bake mode is that they can't cast shadows onto meshes with baked lighting. This can make scenes with outdoor environments and dynamic objects look flat. A good balance between performance and quality is to keep **Dynamic** for the DirectionalLight3D node, and use **Static** for most (if not all) omni and spot lights.

Animation and skinning

Animation and vertex animation such as skinning and morphing can be very expensive on some platforms. You may need to lower the polycount considerably for animated models, or limit the number of them on screen at any given time. You can also reduce the animation rate for distant or occluded meshes, or pause the animation entirely if the player is unlikely to notice the animation being stopped.

The VisibleOnScreenEnabler3D and VisibleOnScreenNotifier3D nodes can be useful for this purpose.

Large worlds

If you are making large worlds, there are different considerations than what you may be familiar with from smaller games.

Large worlds may need to be built in tiles that can be loaded on demand as you move around the world. This can prevent memory use from getting out of hand, and also limit the processing needed to the local area.

There may also be rendering and physics glitches due to floating point error in large worlds. This can be resolved using Large world coordinates. If using large world coordinates is an option, you may be able to use techniques such as orienting the world around the player (rather than the other way around), or shifting the origin periodically to keep things centred around `Vector3(0, 0, 0)`.

Animating thousands of objects

Animating thousands of fish with MultiMeshInstance3D

This tutorial explores a technique used in the game ABZU [https://www.gdcvault.com/play/1024409/Creating-the-Art-of-ABZ] for rendering and animating thousands of fish using vertex animation and static mesh instancing.

In Godot, this can be accomplished with a custom Shader and a MultiMeshInstance3D. Using the following technique you can render thousands of animated objects, even on low end hardware.

We will start by animating one fish. Then, we will see how to extend that animation to thousands of fish.

Animating one Fish

We will start with a single fish. Load your fish model into a MeshInstance3D and add a new ShaderMaterial.

Here is the fish we will be using for the example images, you can use any fish model you like.

Note

The fish model in this tutorial is made by QuaterniusDev [https://quaternius.com] and is shared with a creative commons license. CC0 1.0 Universal (CC0 1.0) Public Domain Dedication https://creativecommons.org/publicdomain/zero/1.0/

Typically, you would use bones and a Skeleton3D to animate objects. However, bones are animated on the CPU and so you end having to calculate thousands of operations every frame and it becomes impossible to have thousands of objects. Using vertex animation in a vertex shader, you avoid using bones and can instead calculate the full animation in a few lines of code and completely on the GPU.

The animation will be made of four key motions:

1. A side to side motion
2. A pivot motion around the center of the fish
3. A panning wave motion
4. A panning twist motion

All the code for the animation will be in the vertex shader with uniforms controlling the amount of motion. We use uniforms to control the strength of the motion so that you can tweak the animation in editor and see the results in real time, without the shader having to recompile.

All the motions will be made using cosine waves applied to `VERTEX` in model space. We want the vertices to be in model space so that the motion is always relative to the orientation of the fish. For example, side-to-side will always move the fish back and forth in its left to right direction, instead of on the `x` axis in the world orientation.

In order to control the speed of the animation, we will start by defining our own time variable using `TIME`.

```
//time_scale is a uniform float
float time = TIME * time_scale;
```

The first motion we will implement is the side to side motion. It can be made by offsetting `VERTEX.x` by `cos` of `TIME`. Each time the mesh is rendered, all the vertices will move to the side by the amount of `cos(time)`.

```
//side_to_side is a uniform float
VERTEX.x += cos(time) * side_to_side;
```

The resulting animation should look something like this:

Next, we add the pivot. Because the fish is centered at $(0, 0)$, all we have to do is multiply `VERTEX` by a rotation matrix for it to rotate around the center of the fish.

We construct a rotation matrix like so:

```
//angle is scaled by 0.1 so that the fish only pivots and
doesn't rotate all the way around
//pivot is a uniform float
float pivot_angle = cos(time) * 0.1 * pivot;
mat2 rotation_matrix = mat2(vec2(cos(pivot_angle), -
sin(pivot_angle)), vec2(sin(pivot_angle), cos(pivot_angle)));
```

And then we apply it in the `x` and `z` axes by multiplying it by `VERTEX.xz`.

```
VERTEX.xz = rotation_matrix * VERTEX.xz;
```

With only the pivot applied you should see something like this:

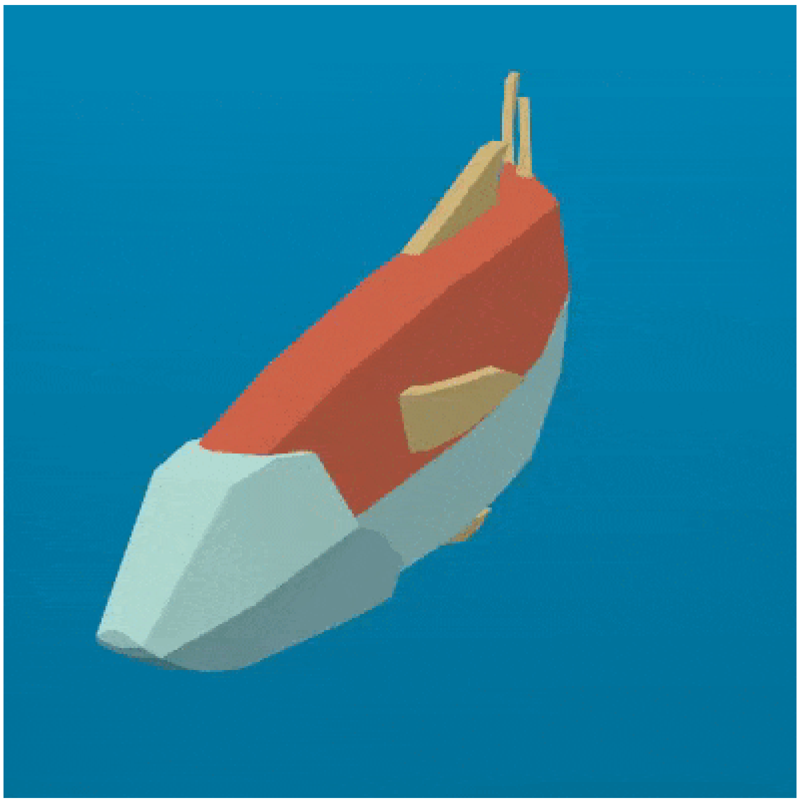
The next two motions need to pan down the spine of the fish. For that, we need a new variable, `body`. `body` is a float that is `0` at the tail of the fish and `1` at its head.

```
float body = (VERTEX.z + 1.0) / 2.0; //for a fish centered at (0, 0) with a length of 2
```

The next motion is a cosine wave that moves down the length of the fish. To make it move along the spine of the fish, we offset the input to `cos` by the position along the spine, which is the variable we defined above, `body`.

```
//wave is a uniform float
VERTEX.x += cos(time + body) * wave;
```

This looks very similar to the side to side motion we defined above, but in this one, by using `body` to offset `cos` each vertex along the spine has a different position in the wave making it look like a wave is moving along the fish.



The last motion is the twist, which is a panning roll along the spine. Similarly to the pivot, we first construct a rotation matrix.

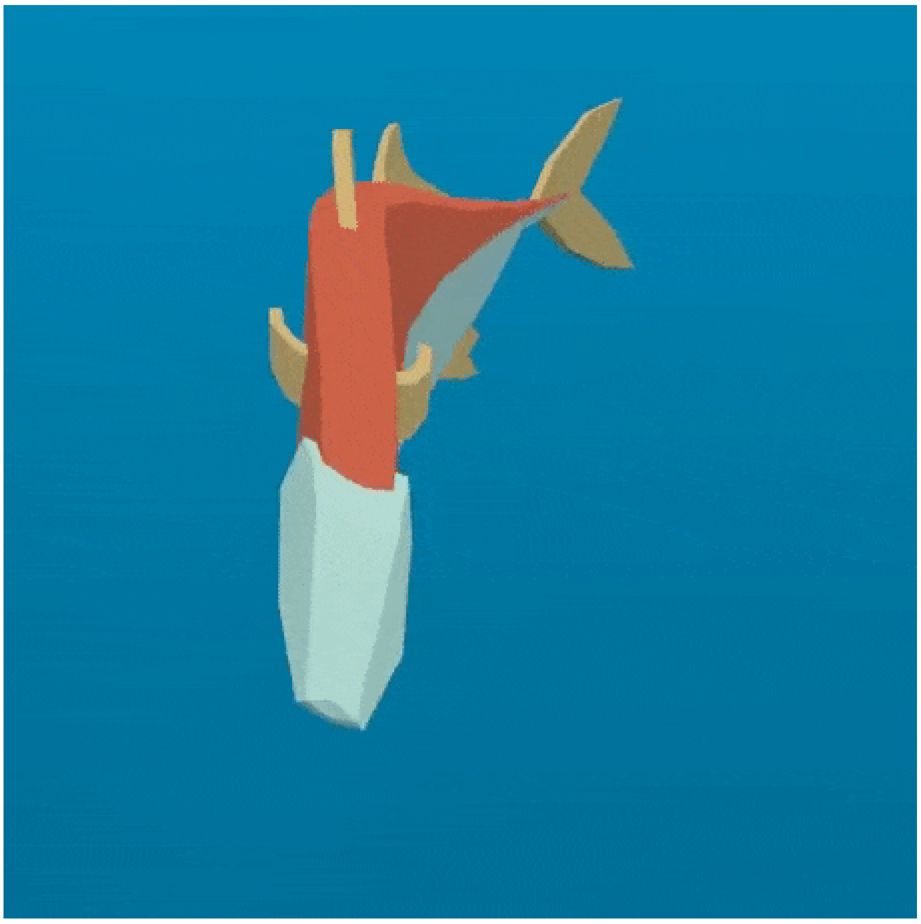
```
//twist is a uniform float
float twist_angle = cos(time + body) * 0.3 * twist;
mat2 twist_matrix = mat2(vec2(cos(twist_angle), -
sin(twist_angle)), vec2(sin(twist_angle), cos(twist_angle)));
```

We apply the rotation in the `xy` axes so that the fish appears to roll around its spine. For this to work, the fish's spine needs to be centered on the `z` axis.

```
VERTEX.xy = twist_matrix * VERTEX.xy;
```

Here is the fish with twist applied:

If we apply all these motions one after another, we get a fluid jelly-like motion.



Normal fish swim mostly with the back half of their body. Accordingly, we need to limit the panning motions to the back half of the fish. To do this, we create a new variable, `mask`.

`mask` is a float that goes from `0` at the front of the fish to `1` at the end using `smoothstep` to control the point at which the transition from `0` to `1` happens.

```
//mask_black and mask_white are uniforms
float mask = smoothstep(mask_black, mask_white, 1.0 - body);
```

Below is an image of the fish with `mask` used as `COLOR`:

For the wave, we multiply the motion by `mask` which will limit it to the back half.

```
//wave motion with mask
VERTEX.x += cos(time + body) * mask * wave;
```

In order to apply the mask to the twist, we use `mix`. `mix` allows us to mix the vertex position between a fully rotated vertex and one that is not rotated. We need to use `mix` instead of multiplying `mask` by the rotated `VERTEX` because we are not adding the motion to the `VERTEX` we are replacing the `VERTEX` with the rotated version. If we multiplied that by `mask`, we would shrink the fish.

```
//twist motion with mask
VERTEX.xy = mix(VERTEX.xy, twist_matrix * VERTEX.xy, mask);
```

Putting the four motions together gives us the final animation.

Go ahead and play with the uniforms in order to alter the swim cycle of the fish. You will find that you can create a wide variety of swim styles using these four motions.

Making a school of fish

Godot makes it easy to render thousands of the same object using a MultiMeshInstance3D node.

A MultiMeshInstance3D node is created and used the same way you would make a MeshInstance3D node. For this tutorial, we will name the MultiMeshInstance3D node `School`, because it will contain a school of fish.

Once you have a MultiMeshInstance3D add a MultiMesh, and to that MultiMesh add your Mesh with the shader from above.

MultiMeshes draw your Mesh with three additional per-instance properties: Transform (rotation, translation, scale), Color, and Custom. Custom is used to pass in 4 multi-use variables using a Color.

`instance_count` specifies how many instances of the mesh you want to draw. For now, leave `instance_count` at `0` because you cannot change any of the other parameters while `instance_count` is larger than `0`. We will set `instance count` in GDScript later.

`transform_format` specifies whether the transforms used are 3D or 2D. For this tutorial, select 3D.

For both `color_format` and `custom_data_format` you can choose between `None`, `Byte`, and `Float`. `None` means you won't be passing in that data (either a per-instance `COLOR` variable, or `INSTANCE_CUSTOM`) to the shader. `Byte` means each number making up the color you pass in will be stored with 8 bits while `Float` means each number will be stored in a floating-point number (32 bits). `Float` is slower but more precise, `Byte` will take less memory and be faster, but you may see some visual artifacts.

Now, set `instance_count` to the number of fish you want to have.

Next we need to set the per-instance transforms.

There are two ways to set per-instance transforms for MultiMeshes. The first is entirely in editor and is described in the MultiMeshInstance3D tutorial.

The second is to loop over all the instances and set their transforms in code. Below, we use GDScript to loop over all the instances and set their transform to a random position.

```
for i in range($School.multimesh.instance_count):
  var position = Transform3D()
  position = position.translated(Vector3(randf() * 100 - 50,
randf() * 50 - 25, randf() * 50 - 25))
  $School.multimesh.set_instance_transform(i, position)
```

Running this script will place the fish in random positions in a box around the position of the MultiMeshInstance3D.

Note

If performance is an issue for you, try running the scene with fewer fish.

Notice how all the fish are all in the same position in their swim cycle? It makes them look very robotic. The next step is to give each fish a different position in the swim cycle so the entire school looks more organic.

Animating a school of fish

One of the benefits of animating the fish using `cos` functions is that they are animated with one parameter, `time`. In order to give each fish a unique position in the swim cycle, we only need to offset `time`.

We do that by adding the per-instance custom value `INSTANCE_CUSTOM` to `time`.

```
float time = (TIME * time_scale) + (6.28318 *
INSTANCE_CUSTOM.x);
```

Next, we need to pass a value into `INSTANCE_CUSTOM`. We do that by adding one line into the `for` loop from above. In the `for` loop we assign each instance a set of four random floats to use.

```
$School.multimesh.set_instance_custom_data(i, Color(randf(),
randf(), randf(), randf()))
```

Now the fish all have unique positions in the swim cycle. You can give them a little more individuality by using `INSTANCE_CUSTOM` to make them swim faster or slower by multiplying by `TIME`.

```
//set speed from 50% - 150% of regular speed
float time = (TIME * (0.5 + INSTANCE_CUSTOM.y) * time_scale)
+ (6.28318 * INSTANCE_CUSTOM.x);
```

You can even experiment with changing the per-instance color the same way you changed the per-instance custom value.

One problem that you will run into at this point is that the fish are animated, but they are not moving. You can move them by updating the per-instance transform for each fish every frame. Although doing so will be faster than moving thousands of MeshInstance3Ds per frame, it'll still likely be slow.

In the next tutorial we will cover how to use GPUParticles3D to take advantage of the GPU and move each fish around individually while still receiving the benefits of instancing.

Controlling thousands of fish with Particles

The problem with MeshInstance3D is that it is expensive to update their transform array. It is great for placing many static objects around the scene. But it is still difficult to move the objects around the scene.

To make each instance move in an interesting way, we will use a GPUParticles3D node. Particles take advantage of GPU acceleration by computing and setting the per-instance information in a Shader.

First create a Particles node. Then, under "Draw Passes" set the Particle's "Draw Pass 1" to your Mesh. Then under "Process Material" create a new ShaderMaterial.

Set the `shader_type` to `particles`.

```
shader_type particles
```

Then add the following two functions:

```
float rand_from_seed(in uint seed) {
  int k;
  int s = int(seed);
  if (s == 0)
    s = 305420679;
  k = s / 127773;
  s = 16807 * (s - k * 127773) - 2836 * k;
  if (s < 0)
    s += 2147483647;
  seed = uint(s);
  return float(seed % uint(65536)) / 65535.0;
}

uint hash(uint x) {
  x = ((x >> uint(16)) ^ x) * uint(73244475);
  x = ((x >> uint(16)) ^ x) * uint(73244475);
  x = (x >> uint(16)) ^ x;
  return x;
}
```

These functions come from the default ParticleProcessMaterial. They are used to generate a random number from each particle's `RANDOM_SEED`.

A unique thing about particle shaders is that some built-in variables are saved across frames. `TRANSFORM`, `COLOR`, and `CUSTOM` can all be accessed in the shader of the mesh, and also in the particle shader the next time it is run.

Next, setup your `start()` function. Particles shaders contain a `start()` function and a `process()` function.

The code in the `start()` function only runs when the particle system starts. The code in the `process()` function will always run.

We need to generate 4 random numbers: 3 to create a random position and one for the random offset of the swim cycle.

First, generate 4 seeds inside the `start()` function using the `hash()` function provided above:

```
uint alt_seed1 = hash(NUMBER + uint(1) + RANDOM_SEED);
uint alt_seed2 = hash(NUMBER + uint(27) + RANDOM_SEED);
uint alt_seed3 = hash(NUMBER + uint(43) + RANDOM_SEED);
uint alt_seed4 = hash(NUMBER + uint(111) + RANDOM_SEED);
```

Then, use those seeds to generate random numbers using `rand_from_seed`:

```
CUSTOM.x = rand_from_seed(alt_seed1);
vec3 position = vec3(rand_from_seed(alt_seed2) * 2.0 - 1.0,
                     rand_from_seed(alt_seed3) * 2.0 - 1.0,
                     rand_from_seed(alt_seed4) * 2.0 - 1.0);
```

Finally, assign `position` to `TRANSFORM[3].xyz`, which is the part of the transform that holds the position information.

```
TRANSFORM[3].xyz = position * 20.0;
```

Remember, all this code so far goes inside the `start()` function.

The vertex shader for your mesh can stay the exact same as it was in the previous tutorial.

Now you can move each fish individually each frame, either by adding to the `TRANSFORM` directly or by writing to `VELOCITY`.

Let's transform the fish by setting their `VELOCITY` in the `start()` function.

```
VELOCITY.z = 10.0;
```

This is the most basic way to set `VELOCITY` every particle (or fish) will have the same velocity.

Just by setting `VELOCITY` you can make the fish swim however you want. For example, try the code below.

```
VELOCITY.z = cos(TIME + CUSTOM.x * 6.28) * 4.0 + 6.0;
```

This will give each fish a unique speed between `2` and `10`.

You can also let each fish change its velocity over time if you set the velocity in the `process()` function.

If you used `CUSTOM.y` in the last tutorial, you can also set the speed of the swim animation based on the `VELOCITY`. Just use `CUSTOM.y`.

```
CUSTOM.y = VELOCITY.z * 0.1;
```

This code gives you the following behavior:

Using a ParticleProcessMaterial you can make the fish behavior as simple or complex as you like. In this tutorial we only set Velocity, but in your own Shaders you can also set `COLOR`, rotation, scale (through `TRANSFORM`). Please refer to the Particles Shader Reference for more information on particle shaders.

Using multiple threads

Threads

Threads allow simultaneous execution of code. It allows off-loading work from the main thread.

Godot supports threads and provides many handy functions to use them.

Note

If using other languages (C#, C++), it may be easier to use the threading classes they support.

Warning

Before using a built-in class in a thread, read Thread-safe APIs first to check whether it can be safely used in a thread.

Creating a Thread

To create a thread, use the following code:

GDScript

```
var thread: Thread

# The thread will start here.
func _ready():
    thread = Thread.new()
    # You can bind multiple arguments to a function Callable.
    thread.start(_thread_function.bind("Wafflecopter"))
```

```
# Run here and exit.
# The argument is the bound data passed from start().
func _thread_function(userdata):
    # Print the userdata ("Wafflecopter")
    print("I'm a thread! Userdata is: ", userdata)


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    thread.wait_to_finish()
```

Your function will, then, run in a separate thread until it returns. Even if the function has returned already, the thread must collect it, so call Thread.wait_to_finish(), which will wait until the thread is done (if not done yet), then properly dispose of it.

Warning

Creating threads at run-time is slow on Windows and should be avoided to prevent stuttering. Semaphores, explained later on this page, should be used instead.

Mutexes

Accessing objects or data from multiple threads is not always supported (if you do it, it will cause unexpected behaviors or crashes). Read the Thread-safe APIs documentation to understand which engine APIs support multiple thread access.

When processing your own data or calling your own functions, as a rule, try to avoid accessing the same data directly from different threads. You may run into synchronization problems, as the data is not always updated between CPU cores when modified. Always use a Mutex when accessing a piece of data from different threads.

When calling Mutex.lock(), a thread ensures that all other threads will be blocked (put on suspended state) if they try to *lock* the same mutex. When the mutex is unlocked by calling Mutex.unlock(), the other threads will be allowed to proceed with the lock (but only one at a time).

Here is an example of using a Mutex:

GDScript

```gdscript
var counter := 0
var mutex: Mutex
var thread: Thread


# The thread will start here.
func _ready():
    mutex = Mutex.new()
    thread = Thread.new()
    thread.start(_thread_function)

    # Increase value, protect it with Mutex.
    mutex.lock()
    counter += 1
    mutex.unlock()


# Increment the value from the thread, too.
func _thread_function():
    mutex.lock()
    counter += 1
    mutex.unlock()


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    thread.wait_to_finish()
    print("Counter is: ", counter) # Should be 2.
```

Semaphores

Sometimes you want your thread to work *"on demand"*. In other words, tell it when to work and let it suspend when it isn't doing anything. For this, Semaphores are used. The function Semaphore.wait() is used in the thread to suspend it until some data arrives.

The main thread, instead, uses Semaphore.post() to signal that data is ready to be processed:

GDScript

```
var counter := 0
var mutex: Mutex
var semaphore: Semaphore
var thread: Thread
var exit_thread := false


# The thread will start here.
func _ready():
    mutex = Mutex.new()
    semaphore = Semaphore.new()
    exit_thread = false

    thread = Thread.new()
    thread.start(_thread_function)


func _thread_function():
    while true:
        semaphore.wait() # Wait until posted.

        mutex.lock()
        var should_exit = exit_thread # Protect with Mutex.
        mutex.unlock()

        if should_exit:
            break

        mutex.lock()
        counter += 1 # Increment counter, protect with Mutex.
        mutex.unlock()
```

```gdscript
func increment_counter():
    semaphore.post() # Make the thread process.


func get_counter():
    mutex.lock()
    # Copy counter, protect with Mutex.
    var counter_value = counter
    mutex.unlock()
    return counter_value


# Thread must be disposed (or "joined"), for portability.
func _exit_tree():
    # Set exit condition to true.
    mutex.lock()
    exit_thread = true # Protect with Mutex.
    mutex.unlock()

    # Unblock by posting.
    semaphore.post()

    # Wait until it exits.
    thread.wait_to_finish()

    # Print the counter.
    print("Counter is: ", counter)
```

Thread-safe APIs

Threads

Threads are used to balance processing power across CPUs and cores. Godot supports multithreading, but not in the whole engine.

Below is a list of ways multithreading can be used in different areas of Godot.

Global scope

Global Scope singletons are all thread-safe. Accessing servers from threads is supported (for RenderingServer and Physics servers, ensure threaded or thread-safe operation is enabled in the project settings!).

This makes them ideal for code that creates dozens of thousands of instances in servers and controls them from threads. Of course, it requires a bit more code, as this is used directly and not within the scene tree.

Scene tree

Interacting with the active scene tree is **NOT** thread-safe. Make sure to use mutexes when sending data between threads. If you want to call functions from a thread, the *call_deferred* function may be used:

```
# Unsafe:
node.add_child(child_node)
# Safe:
node.call_deferred("add_child", child_node)
```

However, creating scene chunks (nodes in tree arrangement) outside the active tree is fine. This way, parts of a scene can be built or instantiated in a thread, then added in the main thread:

```
var enemy_scene = load("res://enemy_scene.scn")
var enemy = enemy_scene.instantiate()
enemy.add_child(weapon) # Set a weapon.
world.call_deferred("add_child", enemy)
```

Still, this is only really useful if you have **one** thread loading data. Attempting to load or create scene chunks from multiple threads may work, but you risk resources (which are only loaded once in Godot) tweaked by the multiple threads, resulting in unexpected behaviors or crashes.

Only use more than one thread to generate scene data if you *really* know what you are doing and you are sure that a single resource is not being used or set in multiple ones. Otherwise, you are safer just using the servers API (which is fully thread-safe) directly and not touching scene or resources.

Rendering

Instancing nodes that render anything in 2D or 3D (such as Sprite) is *not* thread-safe by default. To make rendering thread-safe, set the **Rendering > Driver > Thread Model** project setting to **Multi-Threaded**.

Note that the Multi-Threaded thread model has several known bugs, so it may not be usable in all scenarios.

GDScript arrays, dictionaries

In GDScript, reading and writing elements from multiple threads is OK, but anything that changes the container size (resizing, adding or removing elements) requires locking a mutex.

Resources

Modifying a unique resource from multiple threads is not supported. However handling references on multiple threads is supported, hence loading resources on a thread is as well - scenes, textures, meshes, etc - can be loaded and manipulated on a thread and then added to the active scene on the main thread. The limitation here is as described above, one must be careful not to load the same resource from multiple threads at once, therefore it is easiest to use **one** thread for loading and modifying resources, and then the main thread for adding them.