Godot 4.1

# Godot 4: From zero to full game

**A step-by-step guide to creating computer games in the Godot Engine with examples, tutorials, performance tips, and much more.**


**Filip Rachůnek**

- Shoot the enemies
- Hits, explosions, and more sounds

**Fifth: Improve the player's ship**

- Movement tilt
- Accelerate
- Modules and debris
- Shield
- Calculate collisions and update HUD
- Flash shield and player on hit
- Jet flames
- Bullet trails

**Sixth: Create the first level**

- Design the system of levels, waves, and enemy behaviors
- Implement a level loader
- Create enemy ships with weapons
- Spawn some power-ups
- Launch homing missiles
- Add a boss with multiple weapons and HUD
- Shoot player's missiles
- Optimize level loading and preparation

**Seventh: Game workflow**

- Menu screen
- Pause game
- Settings
- Window size
- Gamma correction
- Vertical synchronization
- Audio mixer
- Internationalization and localization
- Splash screen and icon
- Key bindings
- Intro scene and music

**Eighth: Export and release your game**

- Leverage Godot's export features
- Optimize the game file size

**Ninth: More game ideas**

# Copyright

Copyright © 2023 Filip Rachůnek, all rights reserved.

No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from the author.

# Preface

Are you interested in venturing into game development and creating your very own computer game? Thankfully, there is an abundance of freely available tools, frameworks, assets, and resources at your disposal, making the process easier than ever before. This practical book offers you the opportunity to gain firsthand experience in utilizing **Godot 4** and crafting your own code using **GDScript**, a user-friendly and easy-to-learn language tightly integrated with the Godot Engine.

Unlike a mere theoretical book that only covers the features of a game engine and provides a few basic tips for starting a project, this book takes a different approach. It doesn't leave the reader to face the daunting battle against numerous bugs and difficulties on their own. Every single paragraph, every edge-case tip, every suggestion of a possible workaround is based on my hands-on experience with developing my own game, creating assets for it myself, and continuous testing and debugging on Windows and Mac simultaneously. The official documentation may not provide all the necessary details, and some of these omissions can be quite bothersome. However, this book aims to assist you in overcoming these obstacles effortlessly, allowing you to concentrate on the core aspects of game design and development with a sense of freedom.

Just a few examples of useful tips you will learn here:

1. After your friend installed your game on his computer, he promptly informed you about the choppy and stuttering performance, along with significant FPS drops within the initial 20 seconds of gameplay. Why is it happening, and how can you prevent it?
2. You created a cool-looking particle effect that works perfectly on Mac, but displays ugly visual artifacts on Windows. What can you do about it?
3. The game showcases an interactive explosion using a shader material, and you intend to animate specific attributes such as speed, color palette, smoke effect, and more. Unfortunately, when multiple instances of the explosion are added to the scene, they all animate identically. How can you ensure that each explosion behaves uniquely?
4. Godot provides a simple way to pause the game, but it doesn’t affect animated shader materials. What can you do to pause everything?
5. What is the most efficient method to consistently monitor the statistics of the game scene, enabling you to quickly detect any leaks before they escalate into significant issues in the future?

I have created and recorded a video course on YouTube to supplement the book. If you're interested in observing the game creation process in action, you can visit the provided URL to watch it for free: https://www.youtube.com/@FencerDevLog

Remember to leave a comment! I greatly appreciate receiving feedback and suggestions to enhance the tutorial. Your input is valuable to me and helps me improve.

## Who this book is for

**Short answer:** Anybody who is open to learning and unafraid of exploring new concepts.

**Long answer:** I firmly believe that anyone, regardless of their proficiency in coding basic programs or scripts, can effectively complete a game project. The book aims to facilitate the learning journey by providing detailed explanations, accompanied by helpful screenshots and code examples, ensuring that each step is comprehensible and accessible to all.

While having prior experience with Godot and Blender is beneficial, it is **not mandatory** for following the content. By leveraging both the book and the accompanying video course, you will receive comprehensive guidance on every aspect required to develop and publish your own game. Having a foundational understanding of **GDScript** would be beneficial for comprehending the code examples.

## What will you learn?

Upon completing the book, you will possess a **functional game** that can be run and played, allowing you to make additional improvements as desired. Equally crucial is your **understanding of the underlying principles** guiding this process. You will gain insight into various **alternative approaches** to the challenges encountered in game development, comprehending their advantages and disadvantages. Furthermore, you will have a collection of **unique assets** at your disposal, along with a clear understanding of their creation and purpose. Finally, you will discover that the essential requirements to embark on your project are solely **your time,**

**patience, and enthusiasm**. No additional financial investment is necessary, making it an accessible endeavor for all.

The game development process outlined in the book is not the sole method for creating a game in Godot. However, it serves as a guide to understanding the fundamental principles of Godot that can be applied to various other types of games. Embrace experimentation without fear - in most situations, there are multiple solutions available, allowing you to discover the style that aligns best with your preferences and needs.

## What this book covers

1. **Prepare the workspace:** Emphasizes the significance of utilizing a version control system, and assists you in properly setting it up across various platforms. Additionally, it provides instructions on preparing the environment for Godot and Blender, ensuring seamless integration between the two tools.
2. **The high-level concept:** Provides an overview of the essential logic and workflow of the game, along with detailed, step-by-step guidance on constructing a modular model for the player's ship. It also explains why we prefer creating assets in a 3D environment and then projecting them onto a 2D plane instead of using sprites or pixel art. It discusses the reasons behind this choice and why it works better for the project.
3. **Prepare the basic scene:** Encompasses the process of constructing the initial scene for our game, including the setup of lights, camera, dynamic background, and fundamental game objects. Moreover, it explores multiple techniques for creating and applying materials to models.
4. **Enemies, layers, collisions:** Describes a process that involves several steps to create an asteroid model and integrate it into a game. It also includes configuring the collision system, managing object destruction, implementing shooting mechanics, and utilizing particle and shader systems to generate realistic explosions.
5. **Improve the player’s ship:** Improves the player's ship model and enhances its behavior to make it more advanced and visually appealing. This includes incorporating modular components for breaking up the ship, implementing a shield mechanism, emitting flames from the engines, and adding trails to bullets, among other enhancements.

6. **Create the first level:** Shows how to develop a versatile level system that can be applied universally. It covers aspects such as enemies shooting different types of projectiles, incorporating power-ups, implementing homing missiles, and creating a formidable boss enemy. Additionally, the chapter offers several tips for optimizing performance throughout the development process.
7. **Game workflow:** Takes you step by step through the process of managing multiple scenes in a game, including the introduction, menu, and settings screens, and demonstrates how to smoothly transition between them. It explores various aspects such as incorporating music and sound effects, implementing game localization for different languages, setting up customizable key bindings, and integrating a sophisticated animation system.
8. **Export and release your game:** Outlines the process of preparing a finished game for exporting and sharing with others. It encompasses a range of techniques and suggestions to reduce the file size of the exported game, making it more efficient for distribution.
9. **More game ideas:** Proposes and illustrates supplementary concepts to enhance the appeal of your game for the audience. Accompanying the suggestions are code snippets or screenshots that emphasize the underlying concepts.

**Appendix 1: Modeling in Blender**

In this extra section, I share some of the most frequently used tips and tricks in Blender that I utilized while creating my game. These valuable insights can help you become proficient in creating assets and improve your game development abilities.

**Appendix 2: Godot 4 overview**

As the reader may be unfamiliar with the Godot environment, the second appendix provides the basic description of the key elements such as scenes, nodes, editor, inspector, and other essential tools commonly utilized and referenced throughout the content.

## Conventions used

There are a number of text conventions used throughout this book.

**`Code in text:`** Indicates file or folder names, keywords, annotations, path names, Godot/Blender labels, and user input. Example: “If you want to change the property value in **`AnimationPlayer`**, use the prefix **`@export`** to add it to **`Inspector`**.”

A block of code:

```
func get_all_children(in_node, array = []):
    array.push_back(in_node)
    for child in in_node.get_children():
        array = get_all_children(child, array)
    return array
```

**Bold:** Denotes a new term, a significant word, or any onscreen text that catches your attention. For instance: “To provide an additional layer of engagement, we will include an **achievement system**.”

*Italic:* Utilized to add captions to screenshots and provide additional information or comments in the form of side notes.

Tips or important notes:

> **Tip:** When using **Material Maker**, it's important to ensure that the generated material's final size is not larger than necessary. If your game doesn't showcase extensive and highly detailed models, it's likely unnecessary to attach textures with dimensions as large as **2048x2048** pixels.

## Disclaimer

Although I have made every effort to ensure that the code examples and snippets are error-free and typo-free, I cannot guarantee with absolute certainty that no bugs have been overlooked. If you encounter any difficulties while following the instructions to code your game, you are welcome to download or fork the sample project from GitHub and utilize it as a point of reference. If there is any update to the code, it will be updated in the GitHub repository as well.

https://github.com/FilipRachunek/space-shooter

In addition to completing the entire project, I also created snapshots of the game's status after each chapter. These snapshots can be referenced in case you encounter any mistakes while following the tutorial and are unsure how to fix the code. The URL of the corresponding subproject is included in the header of each chapter.

## About the author

I am Filip Rachůnek, a seasoned developer with over 25 years of full-time experience. With a strong mathematical foundation and extensive programming expertise, I have worked in diverse environments, ranging from large corporations to fledgling startups. In addition to my technical skills, I have also gained valuable leadership experience by guiding and mentoring developers in various technologies. Throughout my career, I have successfully overseen the completion of multiple projects.

Throughout my journey, game development has always been my true passion, driving me to dedicate a significant portion of my personal projects to this field. From crafting a Java-based board game server to creating engaging browser games using JavaScript, I have explored various avenues. However, my discovery of Godot has truly resonated with me, aligning perfectly with my creative vision. I firmly believe that Godot is the ideal platform for me, and I am committed to embracing it for the foreseeable future.

What else? I live in Prague, Czech Republic, with my beloved wife and three kids. I like to compose music, play piano, study chess, and write books. And, of course, I love to play games from other indie developers, and get inspired by them. :-)

Filip.Rachunek.com

# Intro: The importance of a good preparation

Before initiating our work, it is of utmost importance to conduct some preliminary research and assess the available tools for building our game. The objective of this tutorial is to commence the project with little to no financial investment, ergo our emphasis will be on utilizing free software options.

## Game engine

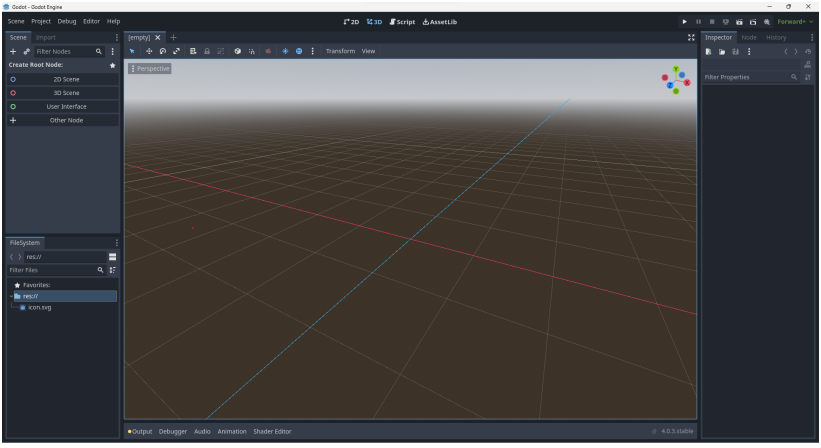
All right, this book is about **Godot**, so I won't pretend that I am going through some decision-making process now. The decision was already made. However, it could be intriguing to explore the factors that influenced my choice of Godot over other well-known game engines.

Before delving extensively into Godot, I conducted several experiments using **Unity** and **Unreal Engine**. Undoubtedly, both of these engines are exceptional and likely a superior choice for large teams due to their Pro and Enterprise licensing options. However, the concept of a **lone developer** is also widely embraced, and there are numerous advantages to undertaking every aspect of game development independently. By not depending on team members, you retain **complete ownership** of the profits and gain a comprehensive understanding of the game from multiple perspectives, among other benefits.

1. **Open-source and free:** Godot is an open-source game engine released under the MIT license, which means it's free to use and modify. This makes it an attractive option for independent developers or those on a tight budget. There are no strings attached. When you create a game using Godot, you retain complete ownership of everything without any commissions or paid licenses involved.
2. **Lightweight and efficient:** Godot is known for its lightweight nature and efficient performance. It has a small installation size, quick startup times, and runs well on lower-end hardware. This makes it suitable for projects with limited resources or for targeting platforms with lower specifications.
3. **Ease of use and beginner-friendly:** Godot offers a user-friendly interface and an intuitive visual scripting system that allows developers to create games without writing code. It also supports traditional programming languages like

GDScript (Python-like), C#, and C++, making it accessible to developers with different skill levels.

4. **Community and documentation:** Although Godot has a smaller user base compared to Unity or Unreal Engine, it has a dedicated and passionate community. The community actively contributes to documentation, tutorials, and asset sharing, providing support and resources to developers. At the time of writing, the official Discord server of Godot boasted nearly 64,000 members, with the beginner-focused channels being among the most actively utilized.
5. **Collaboration and integration:** Godot integrates smoothly with various free tools, particularly Blender. This collaboration proves highly advantageous, particularly considering the limited availability of official assets in Godot's asset store compared to Unity or Unreal. However, this situation encourages us to develop our own assets, resulting in a game that is entirely unique in all aspects.



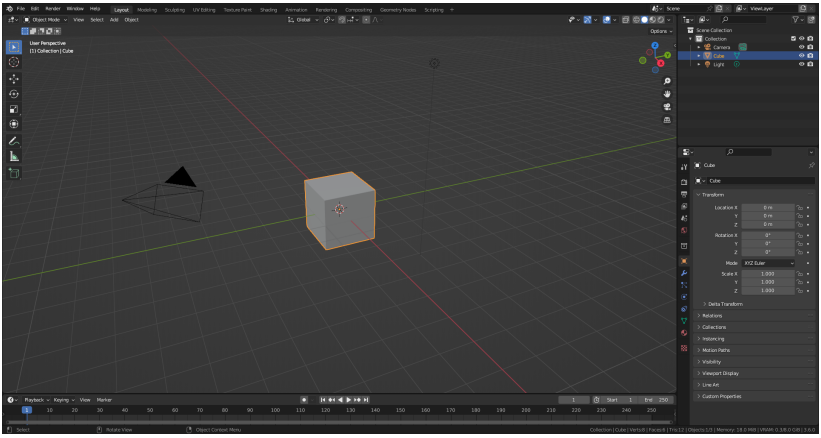

*Godot 4.0.3*

## Graphics

**Blender** can be considered the optimal choice for individuals who prefer not to invest in a commercial product. It encompasses a comprehensive set of features, allowing users to perform various tasks such as 3D modeling, UV mapping, texturing, material creation, rigging, and animation.

Using Blender for creating 3D models in your game offers several advantages:

1. **Open-source and Free:** Blender is an open-source software, meaning it is freely available for anyone to use. This eliminates the need for costly software licenses, making it an accessible option for indie developers or those on a tight budget.
2. **Versatile and Powerful:** Blender is a feature-rich 3D modeling tool with a wide range of capabilities. It supports various modeling techniques, including polygonal modeling, sculpting, and procedural modeling. It also provides advanced features for texturing, rigging, animation, and rendering, giving you a comprehensive suite of tools to create complex and visually appealing 3D models.
3. **Active Community and Resources:** Blender has a large and active community of users and developers. This means you can easily find tutorials, documentation, and helpful resources to learn and strengthen your skills. The Blender community is known for its willingness to share knowledge and provide assistance, making it a supportive environment for beginners and experienced artists alike.
4. **Integration and Interoperability:** Blender is designed to work well with other software and game engines. It supports common file formats for importing and exporting, allowing you to seamlessly integrate your 3D models into various game development pipelines. Blender is also compatible with industry-standard formats, such as FBX and OBJ, enabling smooth collaboration with other artists or game development teams.
5. **Constant Development and Updates:** Blender is continuously being developed and updated by a dedicated team of developers and volunteers. This means you can expect regular updates, bug fixes, and new features that enhance your modeling workflow and keep up with the latest industry standards.



*Blender 3.6*

In addition to Blender, I regularly utilize various other free applications and tools in my work. For instance:

1. For creating occasional 2D graphics like textures and palettes, I rely on **GIMP**. However, there are several other free alternatives available, such as **Krita** or the online tool **Pixlr**.
2. Throughout my journey towards completing a game project, I have come across numerous other free resources that prove to be incredibly useful and convenient. For example:
    a. Laigter (normal map generator)
    b. Material Maker (procedural material tool with its own asset repository)
    c. NormalMap Online (online normal map generator)
    d. DeepBump (Blender plugin to generate normal maps using ML)
    e. Free assets (models, textures, materials):
        i. ambientCG
        ii. sharetextures
        iii. TextureCan
        iv. 3D Textures

That was the first 16 pages of the introduction to the topic. Now, we will skip to the third chapter, from which we will show a part related to importing and starting the player's ship.

# Third: Prepare the first scene

**What we will learn in this chapter:**

1. 3D scene creation.
2. Designing the debug overlay.
3. How to create a 2D layer over a 3D scene.
4. Mapping keys to custom actions.
5. Working with shared materials.
6. How to structure the code of our game.
7. Creating a dynamic star field background.

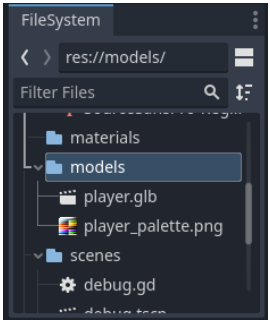
GitHub project: [The end of the chapter 3]

(...)

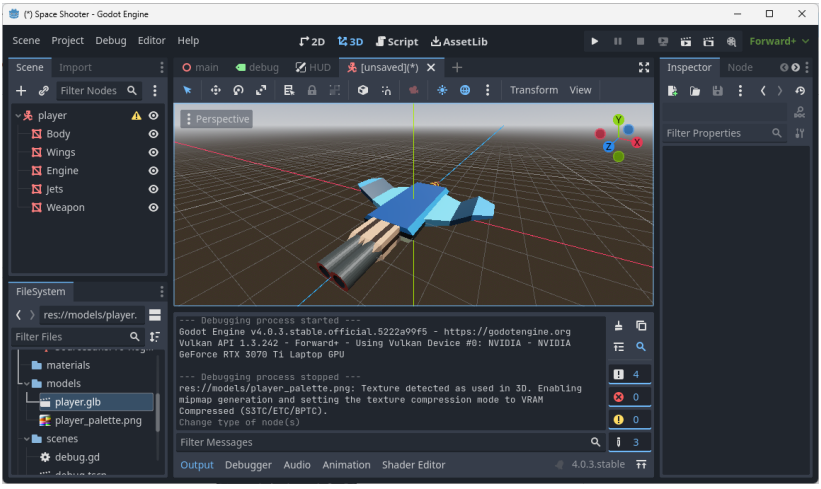
## Player's ship

Our approach will involve establishing the ship as an essential component within the main scene, ensuring that the ship's node remains **consistently present** in the scene tree. During the later phase of the project, we will introduce **dynamic** incorporation of the ship, considering the specific requirements of particular levels in the scenario.

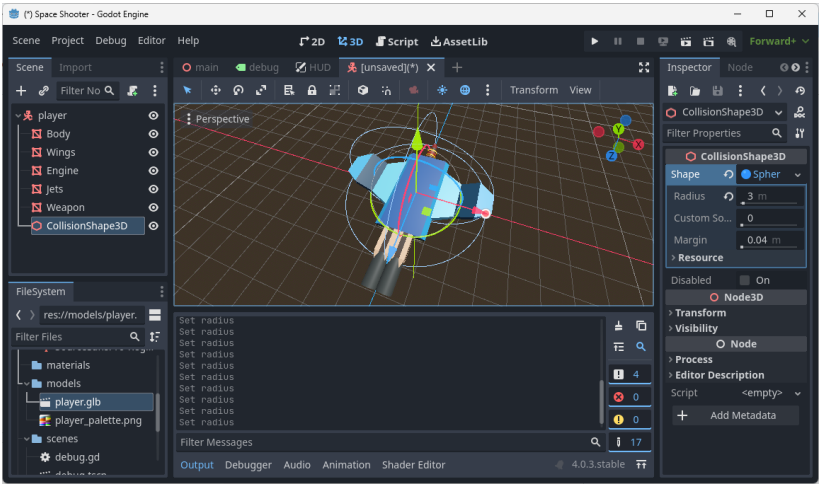
1. Drag your exported ship file **`player.glb`** to the folder **`models`**. Godot should import the model and extract the texture (palette) as a standalone image.



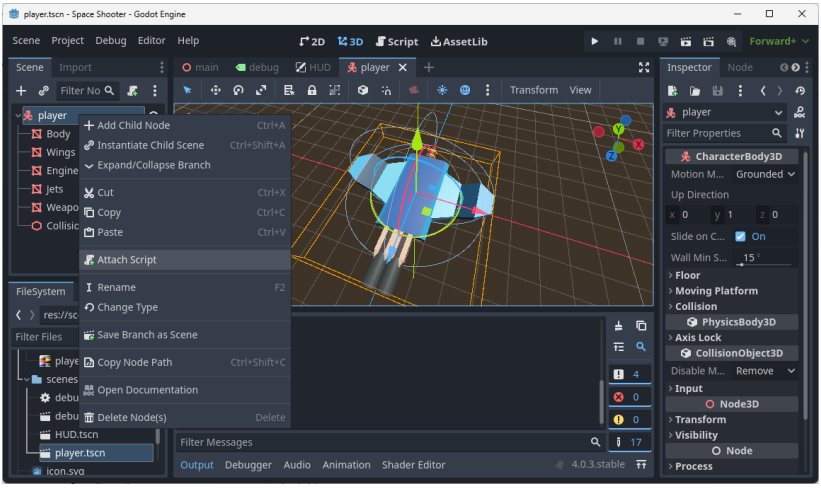
2. Right-click on **`player.glb`**, and select **`New Inherited Scene`**. A root node **`player`** will appear in the Scene panel. Right-click this node, select **`Change Type`**, and change in to **`CharacterBody3D`**. This is a special class provided by the Godot Engine, and it will help us handle the ship's movement and interactions. Also, you should see the ship's modules (body, wings, engine, etc.) as child nodes of this root.




3. To ensure that the ship won't leave the screen, we will soon add the boundaries to the scene. However, the model won't interact with them until we add a collision shape, so let's do it now. Add **`CollisionShape3D`** as another child node of the root, display its properties in **`Inspector`**, and assign a shape - a simple sphere should be fine for now. Open the 3D editor and modify the collision shape size to cover the ship.




4. **`CharacterBody3D`** will solely be utilized for movement and collision with the boundaries, while all other interactions will be managed within the code. Consequently, we will need to duplicate our collision shapes and associate them with an **`Area3D`** node. After we have familiarized ourselves with signals and event handling, I will provide a detailed explanation of this approach.
5. Save the scene as **`player.tscn`** to the folder **`scenes`**.
6. Right-click the root node, and select **`Attach Script`**. Create a new script **`player.gd`**. Godot will use a **`CharacterBody3D`** template to implement the basic movement functionality in the code.




7. Open the scene **`main.tscn`**, and drag **`player.tscn`** to the root as a child node. The ship will appear in the 3D editor. If desired, you can utilize the move gizmo to adjust the initial position of the ship. However, ensure that you do not move it outside the camera view. It is important to maintain a Y coordinate of 0 (zero).
8. Open the script **`player.gd`**. It should contain the pre-generated code related to **`CharacterBody3D`**:

```
extends CharacterBody3D

const SPEED = 5.0
const JUMP_VELOCITY = 4.5

# Get the gravity from the project settings to be synced with RigidBody
nodes.
var gravity = ProjectSettings.get_setting("physics/3d/default_gravity")

func _physics_process(delta):
    # Add the gravity.
    if not is_on_floor():
        velocity.y -= gravity * delta

    # Handle Jump.
    if Input.is_action_just_pressed("ui_accept") and is_on_floor():
        velocity.y = JUMP_VELOCITY

    # Get the input direction and handle the movement/deceleration.
```

```
    # As good practice, you should replace UI actions with custom gameplay
actions.
    var input_dir = Input.get_vector("ui_left", "ui_right", "ui_up",
"ui_down")
    var direction = (transform.basis * Vector3(input_dir.x, 0,
input_dir.y)).normalized()
    if direction:
        velocity.x = direction.x * SPEED
        velocity.z = direction.z * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0, SPEED)
        velocity.z = move_toward(velocity.z, 0, SPEED)

    move_and_slide()
```

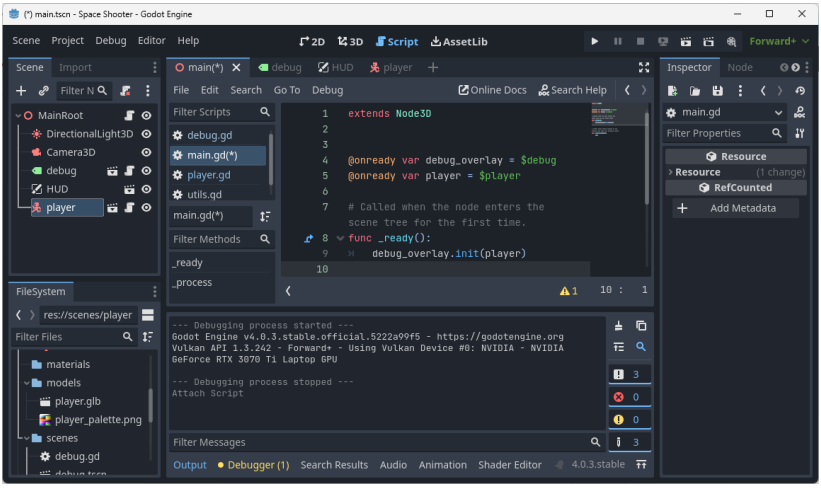
9. As evident from the script, there are certain functionalities that are unnecessary for our ship. For instance, since there is no gravity in space and the ship won't be able to jump, we should eliminate the corresponding sections from the code. The revised version of the code should appear as follows (we increased the constant SPEED as well):

```
extends CharacterBody3D

const SPEED = 30.0

func _physics_process(delta):
    # Get the input direction and handle the movement/deceleration.
    # As good practice, you should replace UI actions with custom gameplay
actions.
    var input_dir = Input.get_vector("ui_left", "ui_right", "ui_up",
"ui_down")
    var direction = (transform.basis * Vector3(input_dir.x, 0,
input_dir.y)).normalized()
    if direction:
        velocity.x = direction.x * SPEED
        velocity.z = direction.z * SPEED
    else:
        velocity.x = move_toward(velocity.x, 0, SPEED)
        velocity.z = move_toward(velocity.z, 0, SPEED)

    move_and_slide()
```

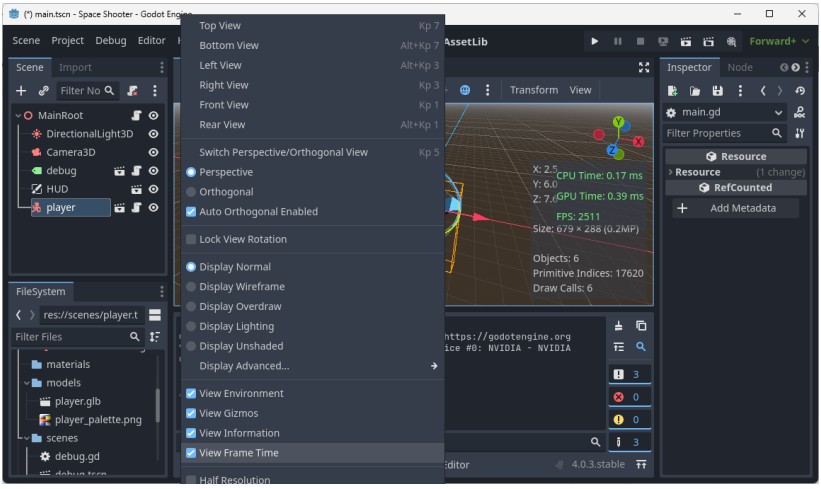
10. Since we finally have the player’s ship in the scene, we can display its data in the debug overlay:

a. Get back to the main scene, right-click the root node (**MainRoot**), and attach a new script **`main.gd`**. Open the script, and add to the top (after **`extends Node3D`**):
   i. **`@onready var debug_overlay = $debug`**
   ii. The value **`$debug`** may vary, depending on the node name in the tree. Simply drag the node to the code, and it will be filled automatically. Do the same for the player's node:
   iii. **`@onready var player = $player`**

b. Add this line to the function **`_ready()`**:
   i. **`debug_overlay.init(player)`**

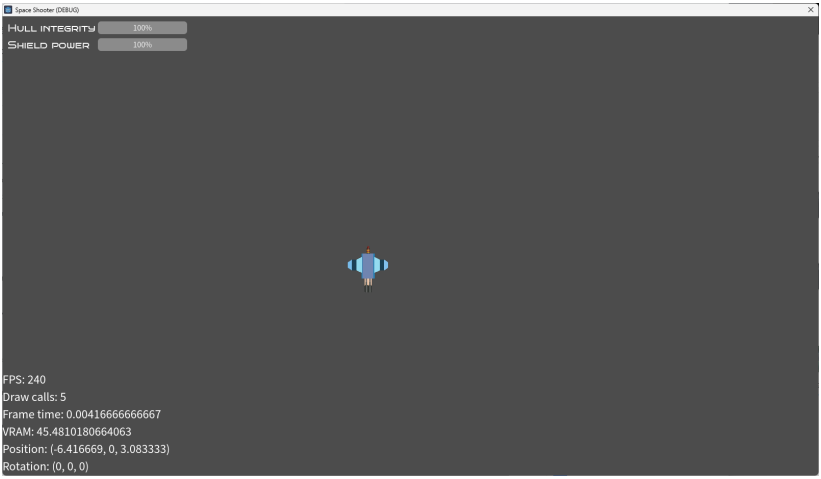



11. To ensure smooth performance and **monitor the scene's performance** as we introduce new objects and enhancements to the ship (such as additional weapons or particle-based jet flames), it would be beneficial to have real-time access to various information within the Godot 3D editor:
   a. Click the three dots in the top-left corner (the label **`Perspective`** is probably displayed next to it).
   b. Enable **`View Information`** and **`View Frame Time`**.
   c. Voilà! We can observe CPU Time, GPU Time, FPS, and other important statistical data.




> **Tip:** If you press **Ctrl** while dragging a node to the script, it will be dropped with a full `@onready` declaration.

Run the project. Since we have already completed the necessary setup by filling in the **`Main Scene`** property, Godot will not prompt for any additional information and will display the game window instantly. You will immediately see our ship positioned in the center of the screen. To **maneuver** the ship, simply press the default move keys, which are the **arrow keys** on your keyboard.



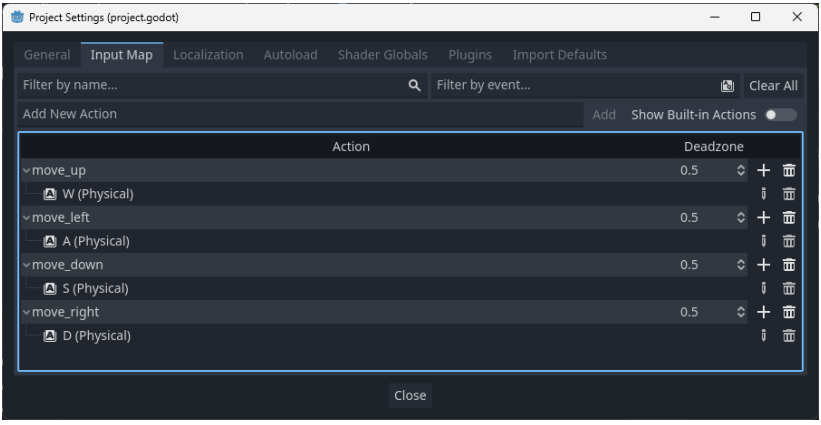

By observing the debug overlay, you will notice that it displays the ship's current **position** and **rotation**. This allows us to confirm that the Y coordinate remains consistently set to 0. To proceed, move the ship to each of the four edges of the game window and take note of the specific coordinates. Record the **X** coordinate for the **left** and **right** edges, as well as the **Z** coordinate for the **top** and **bottom** edges. We will utilize these values shortly to establish the boundaries for the ship's movement.

*Move the ship and observe the coordinates on the debug overlay.*

Based on our observation of moving the ship to the bottom-right corner and reading the coordinates 58.84496 and 32.76777 from the debug overlay, we can conclude that the boundaries can be approximated as **(-60, 60, -35, 35)**. These boundary values will remain consistent even if the game window is resized, thanks to the project settings we configured earlier in this chapter.

While the ship is currently movable using the default arrow keys, you may prefer to utilize other keys, such as the commonly used **WASD** configuration, for controlling the ship. Fortunately, this can be easily configured in the **project settings**:

1. Open **`Project > Project Settings`**, and switch to **`Input Map`**.
2. Add new actions: **move_up**, **move_left**, **move_down**, **move_right**.
3. Click the **+** (plus) icon next to each of them, and press the respective key (**W**, **A**, **S**, **D**) to assign it.

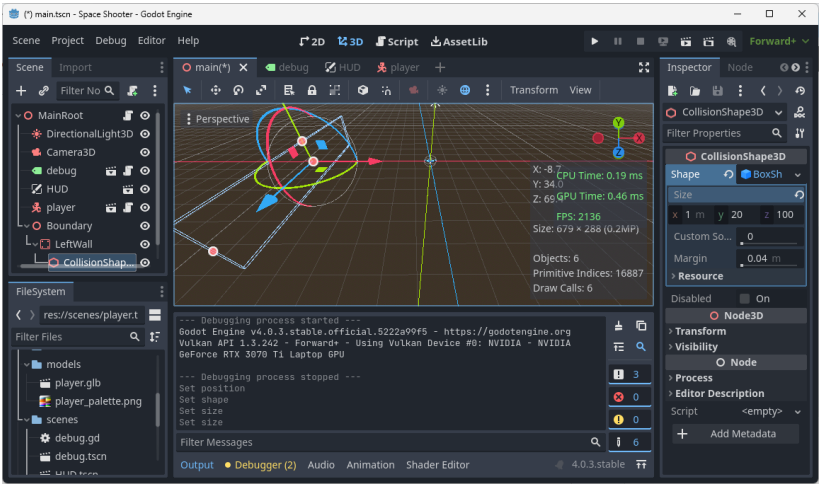



4. Close the dialog, and open the script **`player.gd`**. In the function **`_physics_process`**, find the line containing **`Input.get_vector`**, and replace the previous actions with the new ones.

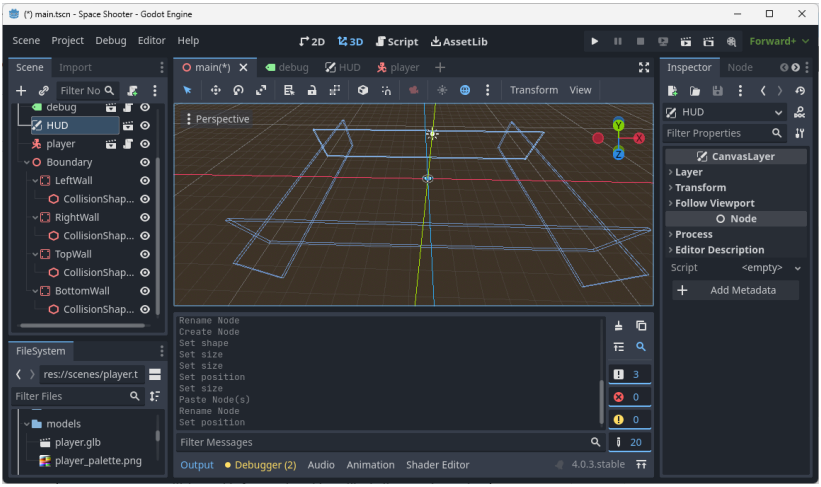
```
var input_dir = Input.get_vector("move_left", "move_right", "move_up",
"move_down")
```

After remapping the keys for ship control, there is one more important issue to address. Currently, the ship can move outside the boundaries of the screen and travel **indefinitely** in any direction. However, according to the logic of our game, it is crucial for the player's ship to remain within the visible viewport at all times. To ensure this, we need to implement **static boundaries** for the ship's movement.

1. Add a new child node **`Node3D`** to the scene **`main.tscn`**. Rename it to **Boundary**.
2. Right-click **Boundary**, and create a child node **LeftWall**. This time, the type will be **`StaticBody3D`**, as we want the player's ship (**`CharacterBody3D`**) to collide with it.
3. The node **LeftWall** doesn't have an impact on the scene until we assign a collision shape to it. Right-click **LeftWall**, and create a child node of the type **`CollisionShape3D`**.
4. Open the 3D editor, select **`LeftWall`**, and move it to the left of the ship. Observe the x-coordinate of the property **`Transform > Position`** in **`Inspector`** to make sure that you set it to the value you previously took from the debug overlay. Or even better, write the value to this field manually. In my case, it was **-60**.
5. Now, select the collision shape of the wall. In **`Inspector`**, set the shape to **`BoxShape`**, click it, and adjust the size to form a wall. I used these values:
   a. **x = 1, y = 20, z = 100**.



Good! We have successfully implemented the **left wall** to restrict the ship's movement in that direction. Now, let's repeat the same process to create three more walls: the **top**, **right**, and **bottom** boundaries. It is crucial to ensure that these walls **intersect** at the corners, leaving no gaps that would allow the ship to escape the designated playing area.




Once you have completed the creation of the four boundaries (left, top, right, and bottom), it is time to run the game and confirm that the ship is **unable to pass through** these invisible walls. Launch the game and carefully observe the ship's movement to ensure that it remains contained within the designated play area, as defined by the boundaries you have implemented.

> **Tip:** To ensure collisions with fast-moving objects like bullets are detected even in cases of a slow FPS (frames per second), it is recommended to make the invisible walls thick enough. This allows for a wider collision area and increases the chances of accurate detection. Implementing a fail-safe measure by periodically checking the bullet's position against the boundaries can be helpful. However, it is not necessary to perform this check during every frame, as it can impact performance.

We will showcase the implementation of this approach in the upcoming fourth chapter, where we introduce enemies and projectiles to the game structure.

Another fast-forward - this time to the fifth chapter - gets us to the instructions on how to utilize Godot's particle systems to build realistic flames behind the ship's jet pipes.

# Fifth: Improve the player's ship

**What we will learn in this chapter:**

1. How to rotate smoothly around one axis.
2. The axis lock and its usage.
3. How to duplicate modules to create a debris.
4. Adding the second light source to the scene.
5. Creating a shield that flashes on hit.
6. Using tweens to enhance progress bars in HUD.
7. How to leverage particles to generate jet flames.
8. Creating bullet trails with 2D elements.

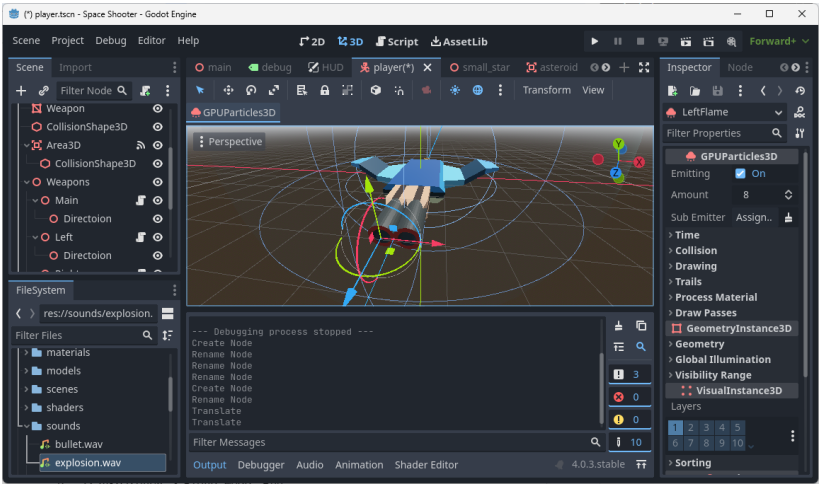
GitHub project: The end of the chapter 5

(...)

## Jet flames

While the ship's movement may not be entirely realistic, and the engines located at the bottom part of the model may not realistically propel it in all directions, it would still be visually appealing to incorporate some form of **jets**. This will allow us to display an animation with a slight touch of randomness. To achieve this effect, let's create another **particle system**.

1. Add a child node `Node3D` to the player's ship's root. Name it **Flames**.
2. Add `GPUParticles3D` as a child node to **Flames**. Call it **Left Flame**. And yes, unlike in the sparks effect, we really use the **GPU** particles here.
3. Open the 3D editor, and move this node to the end of the exhaust pipe of the left engine. Keep the y-coordinate at `0`. If the particle gizmo (the white cloud) is obstructing the view, you can disable it. In the 3D editor, find `View > Gizmos > GPUParticles3D` and deselect it.




Now that the GPUParticles3D node is connected to the player's ship, the jet flames will move in tandem with the ship's motion. However, it appears that the particles are not currently visible. To address this issue, please ensure the following steps are taken:

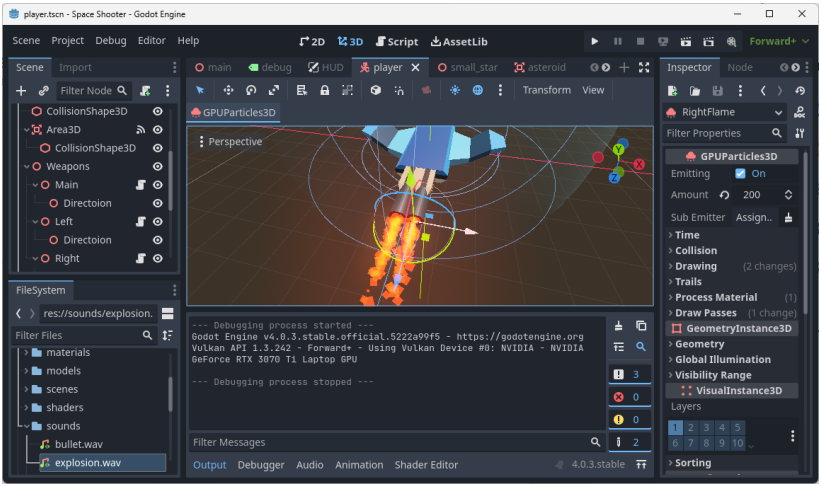
1. Click the node **Left Flame** to open the **`Inspector`**.
2. In the property **`Draw Passes`**, find **`Pass 1`**, and assign **`New QuadMesh`**.
3. Click the mesh, select **`Material`**, and create **`New StandardMaterial3D`**.
4. Click the material, and set the following properties:
    a. **`Transparency > Transparency:`** Alpha.
    b. **`Transparency > Blend Mode:`** Add.
    c. **`Shading > Shading Mode:`** Unshaded.
    d. **`Vertex Color > Use as Albedo:`** On.
    e. **`Albedo > Color`** (keep the default color).
    f. **`Billboard > Mode:`** Particle Billboard.
    g. **`Billboard > Keep Scale:`** On.
5. Scroll up to the **`GPUParticles3D`** properties, find **`Process Material`**, and assign **`New ParticleProcessMaterial`**.
6. Click the process material, and set these properties:
    a. **`Emission Shape > Shape:`** Sphere.
    b. **`Emission Shape > Sphere Radius:`** 1.
    c. **`Direction > Direction:`** (0, 0, 1).
    d. **`Direction > Spread:`** 0.

e. **`Gravity > Gravity:`** (0, 0, 0).
f. **`Initial Velocity > Velocity Min:`** 2.
g. **`Initial Velocity > Velocity Max:`** 10.
h. **`Angular Velocity > Velocity Min:`** 0.
i. **`Angular Velocity > Velocity Max:`** 40.
j. **`Linear Accel > Accel Min:`** 1.
k. **`Linear Accel > Accel Max:`** 5.
l. **`Angle > Angle Min:`** 0.
m. **`Angle > Angle Max:`** 360.
n. **`Scale > Scale Min:`** 0.1.
o. **`Scale > Scale Max:`** 1.
p. **`Scale > Scale Curve:`** New CurveTexture, add another point, and play with the shape a bit. This step is optional, I believe that the flame would look good anyway.
q. **`Color > Color Initial Ramp:`** New GradientTexture1D, and set up some fiery color gradient - red, orange, brown tints.

7. Set the property **`Amount`** to 200. You can adjust the quantity of particles emitted simultaneously to experiment and determine the ideal number required to shape the flame according to your preferences.
8. **`Time > Lifetime:`** 0.3.
9. **`Drawing > Local Coords:`** On.
10. **`Drawing > Draw Order:`** View Depth.

And behold, the flame has arrived! If desired, you can further adjust the parameters to fine-tune the effect, enhancing the color gradient or scaling it to your liking. Once satisfied, you can create the other flame by duplicating this one:

1. Right-click the node **Left Flame** and select **`Duplicate`**.
2. Rename the duplicated node to **Right Flame**.
3. In 3D editor, move the right flame to the end of the right engine.




Despite our previous recommendation to prioritize **`CPUParticles3D`**, we chose to utilize **GPU** particles for the jet flames. Why did we do that? This decision was influenced by the observation that the flames, unlike **dynamically generated** sparks, were **manually** added to the scene by creating a child node in the scene tree. Consequently, the instantiation of the flames occurs **before** the scene is activated and displayed. Interestingly, no visual problems were observed across different platforms, suggesting that we can leverage all the features of **`GPUParticles3D`** without concerns.

By modifying the **Transparency** setting in the particle's material, we must be aware that the flame may become invisible when utilizing post-processing effects on the game scene. For instance, if we replace the simple shield with a more intricate shader that introduces visual distortion to objects behind it. This limitation arises due to the sequential nature of the Godot rendering pipeline, which follows this order of operations:

1. Draws all objects with solid material (no transparency at all).
2. Applies shaders that use SCREEN_TEXTURE to read from the screen buffer (which contains only the elements from the previous step).
3. Draws the objects with transparent materials.

**Tip:** To resolve this issue, one possible solution would be to remove transparency from the particles' material, effectively bypassing steps 4a and 4b in the particle building instructions. However, it's important to note that this would result in a less visually appealing flame effect, as the quads would simply be placed on top of each other without any color blending. Additionally, this modification (with a shader material on the shield) could potentially lead to a decrease in FPS. Ultimately, the decision of whether the additional visual improvements are worth the effort is up to you to determine.

As a next sample, let me include the section from the chapter six that describes how we can add homing missiles to the pool of enemy objects.

# Sixth: Create the first level

**What we will learn in this chapter:**

1. How to use Dictionary to define the lifecycle of enemies.
2. The way to load scripts dynamically in runtime.
3. Enhancing the level system with shooting and power-ups.
4. Creating 3D models from a text in Blender.
5. How to implement a homing missile to follow the player.
6. The logic behind the boss enemies.
7. How to use threads to improve the scene transition.
8. A trick to enforce the pre-compilation of shaders.
9. How to deal with Godot 4.1 thread safety checks.

GitHub project: The end of the chapter 6

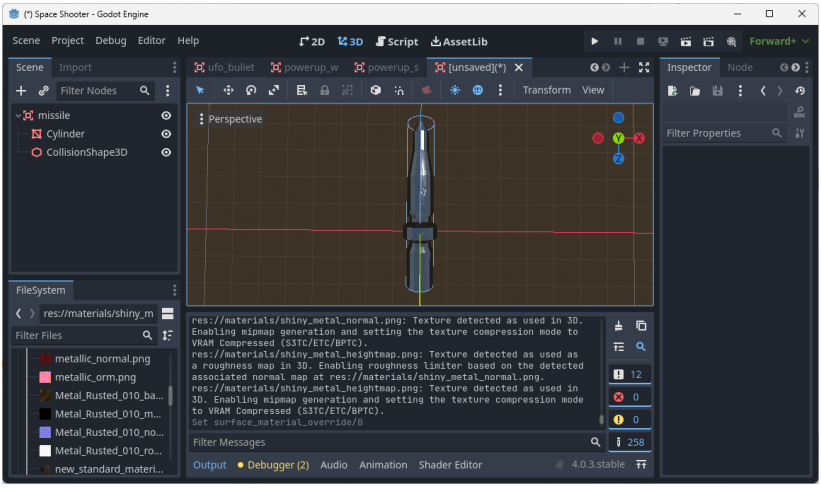
(...)

## Launch homing missiles

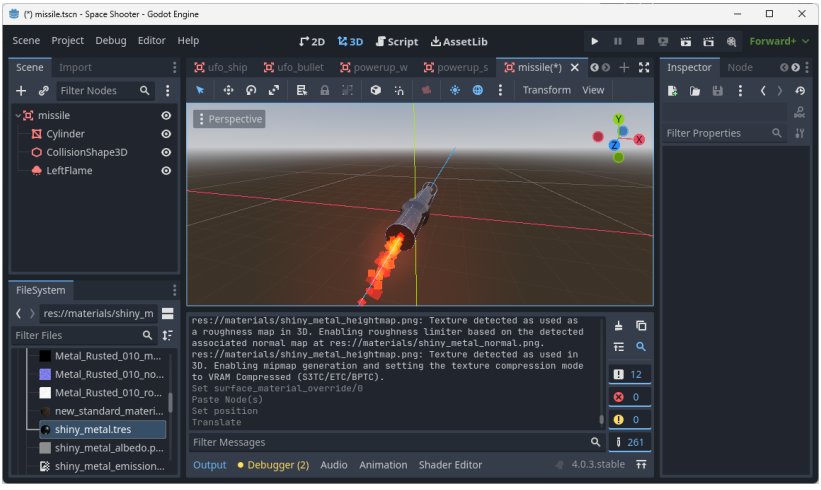
Until now, all enemies in the game have followed a **vertical** movement pattern from the top to the bottom of the screen. As a result, players have been able to **avoid collisions** with most enemies if they so desired. However, to intensify the level of excitement and introduce an element of unpredictability, we will introduce a new enemy type: the **homing missile**. Unlike other enemies, this missile will relentlessly pursue the player's ship regardless of its movements, adding a heightened sense of urgency and challenge to the gameplay.

1. Create a basic missile model in Blender by following the instructions from our previous modeling sessions. Begin by constructing a long cylinder shape and adding ring decorations to it. Then, employ common techniques such as extruding and scaling to attach a missile head to the cylinder.
2. Export the final model as usual - **`glTF 2.0`** without materials. You can reuse the model from the associated GitHub project as well.
3. Import the missile model to Godot. Create a new inherited scene, change the type to **`Area3D`**. Add a cylindrical collision shape to the scene. Select the

**`MeshInstance3D`** node, and assign a shiny metal material to **`Surface Material Override`**. Save the scene as **`missile.tscn`**.




4. Copy the **jet flame** particle node from the scene **`player.tscn`**, paste it to the missile scene. Resize the flame, and relocate it to the right position.



5. Select the root node, and attach the script **`enemy.gd`**. Open the panel **`Node > Signals`**, find the signal **`area_entered`**, and double-click it to connect to the script. Set the right groups (enemy, metal), collision layers (2) and masks (3).
6. Add the missile scene to the script **`tutorial.gd`**:

```
var missile_scene = preload("res://scenes/missile.tscn")
```

7. Within the enemy definition structure, we will add a new property called **`target`**. This property will inform the game engine that the enemy is supposed to track and follow the designated target as long as the target remains alive:

```
func get_missile_wave():
    var wave = []
    wave.append({
        "enemy": missile_scene,
        "spawn": {
            "hit_points": 20.0,
            "coords": Vector3(randf_range(-20, 20), 0,
GameManager.boundary.top),
            "scale": Vector3(1.0, 1.0, 1.0),
            "direction": Vector3(0, 0, 20.0),
            "rotation": Vector3.ZERO,
            "target": GameManager.player,
        },
        "timeline": []
    })
    return wave
```

8. Add one or more missile waves to the function **`init`** in **`tutorial.gd`**.

```
func init(node, more_scenes = []):
    timeline.append({ "timestamp": 1, "wave": get_asteroid_wave() })
    timeline.append({ "timestamp": 2, "wave": get_ufo_ship_wave() })
    timeline.append({ "timestamp": 4, "wave": get_missile_wave() })
    timeline.append({ "timestamp": 4, "wave": get_missile_wave() })
```

9. Implement the homing algorithm in the script **`lifecycle.gd`**:
    a. Save the initial speed to a new variable in **`init`**.
    b. Assign **`spawn.target`** to a new variable **`target`**.
    c. If **`target`** exists and is valid, recalculate all parameters in every frame.

d. Compute **`to_target`** vector and use it to update **`current_direction`** (to continue moving in this direction after the target was destroyed).
e. Calculate the direction **`angle`** (relative to **`Vector2.UP`**) and rotate the missile around y-axis.
f. Utilize the **`move_toward`** function to update the position of the missile at a consistent speed. This approach is simpler and less susceptible to errors compared to using **`lerp`** for the same purpose.

```
var target
var speed

func init(root_node, enemy, _spawn, _timeline):
(...)
    speed = current_direction.length()
    if spawn.has("target"):
        target = spawn.target

func process(enemy, delta):
(...)
    if Utils.is_valid_node(target):
        var to_target = target.global_position - enemy.global_position
        current_direction = to_target.normalized() * speed
        var direction_angle = Vector2(to_target.x,
to_target.z).angle_to(Vector2.UP)
        enemy.rotation.y = direction_angle
        enemy.global_position =
enemy.global_position.move_toward(target.global_position, delta * speed)
    else:
        enemy.global_position.x += current_direction.x * delta
        enemy.global_position.z += current_direction.z * delta
```

Take a look at the game. Two missiles will emerge at the upper boundary of the screen after four seconds, and they will track the player's ship until they are eliminated.


Finally, we switch to the seventh chapter, namely the part about important system settings inside the game - gamma correction, vertical synchronization, and audio mixer.

# Seventh: Game workflow

**What we will learn in this chapter:**

1. How to share WorldEnvironment settings across scenes.
2. The basic 2D layout of controls.
3. Creating simple transitions by rotating a light source.
4. How to pause and resume the game.
5. A trick to pause shaders by a trick with global uniforms.
6. How to set up, read, and write settings.
7. The audio mixer and working with multiple channels.
8. How to localize your game.
9. Using toggle buttons to redefine control keys.
10. Working with AnimationPlayer to create the intro scene.
11. How to play music in the background.

GitHub project: The end of the chapter 7

(...)

## Gamma correction

As we are aware, different computers may have varying **brightness** and **contrast** settings, which poses a risk of a scene appearing excellent on our own device but barely visible on others. Considering this situation, it is essential to provide the player with the option to adjust **gamma correction**. Specifically, we will be modifying the **`Exposure`** value of the environment's **`Tonemap`** feature, as it closely resembles the concept of gamma correction and is convenient to manipulate.

Since relying on mere estimations and hoping for the game to appear as expected is challenging, it is important to incorporate a mechanism for instantly observing changes. There are two approaches to accomplish this:

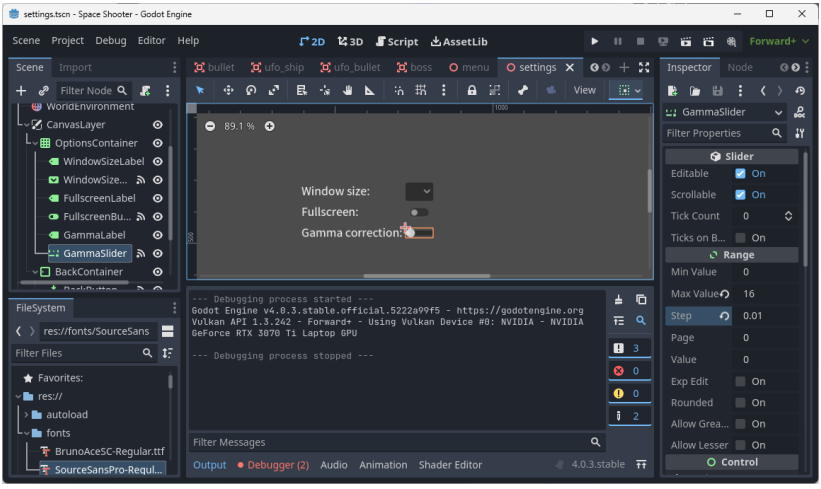
1. Add the **`Exposure`** slider to the **game** screen, and allow the player to adjust it during the gameplay.
2. Display several game assets (ships, asteroids) on the **settings** screen to see the **`Exposure`** changes on them.

In this tutorial, we will implement the second option.

1. Open the scene **`settings.tscn`**, and drag **`asteroid.tscn`** and **`ufo_ship.tscn`** to the scene root as child nodes.
2. Adjust their coordinates (**`Node3D > Transform > Position`**) to display the asteroid to the left and the ship to the right of the centered **OptionsContainer**.
3. Add this code to the script **`settings.gd`** to make the assets slowly rotate around their respective centers:

```
@onready var asteroid = $asteroid
@onready var ufo_ship = $ufo_ship

func _ready():
(...)
    spawn_asteroid()
    spawn_ufo()

func spawn_asteroid():
    var spawn = {
        "hit_points": 20.0,
        "coords": Vector3(0, 0, 0),
        "scale": Vector3(1, 1, 1),
        "direction": Vector3.ZERO,
        "rotation": Utils.get_random_vector3_in_range(0.1, 1.0),
    }
    asteroid.init(self, spawn, [])

func spawn_ufo():
    var spawn = {
        "hit_points": 20.0,
        "coords": Vector3(0, 0, 0),
        "scale": Vector3(1, 1, 1),
        "direction": Vector3.ZERO,
        "rotation": Vector3(0, 0, 0.2),
    }
    ufo_ship.init(self, spawn, [])
```

4. Add two more elements to **OptionsContainer**:
    a. **`Label:`** GammaLabel
    b. **`HSlider:`** GammaSlider
5. Select **GammaSlider** to open **`Inspector`**, and set the property **`Max Value`** to **16** (which is the actual max value of the exposure). Also, change the property **`Step`** to **0.01** to allow fine-grained changes.

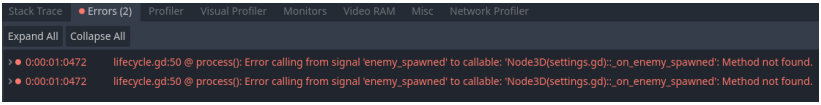



6. Finally, find the signal **`value_changed`** in **`Node > Signals`**, and connect it to a new function in **`settings.gd`**. Add this code to the script:

```
@onready var world_environment = $WorldEnvironment
@onready var gamma_slider = $CanvasLayer/OptionsContainer/GammaSlider

func _ready():
(...)
    gamma_slider.value = options.tonemap_exposure if
options.has("tonemap_exposure") else 1.0

func _on_gamma_slider_value_changed(value):
    options.tonemap_exposure = gamma_slider.value
    OptionsManager.write_options(options)
    world_environment.environment.tonemap_exposure =
options.tonemap_exposure
```

Start the game, click **`Settings`**, drag the bar **`Gamma Correction`**, and observe the changes on the asteroid and the enemy ship.

By the way, you might have noticed that the game wrote two errors to the Errors console:




The reason behind this behavior is the inclusion of two game objects, namely the asteroid and UFO ship, within the **Settings** scene. However, the signal **`enemy_spawned`** lacks a connection to any function, as **`_on_enemy_spawned`** is not implemented in the **`settings.gd`** script. To address this, we can resolve the issue by adding a dummy method to the script:

```
func _on_enemy_spawned(_enemy):
    pass
```

## Vertical synchronization

**V-Sync** (Vertical Synchronization) is a technique used in computer graphics to synchronize the frame rate of a game with the refresh rate of the display. In Godot 4, there are **four** different V-Sync modes available, each with its own pros and cons:

1. VSYNC_DISABLED
    a. Pros: Disabling V-Sync allows the game to run at the **maximum frame rate** supported by the hardware, resulting in smoother and more responsive gameplay.
    b. Cons: Without V-Sync, **screen tearing** can occur, where the display shows parts of multiple frames simultaneously, leading to a disjointed visual experience.
2. VSYNC_ENABLED
    a. Pros: Enabling V-Sync eliminates screen tearing by synchronizing the frame rate with the **display's refresh rate**. This results in a visually coherent and tear-free image.
    b. Cons: V-Sync can introduce **input lag**, as the game waits for the display to refresh before presenting the next frame. This can reduce responsiveness, especially in fast-paced games that require precise timing.
3. VSYNC_ADAPTIVE
    a. Pros: Adaptive V-Sync **dynamically adjusts** the V-Sync behavior based on the current frame rate. If the frame rate is below the display's refresh rate,

V-Sync is enabled to prevent tearing. If the frame rate exceeds the refresh rate, V-Sync is temporarily disabled to reduce input lag.

   b. Cons: Adaptive V-Sync can introduce **occasional screen tearing** when the frame rate fluctuates near the refresh rate threshold. It also adds some complexity to the rendering pipeline, which may result in a slight performance overhead.

4. VSYNC_MAILBOX
   a. Pros: V-Sync Mailbox displays the **most recently rendered image** in the queue during vertical blanking intervals, while the rendering process continues for other images. This technique effectively eliminates screen tearing, ensuring a smooth visual experience.
   b. Cons: By rendering frames as quickly as possible, this mode can potentially reduce input lag, also referred to as the "Fast" V-Sync mode. However, it's important to note that this reduction in input lag is **not guaranteed** and can vary depending on the specific hardware and software configuration.

The initial mode is taken from **`Project > Project Settings > General > Display > Window > V-Sync`**. The default value should be VSYNC_ENABLED.

Many games provide only a **disabled/enabled switch**, and only these two options were available in Godot 3. Since we work with Godot 4, we can enable all **four values** to set in Settings. It would be helpful to provide a brief description of each option for users who may not be familiar with V-Sync and its effects on gameplay. This way, users can make informed decisions based on their preferences and requirements.

Okay, that's a lot of theory. From our perspective, the most straightforward approach would be to keep the default value VSYNC_ENABLED until the player deliberately changes it. Let's add the corresponding elements to the UI:

1. Add the following code to the script **`options_manager.gd`**:

```
var v_sync_list = [
    { "mode": DisplayServer.VSYNC_DISABLED, "label":
"settings.vsync.disabled" },
    { "mode": DisplayServer.VSYNC_ENABLED, "label": "settings.vsync.enabled"
},
    { "mode": DisplayServer.VSYNC_ADAPTIVE, "label":
"settings.vsync.adaptive" },
    { "mode": DisplayServer.VSYNC_MAILBOX, "label": "settings.vsync.mailbox"
},
```

```
]
(...)

func set_v_sync_mode():
    var options = read_options()
    if not options.has("v_sync"):
        options.v_sync = DisplayServer.VSYNC_ENABLED
    DisplayServer.window_set_vsync_mode(options.v_sync)
    write_options(options)
```

2. As evident, we utilize localization keys such as "settings.vsync.enabled" instead of directly embedding English labels into the code. This approach enables us to prepare the text for localization into different languages. We will delve into this topic further in this chapter.

3. Add this line to _ready of the entry scene script (**`menu.gd`**):

```
    OptionsManager.set_v_sync_mode()
```

4. Open the scene **`settings.tscn`**, and add the following elements as child nodes of **`OptionsContainer`**:
    a. **`Label:`** VSyncLabel
    b. **`OptionButton:`** VSyncOptionButton
5. Select VSyncOptionButton, and connect the signal **`item_selected`** from the panel **`Node > Signals`**.
6. Open the script **`settings.gd`**, and add the following code to handle the changes and display current values when the scene is active:

```
@onready var v_sync_option_button =
$CanvasLayer/OptionsContainer/VSyncOptionButton
(...)

func _ready():
(...)
    v_sync_option_button.clear()
    index = 0
    for v_sync in OptionsManager.v_sync_list:
        v_sync_option_button.add_item(v_sync.label)
        if v_sync.mode == options.v_sync:
            v_sync_option_button.select(index)
        index += 1

func _on_v_sync_option_button_item_selected(index):
```

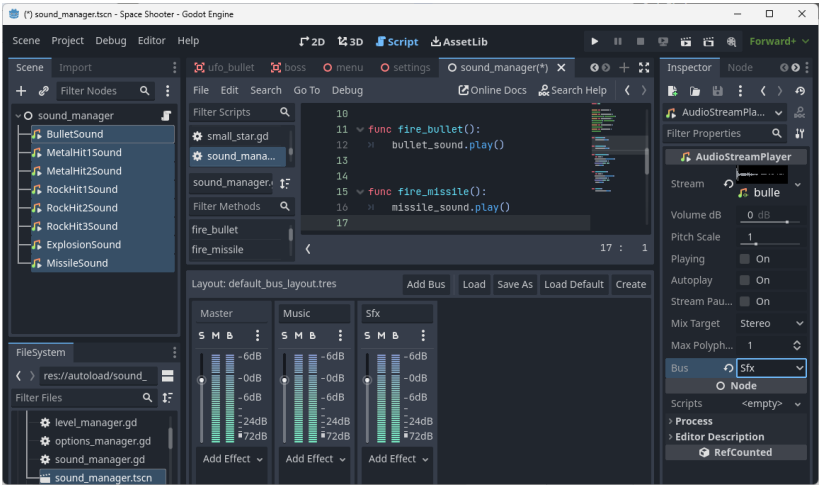
```
    var v_sync = OptionsManager.v_sync_list[index]
    options.v_sync = v_sync.mode
    OptionsManager.write_options(options)
    OptionsManager.set_v_sync_mode()
```

Run the project, visit the **Settings** screen, and change the **V-Sync** option to **settings.vsync.disabled**. After returning to the menu screen, proceed to start the game. Upon doing so, you will notice that the FPS (frames per second) displayed in the debug overlay is no longer limited by the refresh rate of your monitor. In the case of my gaming laptop, the game exhibited a frame rate exceeding 1000 frames per second.


## Audio mixer

Godot's **audio system** offers great flexibility, allowing us to create multiple channels known as **buses**, which serve as pathways for sound output. Through code, we have the ability to control these buses. We can adjust their volume settings, add sound effects, or even disable them to mute the respective channel entirely. In our case, we will configure two buses: one for **music** and another for **sound effects**. We will then bind these buses to their corresponding sliders in the **`Settings`** menu, providing players with the ability to adjust the volume levels as desired.

1. Switch to the tab **`Audio`** at the bottom page of the Godot editor. You should see the master bus.
2. Click **`Add Bus`** twice to add two more buses. Rename the first one to **Music** and the second one to **Sfx**.
3. Open the scene **`sound_manager.tscn`**, and route all game sounds through the **Sfx** bus. Select all **`AudioStreamPlayer`** nodes in the scene tree (**`Shift-Click`**), find the property **`Bus`** in **`Inspector`**, and change the value to **Sfx**.




4. Open the script **`sound_manager.gd`**, and add this code:

```
var master_bus
var music_bus
var sfx_bus
(...)

func _ready():
    master_bus = AudioServer.get_bus_index("Master")
    music_bus = AudioServer.get_bus_index("Music")
    sfx_bus = AudioServer.get_bus_index("Sfx")

func set_master_volume(value):
    # the value is between 0 and 1
    AudioServer.set_bus_volume_db(master_bus, linear_to_db(value))

func set_music_volume(value):
```

```
    AudioServer.set_bus_volume_db(music_bus, linear_to_db(value))

func set_sfx_volume(value):
    AudioServer.set_bus_volume_db(sfx_bus, linear_to_db(value))
```

5. An audio bus uses a **decibel scale** (logarithmic), so we need to calculate it from linear values before applying. Godot provides a function called **`linear_to_db`**, which is precisely what we've just used here.
6. Open the scene **`settings.tscn`**, add three more labels with slider options - **Master volume**, **Music volume**, **Sfx volume**. Set the following parameters for all of them in **`Inspector`**:
    a. **`Min Value:`** 0
    b. **`Max Value:`** 1
    c. **`Step:`** 0.01
    d. **`Value:`** 1
7. Go to **`Node > Signals`**, and connect the signal **`value_changed`** to a new function in **`settings.gd`** for all three sliders. Add this code to the script:

```
@onready var master_volume_slider =
$CanvasLayer/OptionsContainer/MasterVolumeSlider
@onready var music_volume_slider =
$CanvasLayer/OptionsContainer/MusicVolumeSlider
@onready var sfx_volume_slider =
$CanvasLayer/OptionsContainer/SfxVolumeSlider
(...)

func _ready():
(...)
    master_volume_slider.value = options.master_volume if
options.has("master_volume") else 1.0
    music_volume_slider.value = options.music_volume if
options.has("music_volume") else 1.0
    sfx_volume_slider.value = options.sfx_volume if
options.has("sfx_volume") else 1.0

func _on_master_volume_slider_value_changed(value):
    options.master_volume = value
    OptionsManager.write_options(options)
    SoundManager.set_master_volume(value)

func _on_music_volume_slider_value_changed(value):
    options.music_volume = value
    OptionsManager.write_options(options)
```

```
    SoundManager.set_music_volume(value)

func _on_sfx_volume_slider_value_changed(value):
    options.sfx_volume = value
    OptionsManager.write_options(options)
    SoundManager.set_sfx_volume(value)
```

Run the game, and try to change **Sfx volume** to a position in the middle of the slider:


Then go back to the main menu, and start the game. The sound effects should be much quieter than before.

> **Tip:** Currently, we do not implement any positional audio; we simply play the sound files as they were originally recorded. However, if you choose to position the sound effects in the 3D space, the following approach can be utilized:
>
> 1. Replace **`AudioStreamPlayer`** nodes with **`AudioStreamPlayer3D`**.
> 2. To ensure proper spatial location of a sound source, you can accomplish it by attaching an instance of **`AudioStreamPlayer3D`** as a child node to the specific node representing the sound's origin, such as the

**explosion** or **bullet** node. This way, when the sound is played, it will be associated with and emanate from the corresponding node in the virtual space.

3. If you find that the auditory experience doesn't meet your expectations, it is probable that the original sound file was recorded in **stereo**, which can cause conflicts with the 3D sound system. To resolve this issue, you can convert the sound file to mono. Several free tools, such as **Audacity**, are available to assist you with this conversion process.

(...)

You have reached the end of the excerpt from the book "Godot 4: From zero to full game." I hope that what you have read so far has piqued your interest, and you would like to learn more about game development in the Godot Engine.

Please visit my portfolio at https://Filip.Rachunek.com to learn how you can purchase the full book, or email me at Filip.Rachunek@gmail.com.

Good luck with developing your games!