

Unofficial Bevy Cheat Book

This is a reference-style book for the [Bevy game engine](#) (GitHub).


It aims to teach Bevy concepts in a concise way, help you be productive, and discover the knowledge you need.

This book aggregates a lot of community wisdom that is often not covered by official documentation, saving you the need to struggle with issues that others have figured out already!

While it aims to be exhaustive, documenting an entire game engine is a monumental task. I focus my time on whatever I believe the community needs most.

Therefore, there are still a lot of omissions, both for basics and advanced topics. Nevertheless, I am confident this book will prove to be a valuable resource to you!

Welcome! May this book serve you well!

(don't forget to  the book's [GitHub repository](#), and consider [donating](#) 😊)

How to use this book

The pages in this book are not designed to be read in order. Each page covers a standalone topic. Feel free to jump to whatever interests you.

If you have a specific topic in mind that you would like to learn about, you can find it from the table-of-contents (sidebar) or using the search function (in the top bar).

The [Chapter Overview](#) page will give you a general idea of how the book is structured.

If you are new to Bevy, or would like a more guided experience, try the [tutorial page](#). It will help you navigate the book in an order that makes sense for learning, from beginner to advanced topics.

The [Bevy Builtins](#) page is a concise cheatsheet of useful information about types and features provided by Bevy.

Recommended Additional Resources

Bevy has a rich collection of [official code examples](#).

Check out [bevy-assets](#), for community-made resources.

Our community is very friendly and helpful. Feel welcome to join the [Bevy Discord](#) to chat, ask questions, or get involved in the project!


If you want to see some games made with Bevy, see [itch.io](#) or [Bevy Assets](#).

Maintenance

This version of the book is for Bevy release 0.7.

I intend to keep this book up-to-date and relevant with every new Bevy release. I also try to regularly make improvements to it, when I can manage it.

Support Me


 GitHub Sponsors

I'd like to keep improving and maintaining this book, to provide a high-quality independent learning resource for the Bevy community.

Your donation helps me work on such freely-available content. Thank you! ❤️

Support Bevy

If you like the Bevy Game Engine, you should consider donating to the official project.

 GitHub Sponsors

License

Copyright © 2021-2022 Ida Iyes.

All code in the book is provided under the [MIT-0 License](#). At your option, you may also use it under the regular MIT License.

The text of the book is provided under the [CC BY-NC-SA 4.0](#).

Exception: If used for the purpose of contribution to the "Official Bevy Project", the entire content of the book may be used under the [MIT-0 License](#).

"Official Bevy Project" is defined as:

- Any files contained in the Git repository hosted at <https://github.com/bevyengine/bevy>
- Any files contained in the Git repository hosted at <https://github.com/bevyengine/bevy-website>
- Anything publicly visible on the bevyengine.org website

Contributions

Development of this book is hosted on [GitHub](#).

Please file GitHub Issues for any wrong/confusing/misleading information, as well as suggestions for new content you'd like to be added to the book.

Contributions are accepted, with some limitations.

See the [Contributing](#) section for all the details.

Stability Warning

Bevy is still a very new and experimental game engine! It has only been public since August 2020!

While improvements have been happening at an incredible pace, and development is active, Bevy simply hasn't yet had the time to mature.

There are no stability guarantees and breaking changes happen often!

Usually, it not hard to adapt to changes with new releases (or even track the main git development branch), but you have been warned!

Chapter Overview

This book is organized into a number of different chapters, covering different aspects of working with Bevy. The is designed to be useful as a reference and learning tool, so you can jump to what interests you and learn about it.

If you would like a more guided tutorial-like experience, or to browse the book by relative difficulty (from beginner to advanced), try the [guided tutorial page](#). It recommends topics in a logical order for learning.

The [Bevy Builtins](#) page is a concise cheatsheet of useful information about types and features provided by Bevy.

The book has the following general chapters:

- [Bevy Setup Tips](#): project setup advice, recommendations for tools and plugins
- [Common Pitfalls](#): solutions for common issues encountered by the community
- [Bevy Programming Framework](#): how to write code in Bevy (foundational knowledge)
- [Programming Patterns](#): opinionated advice, patterns, idioms
- [\[WIP\] Bevy Render \(GPU\) Framework](#): working with the GPU and Bevy's rendering
- [Bevy Cookbook](#): miscellaneous code examples beyond the ones in Bevy official repos
- [Bevy on Different Platforms](#): information about working with specific plaforms / OSs

The following chapters are focused on covering specific Bevy feature areas:

- [Bevy Game Engine Core](#): important features that don't belong in any other chapter
- [Bevy Asset Management](#): working with data assets
- [Input Handling](#): working with different input devices
- [Window Management](#): working with the OS window
- [Bevy 2D](#): Bevy's features for 2D games
- [Bevy 3D](#): Bevy's features for 3D games

New to Bevy? Guided Tutorial!

Welcome to Bevy! :) We are glad to have you in our community!

Make sure to also look at [the official Bevy examples](#). If you need help, use [GitHub Discussions](#), or feel welcome to come chat and ask for help in [Discord](#).

This page is intended for new learners. It will guide you through this book in an order that makes sense for learning: from the basics, towards more advanced topics. This is unlike the main table-of-contents (the left sidebar), which was designed to be a reference for Bevy users of any skill level.

This tutorial page does not list/link every page in the book. It is a guide to help you gain comprehensive general knowledge. The book also has many pages dedicated to solutions for specific problems; those are not listed here.

Feel free to jump around the book and read whatever interests you.

You will be making something cool with Bevy in no time! ;)

If you run into issues, be sure to check the [Common Pitfalls](#) chapter, to see if this book has something to help you. Solutions to some of the most common issues that Bevy community members have encountered are documented there.

Basics

These are the absolute essentials of using Bevy – the minimum concepts to get you started. Every Bevy project, even a simple one, would require you to be familiar with these concepts.

You could conceivably make something like a simple game-jam game or prototype, using just this knowledge. Though, as your project grows, you will likely quickly need to learn more.

- [Bevy Setup Tips](#): Configuring your development tools and environment
 - [Getting Started](#)
- [Bevy Programming Framework](#): How to write Bevy code, structure your data and logic
 - [Intro to ECS](#)
 - [Entities and Components](#)
 - [Resources](#)
 - [Systems](#)
 - [App Builder](#)
 - [Queries](#)
 - [Commands](#)
 - [Events](#)
- [Bevy Game Engine Core](#): Basic features of Bevy, needed for making any game
 - [Coordinate System](#)
 - [Transforms](#)
 - [Time and Timers](#)
- [Bevy Asset Management](#): How to work with assets
 - [Handles](#)

- [Load Assets with AssetServer](#)
- [Input Handling](#): Using various input devices
- [Window Management](#): Setting up the OS Window (or fullscreen) for your game
 - [Change the Background Color](#)

Next Steps

You will likely need to learn about at least some of these topics, to make a non-trivial Bevy project. After you are confident with the basics, you can familiarize yourself with these, to become a proficient Bevy user.

- [Bevy Setup Tips](#)
 - [Bevy Dev Tools and Editors](#)
 - [Community Plugin Ecosystem](#)
- [Bevy Programming Framework](#)
 - [System Order of Execution](#)
 - [System Sets](#)
 - [Local Resources](#)
 - [Plugins](#)
 - [Labels](#)
 - [States](#)
 - [Change Detection](#)
 - [Query Sets](#)
 - [Stages](#)
- [Programming Patterns](#)
 - [Generic Systems](#)
 - [Component Storage](#)
- [Bevy Asset Management](#):
 - [Access the Asset Data](#)
 - [React to Changes with Asset Events](#)
 - [Hot-Reloading Assets](#)

Advanced

These are more specialized topics, may be useful in complex projects. Most typical Bevy users are unlikely to need to know these.

- [Bevy Programming Framework](#)
 - [Run Criteria](#)
 - [Removal Detection](#)
 - [System Chaining](#)
 - [Direct World Access](#)
 - [Exclusive Systems](#)
 - [Non-Send](#)
- [Programming Patterns](#)

- [Manual Event Clearing](#)

Solutions to Specific Problems

These are pages that teach you solutions to specific tasks that you might encounter in your project.

- [Convert cursor to world coordinates](#)
- [Write tests for systems](#)
- [Track asset loading progress](#)
- [Grab/Capture the Mouse Cursor](#)
- [Set the Window Icon](#)
- [Input Text](#)
- [Drag-and-Drop files](#)
- [Custom Camera Projection](#)

List of Bevy Builtins

This page is a quick condensed listing of all the important things provided by Bevy.

- `SystemParams`
- `Assets`
- `File Formats`
- `wgpu Backends`
- `Bundles`
- `Resources (Configuration)`
- `Resources (Engine User)`
- `Resources (Input)`
- `Events (Input)`
- `Events (System/Control)`
- `Components`
- `GLTF Asset Labels`
- `Stages`

SystemParams

These are all the special types that can be used as `system` parameters.

(List in API Docs)

- `Commands` : Manipulate the ECS using `commands`
- `Res<T>` : Shared access to a `resource`
- `ResMut<T>` : Exclusive (mutable) access to a `resource`
- `Option<Res<T>>` : Shared access to a resource that may not exist
- `Option<ResMut<T>>` : Exclusive (mutable) access to a resource that may not exist
- `Query<T, F = ()>` (can contain tuples of up to 15 types): Access to `entities and components`
- `ParamSet` (with up to 8 params): Resolve [conflicts between incompatible system parameters]
[cb::paramset]
- `Local<T>` : Data `local` to the system
- `EventReader<T>` : Receive `events`
- `EventWriter<T>` : Send `events`
- `RemovedComponents<T>` : Removal detection
- `NonSend<T>` : Shared access to `Non-Send` (main thread only) data
- `NonSendMut<T>` : Mut access to `Non-Send` (main thread only) data
- `&World` : Read-only `direct access to the ECS World`
- `Entities` : Low-level ECS metadata: All entities
- `Components` : Low-level ECS metadata: All components
- `Bundles` : Low-level ECS metadata: All bundles
- `Archetypes` : Low-level ECS metadata: All archetypes
- `SystemChangeTick` : Low-level ECS metadata: Tick used for change detection
- tuples containing any of these types, with up to 16 members

Your function can have a maximum of 16 total parameters. If you need more, group them into tuples to work around the limit. Tuples can contain up to 16 members, but can be nested

Indefinitely.

Assets

These are the Asset types registered by Bevy by default.

- `Image` : Pixel data, used as a texture for 2D and 3D rendering; also contains the `SamplerDescriptor` for texture filtering settings
- `TextureAtlas` : 2D "Sprite Sheet" defining sub-images within a single larger image
- `Mesh` : 3D Mesh (geometry data), contains vertex attributes (like position, UVs, normals)
- `Shader` : GPU shader code, in one of the supported languages (WGSL/SPIR-V/GLSL)
- `ColorMaterial` : Basic "2D material": contains color, optionally an image
- `StandardMaterial` : "3D material" with support for Physically-Based Rendering
- `AnimationClip` : Data for a single animation sequence, can be used with `AnimationPlayer`
- `Font` : Font data used for text rendering
- `Scene` : Scene composed of literal ECS entities to instantiate
- `DynamicScene` : Scene composed with dynamic typing and reflection
- `Gltf` : **GLTF Master Asset**: index of the entire contents of a GLTF file
- `GltfNode` : Logical GLTF object in a scene
- `GltfMesh` : Logical GLTF 3D model, consisting of multiple `GltfPrimitive`s
- `GltfPrimitive` : Single unit to be rendered, contains the Mesh and Material to use
- `AudioSource` : Raw audio data for `bevy_audio`
- `AudioSink` : Audio that is currently active, can be used to control playback
- `FontAtlasSet` : (internal use for text rendering)
- `SkinnedMeshInverseBindposes` : (internal use for skeletal animation)

File Formats

These are the asset file formats (asset loaders) supported by Bevy. Support for each one can be enabled/disabled using [cargo features](#). Some are enabled by default, many are not.

Image formats (loaded as `Image` assets):

Format	Cargo feature	Default?	Filename extensions
PNG	"png"	Yes	.png
HDR	"hdr"	Yes	.hdr
JPEG	"jpeg"	No	.jpg , .jpeg
TGA	"tga"	No	.tga
BMP	"bmp"	No	.bmp
DDS	"dds"	No	.dds
KTX2	"ktx2"	No	.ktx2

Format	Cargo feature	Default?	Filename extensions
Basis	"basis-universal"	No	.basis

Audio formats (loaded as `AudioSource` assets):

Format	Cargo feature	Default?	Filename extensions
OGG Vorbis	"vorbis"	Yes	.ogg
FLAC	"flac"	No	.flac
WAV	"wav"	No	.wav
MP3	"mp3"	No	.mp3

3D asset (model or scene) formats:

Format	Cargo feature	Default?	Filename extensions
GLTF	"bevy_gltf"	Yes	.gltf, .glb

Shader formats (loaded as `Shader` assets):

Format	Cargo feature	Default?	Filename extensions
SPIR-V	n/a	Yes	.spv
WGSL	n/a	Yes	.wgsl
GLSL	n/a	Yes	.vert, .frag

Font formats (loaded as `Font` assets):

Format	Cargo feature	Default?	Filename extensions
TrueType	n/a	Yes	.ttf
OpenType	n/a	Yes	.otf

There are [unofficial plugins](#) available for adding support for even more file formats.

wgpu Backends

`wgpu` (and hence Bevy) supports the following backends for each platform:

- Vulkan (Linux/Windows/Android)
- DirectX 12 (Windows)
- Metal (Apple)
- WebGL2 (Web)
- WebGPU (Web; experimental)
- GLES3 (Linux/Android; legacy)
- DirectX 11 (Windows; legacy; WIP (not yet ready for use))

Bundles

Bevy's built-in [bundle](#) types, for spawning different common kinds of entities.

(List in API Docs)

Any tuples of up to 15 [Component](#) types are valid bundles.

General:

- [TransformBundle](#) : Contains the [transform](#) types [Transform](#) and [GlobalTransform](#) to enable using the entity in a [parent-child hierarchy](#)

Bevy 3D:

- [PerspectiveCameraBundle](#) : 3D camera with a perspective projection
- [OrthographicCameraBundle](#) : Camera with an orthographic projection, 2D or 3D
- [MaterialMeshBundle](#) : 3D Object/Primitive: a Mesh and the Material to draw it with
- [PbrBundle](#) : 3D object with the standard Physically-Based Material
([MaterialMeshBundle](#)<[StandardMaterial](#)>)
- [DirectionalLightBundle](#) : 3D directional light (like the sun)
- [PointLightBundle](#) : 3D point light (like a lamp or candle)

Bevy 2D:

- [OrthographicCameraBundle](#) : Camera with an orthographic projection, 2D or 3D
- [SpriteBundle](#) : 2D sprite, using a whole image ([Image](#) asset)
- [SpriteSheetBundle](#) : 2D sprite, using a sub-rectangle in a larger image ([TextureAtlas](#) asset)
- [MaterialMesh2dBundle](#) : 2D shape, with custom Mesh and Material (similar to 3D objects)
- [Text2dBundle](#) : Text to be drawn in the 2D world (not the UI)

Bevy UI:

- [UiCameraBundle](#) : The UI Camera
- [NodeBundle](#) : Empty node element (like HTML `<div>`)
- [ButtonBundle](#) : Button element
- [ImageBundle](#) : Image element
- [TextBundle](#) : Text element

Resources

Configuration Resources

These resources allow you to change the settings for how various parts of Bevy work.

Some of them affect the low-level initialization of the engine, so must be present from the start to take effect. You need to insert these at the start of your [app builder](#):

- [LogSettings](#) : Configure what messages get logged to the console
- [WindowDescriptor](#) : Settings for the primary application window

- `WgpuSettings` : Low-level settings for the GPU API and backends
- `AssetServerSettings` : Configuration of the `AssetServer`
- `DefaultTaskPoolOptions` : Settings for the CPU task pools (multithreading)
- `WinitSettings` : Settings for the OS Windowing backend, including update loop / power-management settings

These may be inserted at the start, but should also be fine to change at runtime (from a `system`):

- `ClearColor` : Global renderer background color to clear the window at the start of each frame
- `AmbientLight` : Global renderer "fake lighting", so that shadows don't look too dark / black
- `Msa` : Global renderer setting for Multi-Sample Anti-Aliasing (some platforms might only support the values 1 and 4)
- `ClusterConfig` : Configuration of the light clustering algorithm, affects the performance of 3D scenes with many lights
- `WireframeConfig` : Global toggle to make everything be rendered as wireframe
- `GamepadSettings` : Gamepad input device settings, like joystick deadzones and button sensitivities

Engine Resources

These resources provide access to different features of the game engine at runtime.

Access them from your `systems`, if you need their state, or to control the respective parts of Bevy.

- `FixedTimesteps` : The state of all registered `FixedTimestep` drivers
- `Time` : Global time-related information (current frame delta time, time since startup, etc.)
- `AssetServer` : Control the asset system: Load assets, check load status, etc.
- `Gamepads` : List of IDs for all currently-detected (connected) gamepad devices
- `Windows` : All the open windows (the primary window + any additional windows in a multi-window gui app)
- `WinitWindows` ([non-send][cb::non-send]): Raw state of the `winit` backend for each window
- `Audio` : Use this to play sounds via `bevy_audio`
- `AsyncComputeTaskPool` : Task pool for running background CPU tasks
- `ComputeTaskPool` : Task pool where the main app schedule (all the systems) runs
- `IoTaskPool` : Task pool where background i/o tasks run (like asset loading)
- `Diagnostics` : Diagnostic data collected by the engine (like frame times)
- `SceneSpawner` : Direct control over spawning Scenes into the main app World
- `TypeRegistryArc` : Access to the Reflection Type Registry
- `AdapterInfo` : Information about the GPU hardware that Bevy is running on

Input Handling Resources

These resources represent the current state of different input devices. Read them from your `systems` to [handle user input][cb::input].

- `Input<KeyCode>` : Keyboard key state, as a binary `Input` value
- `Input<MouseButton>` : Mouse button state, as a binary `Input` value

- `Input<GamepadButton>` : Gamepad buttons, as a binary `Input` value
- `Axis<GamepadAxis>` : Analog `Axis` gamepad inputs (joysticks and triggers)
- `Axis<GamepadButton>` : Gamepad buttons, represented as an analog `Axis` value
- `Touches` : The state of all fingers currently touching the touchscreen

Events

Input Events

These `events` fire on activity with input devices. Read them to `[handle user input][cb::input]`.

- `MouseButtonInput` : Changes in the state of mouse buttons
- `MouseWheel` : Scrolling by a number of pixels or lines (`MouseScrollUnit`)
- `MouseMotion` : Relative movement of the mouse (pixels from previous frame), regardless of the OS pointer/cursor
- `CursorMoved` : New position of the OS mouse pointer/cursor
- `KeyboardInput` : Changes in the state of keyboard keys (keypresses, not text)
- `ReceivedCharacter` : Unicode text input from the OS (correct handling of the user's language and layout)
- `TouchInput` : Change in the state of a finger touching the touchscreen
- `GamepadEvent` : Changes in the state of a gamepad or any of its buttons or axes
- `GamepadEventRaw` : Gamepad events unaffected by `GamepadSettings`

System and Control Events

Events from the OS / windowing system, or to control Bevy.

- `RequestRedraw` : In an app that does not refresh continuously, request one more update before going to sleep
- `AppExit` : Tell Bevy to shut down
- `CloseWindow` : Tell Bevy to close a window
- `CreateWindow` : Tell Bevy to open a new window
- `FileDragAndDrop` : The user drag-and-dropped a file into our app
- `CursorEntered` : OS mouse pointer/cursor entered one of our windows
- `CursorLeft` : OS mouse pointer/cursor exited one of our windows
- `WindowCloseRequested` : OS wants to close one of our windows
- `WindowCreated` : New application window opened
- `WindowFocused` : One of our windows is now focused
- `WindowMoved` : OS/user moved one of our windows
- `WindowResized` : OS/user resized one of our windows
- `WindowScaleFactorChanged` : One of our windows has changed its DPI scaling factor
- `WindowBackendScaleFactorChanged` : OS reports change in DPI scaling factor for a window

Components

The complete list of individual component types is too specific to be useful to list here.

See: [\(List in API Docs\)](#)

Curated/opinionated list of the most important built-in component types:

- **Transform** : Local transform (relative to parent, if any)
- **GlobalTransform** : Global transform (in the world)
- **Parent** : Entity's parent, if in a hierarchy
- **Children** : Entity's children, if in a hierarchy
- **Handle<T>** : Reference to an asset of specific type
- **Visibility** : Manually control visibility, whether to display the entity (hide/show)
- **RenderLayers** : Group entities into "layers" and control which "layers" a camera should display
- **AnimationPlayer** : Make the entity capable of playing animations; used to control animations
- **Camera** : Camera used for rendering
- **CameraUi** : Marker to identify the camera used for the UI render pass
- **Camera2d** : Marker to identify the camera used for the main 2D render pass
- **Camera3d** : Marker to identify the camera used for the main 3D render pass
- **OrthographicProjection** : Orthographic projection for a camera
- **PerspectiveProjection** : Perspective projection for a camera
- **Sprite** : (2D) Properties of a sprite, using a whole image
- **TextureAtlasSprite** : (2D) Properties of a sprite, using a sprite sheet
- **PointLight** : (3D) Properties of a point light
- **DirectionalLight** : (3D) Properties of a directional light
- **NoFrustumCulling** : (3D) Cause this mesh to always be drawn, even when not visible by any camera
- **NotShadowCaster** : (3D) Disable entity from producing dynamic shadows
- **NotShadowReceiver** : (3D) Disable entity from having dynamic shadows of other entities
- **Wireframe** : (3D) Draw object in wireframe mode
- **Node** : (UI) Mark entity as being controlled by the UI layout system
- **Style** : (UI) Layout properties of the node
- **Interaction** : (UI) Track interaction/selection state: if the node is clicked or hovered over
- **UiImage** : (UI) Image to be displayed as part of a UI node
- **UiColor** : (UI) Color to use for a UI node
- **Button** : (UI) Marker for a pressable button
- **Text** : Text to be displayed

GLTF Asset Labels

Asset path labels to refer to GLTF sub-assets.

The following asset labels are supported (`{}` is the numerical index):

- `Scene{}` : GLTF Scene as Bevy `Scene`
- `Node{}` : GLTF Node as `GltfNode`
- `Mesh{}` : GLTF Mesh as `GltfMesh`
- `Mesh{}/Primitive{}` : GLTF Primitive as Bevy `Mesh`
- `Texture{}` : GLTF Texture as Bevy `Image`
- `Material{}` : GLTF Material as Bevy `StandardMaterial`
- `DefaultMaterial` : as above, if the GLTF file contains a default material with no index
- `Animation{}` : GLTF Animation as Bevy `AnimationClip`
- `Skin{}` : GLTF mesh skin as Bevy `SkinnedMeshInverseBindposes`

Stages

Internally, Bevy has at least these built-in `stages`:

- In the `main app` (`CoreStage`): `First` , `PreUpdate` , `Update` , `PostUpdate` , `Last`
- In the render `sub-app` (`RenderStage`): `Extract` , `Prepare` , `Queue` , `PhaseSort` , `Render` , `Cleanup`

Bevy Setup Tips

This chapter is a collection of additional tips for configuring your project or development tools, collected from the Bevy community, beyond what is covered in Bevy's [official setup documentation](#).

Feel free to suggest things to add under this chapter.

Also see the following other relevant content from this book:

- [Platform-specific information](#)
- [Configuration to fix slow performance](#)

This page covers the basic setup needed for Bevy development.

For the most part, Bevy is just like any other Rust library. You need to install Rust and setup your dev environment just like for any other Rust project. You can install Rust using [Rustup](#). See [Rust's official setup page](#).

On Linux, you need the development files for some system libraries. See the [official Bevy Linux dependencies page](#).

Also see the [Setup page in the official Bevy Book](#) and the [official Bevy Readme](#).

GPU Drivers

On Linux, Bevy currently requires Vulkan for graphics.

On Windows, either Vulkan or DirectX 12 can be used.

Make sure you have compatible hardware and drivers installed on your system. Your users will also need to satisfy this requirement.

If Bevy is not working, install the latest drivers, or check with your Linux distribution whether Vulkan needs additional packages to be installed.

OpenGL and DirectX 11 support for legacy systems is planned, but not available yet.

macOS should work without any special driver setup, using Metal.

Web games are supported and should work in any modern browser, using WebGL2.

Creating a New Project

You can simply create a new Rust project, either from your IDE/editor, or the commandline:

```
cargo new --bin my_game
```

(creates a project called `my_game`)

The `Cargo.toml` file contains all the configuration of your project. Add the latest version of `bevy` as a dependency. Your file should now look something like this:

```
[package]
name = "my_game"
version = "0.1.0"
```

```
version = "0.1.0"
edition = "2021"

[dependencies]
bevy = "0.7"
```

The `src/main.rs` file is your main source code file. This is where you start writing your Rust code. For a minimal Bevy app, you need at least the following:

```
use bevy::prelude::*;

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .run();
}
```

You can now compile and run your project. The first time, this will take a while, as it needs to build the whole Bevy engine and dependencies. Subsequent runs should be fast. You can do this from your IDE/editor, or the commandline:

```
cargo run
```

Optional Extra Setup

You will likely quickly run into unusably slow performance with the default Rust unoptimized dev builds. [See here how to fix.](#)

Also, iterative recompilation speed is important to keep you productive, so you don't have to wait long for the Rust compiler to rebuild your project every time you want to test your game. [Bevy's getting started page](#) has advice about how to speed up compile times.

Also have a look in the [Dev Tools and Editors](#) page for suggestions about additional external dev tools that may be helpful.

What's Next?

Have a look at the [guided tutorial](#) page of this book, and Bevy's [official examples](#).

Check out the [Bevy Assets Website](#) to find other tutorials and learning resources from the community, and plugins to use in your project.

Join the community on [Discord](#) to chat with us!

Running into Issues?

If something is not working, be sure to check the [Common Pitfalls](#) chapter, to see if this book has something to help you. Solutions to some of the most common issues that Bevy community members have encountered are documented there.

If you need help, use [GitHub Discussions](#), or feel welcome to come chat and ask for help in [Discord](#).

Using bleeding-edge Bevy (bevy main)

Bevy development moves very fast, and there are often exciting new things that are yet unreleased. This page will give you advice about using development versions of bevy.

Quick Start

If you are *not* using any 3rd-party plugins and just want to use the bevy main development branch:

```
[dependencies]
bevy = { git = "https://github.com/bevyengine/bevy" }
```

However, if you *are* working with external plugins, you should read the rest of this page. You will likely need to do more to make everything compatible.

Should you use bleeding-edge Bevy?

Currently, Bevy does not make patch releases (with rare exceptions for critical bugs), only major releases. The latest release is often missing the latest bug fixes, usability improvements, and features. It may be compelling to join in on the action!

If you are new to Bevy, this might not be for you; you might be more comfortable using the released version. It will have the best compatibility with community plugins and documentation.

The biggest downside to using unreleased versions of Bevy is 3rd-party plugin compatibility. Bevy is unstable and breaking changes happen often. However, many actively-maintained community plugins have branches for tracking the latest Bevy main branch, although they might not be fully up-to-date. It's possible that a plugin you want to use does not work with the latest changes in Bevy main, and you may have to fix it yourself.

The frequent breaking changes might not be a problem for you, though. Thanks to cargo, you can update bevy at your convenience, whenever you feel ready to handle any possible breaking changes.

If you choose to use Bevy main, you are highly encouraged to interact with the Bevy community on [Discord](#) and [GitHub](#), so you can keep track of what's going on, get help, or participate in discussions.

Common pitfall: mysterious compile errors

When changing between different versions of Bevy (say, transitioning an existing project from the released version to the git version), you might get lots of strange unexpected build errors.

You can typically fix them by removing `Cargo.lock` and the `target` directory:

```
rm -rf Cargo.lock target
```

See [this page](#) for more info. See this [cargo issue](#) about this bug.

If you are still getting errors, it is probably because cargo is trying to use multiple different versions of bevy in your dependency tree simultaneously. This can happen if some of the plugins you use have specified a different Bevy version/commit from your project.

Make sure you use the correct branch of each plugin you depend on, with support for Bevy main.

If you have issues, they might still be fixable. Read the next section below for advice on how to configure your project in a way that minimizes the chances of this happening.

How to use bleeding-edge bevy?

```
[dependencies]
# recommended: specify a known-working commit hash to pin to
# (specify it in the URL, to make the patch tricks below work)
bevy = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" }

# add any 3rd-party plugins you use, and make sure to use the correct branch
# (alternatively, you could also specify a commit hash, with "rev")
bevy_thing = { git = "https://github.com/author/bevy_thing?branch=bevy_main" }

# For each plugin we use, patch them to use the same bevy commit as us:

# If they have specified a different commit:
# (you need to figure this out)
[patch."https://github.com/bevyengine/bevy?rev=146123ea"]
bevy = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" }

# For those that have not specified anything:
[patch."https://github.com/bevyengine/bevy"]
bevy = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" }
```

Some 3rd-party plugins depend on specific bevy sub-crates, rather than the full bevy. You may additionally have to patch those individually:

```
[patch."https://github.com/bevyengine/bevy"]
# specific crates as needed by the plugins you use (check their `Cargo.toml`)
bevy_ecs = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" }
bevy_math = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" }
# ... and so on
```

To collect all the information you need, in order to fully patch all your dependencies, you can either look at their `Cargo.toml`, or figure it out by running `cargo tree` or searching inside your `Cargo.lock` file for duplicate entries (multiple copies of bevy crates).

Make sure to delete `Cargo.lock` every time you make a change to your dependencies configuration, to force cargo to resolve everything again.

Updating Bevy

It is recommended that you specify a known-good Bevy commit in your `Cargo.toml`, so that you can be sure that you only update it when you actually want to do so, avoiding unwanted breakage.

```
bevy = { git = "https://github.com/bevyengine/bevy?rev=7a1bd34e" }
```

Even if you do not, the `Cargo.lock` file always keeps track of the exact version (including the git commit) you are working with. You will not be affected by new changes in upstream bevy or plugins, until you update it.

To update, run:

```
cargo update
```

or delete `Cargo.lock`.

Make sure you do this every time you change the configuration in your `Cargo.toml`. Otherwise you risk errors from cargo not resolving dependencies correctly.

Advice for plugin authors

If you are publishing a plugin crate, here are some recommendations:

- Have a separate branch in your repository, to keep support for bevy main separate from your main version for the released version of bevy
- Put information in your README to tell people how to find it
- Set up CI to notify you if your plugin is broken by new changes in bevy

Feel free to follow all the advice from this page, including cargo patches as needed. They only apply when you build your project directly, not as a dependency, so they do not affect your users.

CI Setup

Here is an example for GitHub Actions. This will run at 8:00 AM (UTC) every day to verify that your code still compiles. GitHub will notify you when it fails.

```
name: check if code still compiles
```

```
on:
  schedule:
    - cron: '0 8 * * *'

env:
  CARGO_TERM_COLOR: always

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Install Dependencies
        run: sudo apt-get update && sudo apt-get install --no-install-recommends pkg-
config libx11-dev libasound2-dev libudev-dev

      - uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
          override: true

      - name: Check code
        run: cargo update && cargo check --lib --examples
```

This page contains tips for different text editors and IDEs.

Bevy is, for the most part, like any other Rust project. If your editor/IDE is set up for Rust, that might be all you need. This page contains additional information that may be useful for Bevy specifically.

Please help improve this page by providing suggestions for things to add.

CARGO_MANIFEST_DIR

When running your app/game, Bevy will search for the `assets` folder in the path specified in the `CARGO_MANIFEST_DIR` environment variable. This allows `cargo run` to work correctly from the terminal.

If you are using your editor/IDE to run your project in a non-standard way (say, inside a debugger), you have to be sure to set that correctly.

VSCode

Here is a snippet showing how to create a run configuration for debugging Bevy (with lldb):

(this is for development on Bevy itself, and testing with the `breakout` example)

(adapt to your needs if using for your project)

```
{
  "type": "lldb",
  "request": "launch",
  "name": "Debug example 'breakout'",
  "cargo": {
    "args": [
      "build",
      "--example=breakout",
      "--package=bevy"
    ],
    "filter": {
      "name": "breakout",
      "kind": "example"
    }
  },
  "args": [],
  "cwd": "${workspaceFolder}",
  "env": {
    "CARGO_MANIFEST_DIR": "${workspaceFolder}",
  }
}
```

Dev Tools and Editors for Bevy

Bevy does not yet have an official editor or other such tools. An official editor is planned as a long-term future goal. In the meantime, here are some community-made tools to help you.

Editor

`bevy_inspector_egui` gives you a simple editor-like property inspector window in-game. It lets you modify the values of your components and resources in real-time as the game is running.

`bevy_editor_pls` is an editor-like interface that you can embed into your game. It has even more features, like switching app states, fly camera, performance diagnostics, and inspector panels.

Diagnostics

`bevy_mod_debugdump` is a tool to help visualize your [App Schedule](#) (all of the registered [systems](#) with their [ordering dependencies](#) and [stages](#)), and the Bevy Render Graph.

`bevy_lint` is a linter (based on `dylint`) that can automatically check your Bevy code for some common issues.

If you are getting confusing/cryptic compiler error messages (like [these](#)) and you cannot figure them out, `bevycheck` is a tool you could use to help diagnose them. It tries to provide more user-friendly Bevy-specific error messages.

Community Plugins Ecosystem

There is a growing ecosystem of unofficial community-made plugins for Bevy. They provide a lot of functionality that is not officially included with the engine. You might greatly benefit from using some of these in your projects.

To find such plugins, you should search the [Bevy Assets](#) page on the official Bevy website. This is the official registry of known community-made things for Bevy. If you publish your own plugins for Bevy, you should [contribute a link to be added to that page](#).

Please beware that some 3rd-party plugins may use unusual licenses. Be sure to check the license before using a plugin in your project.

Other pages in this book with valuable information when using 3rd-party plugins:

- Some plugins may require you to [configure Bevy in some specific way](#).
- If you are [using bleeding-edge unreleased Bevy \(main\)](#), you may encounter difficulties with plugin compatibility.

Plugin Recommendations

This here is my personal, curated, opinionated list of recommendations, featuring the most important plugins (in my opinion) in the Bevy ecosystem.

My goal here is to help direct new Bevy users to some known-good resources, so you can start working on the kinds of games you want to make. :)

The plugins listed here are compatible with the current Bevy release and use permissive licenses (like Bevy itself).

This page is limited. I can only recommend plugins I know enough about. Please also check the [Bevy Assets](#) page to find even more things. :)

Development Tools and Editors

These are listed on a [separate page](#).

Code Helpers

`bevy_asset_loader` is a more flexible and opinionated helper for managing and loading [assets] [cb::assets]. Uses custom syntax to let you declare your assets more conveniently.

`iyes_loopless` provides alternative improved implementations of [states](#), [run criteria](#), and [fixed timestep](#), that do not suffer from the major usability limitations of the ones provided with Bevy.

Input Mappings

To help with your game's [input handling](#) needs, try the [Input Manager plugin](#) by Leafwing Studios. It is a very flexible way to handle your game's bindings / mappings.

Audio

Use `bevy_kira_audio` instead of the built-in `bevy_audio`.

The built-in audio is very limited in features, and you are likely going to need this plugin for pretty much any game with audio.

See [this page](#) for help on how to set it up.

Camera

`bevy_config_cam` is a nice plugin for easily adding camera controls to your Bevy 3D project. It gives you a choice of various common camera behaviors (like follow, top-view, FPS-style, free-roaming).

Cameras are something that can be very game-specific. As you progress with your project, you would probably want to implement your own custom camera control logic for your game. However, this plugin is amazing when you are starting out on a new project.

Tilemap

If you are making a 2D game based on a tile-map, there are plugins to help do it efficiently with high performance. It is better to use one of these plugins, instead of just spawning lots of individual Bevy sprites for each tile.

`bevy_ecs_tilemap`:

- Uses one ECS Entity per tile, lets you work with the tilemap in an ECS-idiomatic way.
- Very efficient rendering, using techniques like texture arrays, chunks, morton encoding, ...
- Lots of features: Square/Hexagon/Isometric grids, animation, layers, chunks, ...

[`bevy_ecs_ldtk`] [`project::bevy_ecs_ldtk`] implements loading of entire maps/levels created with the LDTK editor, into Bevy. Based on `bevy_ecs_tilemap` internally, for efficient performance.

Shapes / Vector Graphics / Canvas

If you want to draw 2D shapes, use the `bevy_prototype_lyon` plugin.

Game AI

`big-brain` is a plugin for game AI behaviors (Utility AI).

GUI

There are a few alternatives to Bevy UI available.

`bevy_egui` integrates the `egui` toolkit into Bevy. It is a mature immediate-mode GUI library (like the popular Dear ImGui, but in Rust). It is very feature-rich and provides lots of widgets. It was not really designed for making flashy gamey UIs (though it may very well be fine for your game). It's great for editor-like UIs, debug UIs, or non-game applications.

`kayak_ui` is a new experimental game-centric UI library for Bevy, which uses a XML-like declarative syntax for constructing UIs.

`ui4` is another notable experimental UI library for Bevy.

UI Navigation

If you are using the builtin Bevy UI, there is a nice plugin available for navigation (moving between buttons and other focusable UI elements): `bevy-ui-navigation`.

Physics

Bevy can integrate with the [Rapier physics engine](#).

There are two plugins you can choose from:

- `heron`
 - Idiomatic to Bevy. Nice user-friendly integration and workflow.
 - Likely to be easier to use and more intuitive than `bevy_rapier`.
 - May have more limited functionality.
- `bevy_rapier`
 - This is a "raw" plugin that gives you direct access to Rapier.
 - Gives you the most control, but may be hard to use and not idiomatic-Bevy.
 - You will probably need to read a lot of documentation, harder to learn.

Animation

Starting from Bevy 0.7, there is built-in support for playing predefined asset-driven animations, including 3D skeletal animation.

However, for "programmatic" / code-driven animation, you may need something else. Try `bevy_tweening`. This might be good enough for moving objects around, moving the camera, smoothly changing colors, or other such transitions.

For animated 2D sprites, try `benimator`. This is for using sprite-sheet assets with many frames of animation.

File Formats

Additional asset loaders, for loading assets from file formats other than [those that Bevy officially supports](#).

- Wavefront OBJ 3D models: `bevy_obj`
- STL 3D models: `bevy_stl`
- MagicaVoxel VOX: `bevy_vox`

Bevy is very modular and configurable. It is implemented as many separate cargo crates, allowing you to remove the parts you don't need. Higher-level functionality is built on top of lower-level foundational crates, and can be disabled or replaced with alternatives.

The lower-level core crates (like the Bevy ECS) can also be used completely standalone, or integrated into otherwise non-Bevy projects.

Bevy Cargo Features

In Bevy projects, you can enable/disable various parts of Bevy using cargo features.

Many common features are enabled by default. If you want to disable some of them, note that, unfortunately, Cargo does not let you disable individual default features, so you need to disable all default bevy features and re-enable the ones you need.

Here is how you might configure your Bevy:

```
[dependencies.bevy]
version = "0.7"
# Disable the default features if there are any that you do not want
```

```

# Disable the default features if there are any that you do not want
default-features = false
features = [
    # These are the default features:
    # (re-enable whichever you like)

    # Bevy functionality:
    "animation",          # Animation support
    "bevy_gilrs",          # Gamepad input support
    "bevy_audio",          # Builtin audio
    "bevy_winit",          # Window management
    "x11",                 # Linux: Support X11 windowing system
    "filesystem_watcher", # Asset hot-reloading
    "render",              # Graphics Rendering

    ## "render" actually just includes:
    ## (feel free to use just a subset of these, instead of "render")
    "bevy_render",         # Rendering framework core
    "bevy_core_pipeline",  # Higher-level rendering abstractions
    "bevy_sprite",         # 2D (sprites) rendering
    "bevy_pbr",            # 3D (physically-based) rendering
    "bevy_gltf",           # GLTF 3D assets format support
    "bevy_text",           # Text/font rendering
    "bevy_ui",             # UI toolkit

    # File formats:
    "png",
    "hdr",
    "vorbis",

    # These are other features that may be of interest:
    # (add any of these that you need)

    # Bevy functionality:
    "wayland",             # Linux: Support Wayland windowing system
    "subpixel_glyph_atlas", # Subpixel antialiasing for text/fonts
    "serialize",           # Support for `serde` Serialize/Deserialize

    # File formats:
    "ktx2", # preferred format for GPU textures
    "dds",
    "jpeg",
    "bmp",
    "tga",
    "basis-universal",
    "zstd", # needed if using zstd in KTX2 files
    "flac",
    "mp3",
    "wav",

    # Development/Debug features:
    "dynamic",             # Dynamic linking for faster compile-times
    "trace",               # Enable tracing for performance measurement
    "trace_tracy",         # Tracing using `tracy`
    "trace_chrome",        # Tracing using the Chrome format
    "wgpu_trace",          # WGPU/rendering tracing
]

```

(See [here](#) for a full list of Bevy's cargo features.)

Graphics / Rendering

For a graphical application or game (most Bevy projects), you can include `render` and `bevy_winit`. For [Linux](#) support, you need at least one of `x11` or `wayland`.

However, `render` is a meta-feature; it simply enables all the graphics-related features of Bevy. If you want, you can strip it down and include only what you need.

`bevy_render` and `bevy_core_pipeline` are required for any application using Bevy rendering.

If you only need 2D and no 3D, add `bevy_sprite`.

If you only need 3D and no 2D, add `bevy_pbr`. If you are [loading 3D models from GLTF files](#), add `bevy_gltf`.

If you are using Bevy UI, you need `bevy_text` and `bevy_ui`.

If you don't need any graphics (like for a dedicated game server, scientific simulation, etc.), you may remove all of these features.

Audio

Bevy's audio is very limited in functionality. It is recommended that you use the `bevy_kira_audio` plugin instead. Disable `bevy_audio` and `vorbis`.

See [this page](#) for more information.

File Formats

You can use the relevant cargo features to enable/disable support for loading assets with various different file formats.

See [here](#) for more information.

Input Devices

If you do not care about [gamepad \(controller/joystick\)](#) support, you can disable `bevy_gilrs`.

Linux Windowing Backend

On [Linux](#), you can choose to support X11, Wayland, or both. Only `x11` is enabled by default, as it is the legacy system that should be compatible with most/all distributions, to make your builds smaller and compile faster. You might want to additionally enable `wayland`, to fully and natively support modern Linux environments. This will add a few extra transitive dependencies to your project.

Asset hot-reloading

The `filesystem_watcher` feature controls support for [hot-reloading of assets](#), supported on desktop platforms.

Development Features

While you are developing your project, these features might be useful:

Dynamic Linking

`dynamic` causes Bevy to be built and linked as a shared/dynamic library. This will make incremental builds *much* faster.

This is only supported on desktop platforms. Known to work very well on Linux, Windows/macOS should work, but might have issues.

Do not enable this for release builds you intend to publish to other people; it introduces unneeded complexity (you need to bundle extra files) and potential for things to not work correctly. Use this only during development.

For this reason, it may be convenient to specify the feature as a commandline option to cargo, instead of putting it in your `Cargo.toml`. Simply run your project like this:

```
cargo run --features bevy/dynamic
```

Tracing

The features `trace` and `wgpu_trace` may be useful for profiling and diagnosing performance issues.

`trace_chrome` and `trace_tracy` choose the backend you want to use to visualize the traces.

Common Pitfalls

This chapter covers some common issues or surprises that you might be likely to encounter when working with Bevy, with specific advice about how to address them.

Strange Build Errors

Sometimes, you can get strange and confusing build errors when trying to compile your project.

Update your Rust

First, make sure your Rust is up-to-date. When using Bevy, you must use at least the latest stable version of Rust (or nightly).

If you are using `rustup` to manage your Rust installation, you can run:

```
rustup update
```

Clear the cargo state

Many kinds of build errors can often be fixed by forcing `cargo` to regenerate its internal state (recompute dependencies, etc.). You can do this by deleting the `Cargo.lock` file and the `target` directory.

```
rm -rf target Cargo.lock
```

Try building your project again after doing this. It is likely that the mysterious errors will go away.

This trick often fixes the broken build, but if it doesn't help you, your issue might require further investigation. Reach out to the Bevy community via GitHub or [Discord](#), and ask for help.

If you are using bleeding-edge Bevy ("main"), and the above does not solve the problem, your errors might be caused by 3rd-party plugins. See [this page](#) for solutions.

New Cargo Resolver

Cargo recently added a new dependency resolver algorithm, that is incompatible with the old one. Bevy *requires* the new resolver.

If you are just creating a new blank Cargo project, don't worry. This should already be setup correctly by `cargo new`.

If you are getting weird compiler errors from Bevy dependencies, read on. Make sure you have the correct configuration, and then [clear the cargo state](#).

Single-Crate Projects

In a single-crate project (if you only have one `Cargo.toml` file in your project), if you are using the latest Rust2021 Edition, the new resolver is automatically enabled.

So, you need either one of these settings in your `Cargo.toml` :

```
[package]
edition = "2021"
```

or

```
[package]
resolver = "2"
```

Multi-Crate Workspaces

In a multi-crate Cargo workspace, the resolver is a global setting for the whole workspace. It will *not* be enabled by default.

This can bite you if you are transitioning a single-crate project into a workspace.

You *must* add it manually to the top-level `Cargo.toml` for your Cargo Workspace:

```
[workspace]
resolver = "2"
```

What errors?

One common example is the "failed to select a version" error, which can look something like this:

```
error: failed to select a version for `web-sys`.
... required by package `wgpu v0.9.0`
which is depended on by `bevy v0.5.0`
```

```
... which is depended on by `bevy_wgpu v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy_internal v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy-scratchpad v0.1.0 (C:\Users\Alice\Documents\bevy-scratchpad)`  
versions that meet the requirements `=0.3.50` are: 0.3.50
```

all possible versions conflict with previously selected packages.

```
previously selected package `web-sys v0.3.46`  
... which is depended on by `bevy_app v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy_asset v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy_audio v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy_internal v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy v0.5.0`  
(https://github.com/bevyengine/bevy#6a8a8c9d)`  
... which is depended on by `bevy-scratchpad v0.1.0 (C:\Users\Alice\Documents\bevy-scratchpad)`
```

failed to select a version for `web-sys` which could resolve this conflict

(there are many variations, yours might not be identical to the example above)

Another related error are seemingly-nonsensical compiler messages about conflicts with Bevy's internal types (like "expected type `Transform`, found type `Transform`").

Why does this happen?

Such errors are often caused by `cargo`'s internal state being broken. Usually, it is because of dependencies not being resolved properly, causing cargo to try to link multiple versions of Bevy into your project. This often occurs when transitioning your project between the release and the git version of Bevy. Cargo remembers the versions it was previously using, and gets confused.

See this [cargo issue](#) about this bug. If you have any interesting information to add, you can help by contributing to that issue.

Performance

Unoptimized debug builds

You can enable compiler optimizations in debug/dev mode!

Even `opt-level=1` is enough to make Bevy not painfully slow! You can also enable higher optimizations for dependencies, but not your own code, to keep recompilations fast!

```
# in `Cargo.toml` or `.cargo/config.toml`  
  
# Enable only a small amount of optimization in debug mode  
[profile.dev]  
opt-level = 1  
  
# Enable high optimizations for dependencies (incl. Bevy), but not for our code:  
[profile.dev.package."*"]  
opt-level = 3
```

Why is this necessary?

Rust without compiler optimizations is *very slow*. With Bevy in particular, the default cargo build debug settings will lead to *awful* runtime performance. Assets are slow to load and FPS is low.

Common symptoms:

- Loading high-res 3D models with a lot of large textures, from GLTF files, can take minutes! This can trick you into thinking that your code is not working, because you will not see anything on the screen until it is ready.
- After spawning even a few 2D sprites or 3D models, framerate may drop to unplayable levels.

Why not use `--release`?

You may have heard the advice: just run with `--release`! However, this is bad advice. Don't do it.

Release mode also disables "debug assertions": extra checks useful during development. Many libraries also include additional stuff under that setting. In Bevy and WGPU that includes validation for shaders and GPU API usage. Release mode disables these checks, causing less-informative crashes, issues with hot-reloading, or potentially buggy/invalid logic going unnoticed.

Release mode also makes incremental recompilation slow. That negates Bevy's fast compile times, and can be very annoying while you develop.

With the advice at the top of this page, you don't need to build with `--release`, just to test your game with adequate performance. You can use it for *actual* release builds that you send to your users.

If you want, you can also enable LTO (Link Time Optimization) for the actual release builds, to

if you want, you can also enable LTO (Link-Time-Optimization) for the actual release builds, to squeeze out even more performance at the cost of very slow compile times:

```
[profile.release]  
lto = "thin"
```

Error adding function as system

You can sometimes get confusing arcane compiler errors when you try to add systems to your Bevy app.

The errors can look like this:

```
the trait bound `for<'r, 's, 't0> fn(bevy::prelude::Query<'r, 's, (&'t0 Param))
{my_system}: IntoSystem<(), (), _>` is not satisfied
```

This is caused by your function having incompatible parameters. Bevy can only accept special types as system parameters.

You might also errors that look like this:

```
the trait bound `Component: WorldQuery` is not satisfied
the trait `WorldQuery` is not implemented for `Component`
```

```
this struct takes at most 2 type arguments but 3 type arguments were supplied
```

These errors are caused by a malformed query.

Common beginner mistakes

- Using `&mut Commands` (bevy 0.4 syntax) instead of `Commands`.
- Using `Query<MyStuff>` instead of `Query<&MyStuff>` or `Query<&mut MyStuff>`.
- Using `Query<&ComponentA, &ComponentB>` instead of `Query<(&ComponentA, &ComponentB)>` (forgetting the tuple)
- Using your resource types directly without `Res` or `ResMut`.
- Using your component types directly without putting them in a `Query`.
- Using other arbitrary types in your function.

Note that `Query<Entity>` is correct, because the Entity ID is special; it is not a component.

Supported types

Only the following types are supported as system parameters:

(List in [API Docs](#))

- `Commands`: Manipulate the ECS using [commands](#)
- `Res<T>`: Shared access to a [resource](#)
- `ResMut<T>`: Exclusive (mutable) access to a [resource](#)
- `Option<Res<T>>`: Shared access to a resource that may not exist
- `Option<ResMut<T>>`: Exclusive (mutable) access to a resource that may not exist
- `Query<T, F = ()>` (can contain tuples of up to 15 types): Access to [entities and components](#)
- `ParamSet` (with up to 8 params): Resolve [conflicts between incompatible system parameters]

[cb::paramset]

- `Local<T>` : Data `local` to the system
- `EventReader<T>` : Receive `events`
- `EventWriter<T>` : Send `events`
- `RemovedComponents<T>` : Removal detection
- `NonSend<T>` : Shared access to `Non-Send` (main thread only) data
- `NonSendMut<T>` : Mut access to `Non-Send` (main thread only) data
- `&World` : Read-only `direct access to the ECS World`
- `Entities` : Low-level ECS metadata: All entities
- `Components` : Low-level ECS metadata: All components
- `Bundles` : Low-level ECS metadata: All bundles
- `Archetypes` : Low-level ECS metadata: All archetypes
- `SystemChangeTick` : Low-level ECS metadata: Tick used for change detection
- tuples containing any of these types, with up to 16 members

Your function can have a maximum of 16 total parameters. If you need more, group them into tuples to work around the limit. Tuples can contain up to 16 members, but can be nested indefinitely.

I can't see my UI!

If you are trying to build a UI, but it is not showing on the screen, you probably forgot to spawn a UI Camera. The UI Camera is required for Bevy to render UI.

```
commands.spawn_bundle(UiCameraBundle::default());
```

2D objects not displaying

Bevy's 2D [coordinate space](#) is set up so that your background can be at $Z=0.0$, and other sprite layers can be at positive $+Z$ coordinates above that.

Therefore, to see your scene, the camera needs to be placed far away, at a large $+Z$ coordinate, looking towards $-Z$.

If you are overriding the camera [transform](#) / creating your own transform, *you need to do this!* The default transform (with $Z=0.0$) will place the camera so that your sprites (at positive $+Z$ coordinates) would be behind the camera, and you wouldn't see them! You need to either set a large Z coordinate, or preserve/copy the Z value from the [Transform](#) that is generated by Bevy's builtin Bundle constructor (`OrthographicCameraBundle::new_2d()`).

By default, when you create a 2D camera using Bevy's built-in Bundle constructor (`OrthographicCameraBundle::new_2d()`), Bevy sets the camera [Transform](#) to have $Z=999.9$. This is close to the default clipping plane (visible range of Z axis), which is set to 1000.0 .

3D objects not displaying

This page will list some common issues that you may encounter, if you are trying to spawn a 3D object, but cannot see it on the screen.

Missing Vertex Attributes

Make sure your `Mesh` includes all vertex attributes required by your shader/material.

Bevy's default PBR `StandardMaterial` requires all meshes to have:

- Positions
- Normals
- UVs (even if there are no textures / just a solid color)
- Tangents (if using normal maps, otherwise not required)

If you are generating your own mesh data, make sure to include all of the above.

If you are loading it from asset files, make sure they include everything that is needed. In particular, if you are using normal maps, make sure to include Tangents when creating your GLTF files.

Incorrect usage of Bevy GLTF assets

Refer to the [GLTF page](#) to learn how to correctly use GLTF with Bevy.

GLTF files are complex. They contain many sub-assets, represented by different Bevy types. Make sure you are using the correct thing.

Make sure you are spawning a GLTF Scene, or using the correct `Mesh` and `StandardMaterial` associated with the correct GLTF Primitive.

If you are using an asset path, be sure to include a label for the sub-asset you want:

```
asset_server.load("my.gltf#Scene0");
```

If you are spawning the top-level `Gltf` master asset, it won't work.

If you are spawning a GLTF Mesh, it won't work.

Unsupported GLTF

Bevy does not fully support all features of the GLTF format and has some specific requirements about the data. Not all GLTF files can be loaded and rendered in Bevy. Unfortunately, in many of these cases, you will not get any error or diagnostic message.

Commonly-encountered limitations:

- Textures embedded in ascii (`image1.tif`) files (base64 encoding) cannot be loaded. Put your

- Textures embedded in ascii (`*.gltf`) files (base64 encoding) cannot be loaded. Put your textures in external files, or use the binary (`*.glb`) format.
- Mipmaps are only supported if the texture files (in KTX2 or DDS format) contain them. The GLTF spec requires missing mipmap data to be generated by the game engine, but Bevy does not support this. If your assets are missing mipmaps, textures will look grainy/noisy.
- Bevy's renderer requires all meshes/primitives to have per-vertex positions, UVs, and normals. Make sure all of this data is included.
- Meshes/primitives without textures (if the material is just a solid color) must still include UVs regardless. Bevy will not render meshes without UVs.
- When using normal maps in your material, tangents must also be included in the mesh. Assets with normal maps but without tangents are valid; other software would typically autogenerate the tangents if they are missing, but Bevy does not support this yet. Be sure to tick the checkbox for including tangents when exporting.
- Spot lights are not supported. Bevy currently only has Point lights and Directional lights.

This list is not exhaustive. There may be other unsupported scenarios that I did not know of or forgot to include here. :)

Unoptimized / Debug builds

Maybe your asset just takes a while to load? Bevy is very slow without compiler optimizations. It's actually possible that complex GLTF files with big textures can take over a minute to load and show up on the screen. It would be almost instant in optimized builds. [See here](#).

Vertex Order and Culling

By default, the Bevy renderer assumes Counter-Clockwise vertex order and has back-face culling enabled.

If you are generating your `Mesh` from code, make sure your vertices are in the correct order.

Borrow multiple fields from struct

When you have a [component](#) or [resource](#), that is larger struct with multiple fields, sometimes you want to borrow several of the fields at the same time, possibly mutably.

```
struct MyThing {
    a: Foo,
    b: Bar,
}

fn my_system(mut q: Query<&mut MyThing>) {
    for thing in q.iter_mut() {
        helper_func(&thing.a, &mut thing.b); // ERROR!
    }
}

fn helper_func(foo: &Foo, bar: &mut Bar) {
    // do something
}
```

This can result in a compiler error about conflicting borrows:

```
error[E0502]: cannot borrow `thing` as mutable because it is also borrowed as immutable
|
|         helper_func(&thing.a, &mut thing.b); // ERROR!
|         -----             ^^^^^^ mutable borrow occurs here
|         |                 |
|         |                 immutable borrow occurs here
|         |                 immutable borrow later used by call
|         |
```

The solution is to use the "reborrow" idiom, a common but non-obvious trick in Rust programming:

```
// add this at the start of the for loop, before using `thing`:
let thing = &mut *thing;
```

Note that this line triggers [change detection](#). Even if you don't modify the data afterwards, the component gets marked as changed.

Explanation

Bevy typically gives you access to your data via special wrapper types (like [Res<T>](#), [ResMut<T>](#), and [Mut<T>](#) (when [querying](#) for components mutably)). This lets Bevy track access to the data.

These are "smart pointer" types that use the Rust [Deref](#) trait to dereference to your data. They usually work seamlessly and you don't even notice them.

However, in a sense, they are opaque to the compiler. The Rust language allows fields of a struct to be borrowed individually, when you have direct access to the struct, but this does not work when it is wrapped in another type.

The "reborrow" trick shown above, effectively converts the wrapper into a regular Rust reference. `*thing` dereferences the wrapper via [DerefMut](#), and then `&mut` borrows it mutably. You now have

`&mut MyStuff` instead of `Mut<MyStuff>`.

Bevy Time vs. Rust/OS time

Do *not* use `std::time::Instant::now()` to get the current time. Get your timing information from Bevy, using `Res<Time>`.

Rust (and the OS) give you the precise time of the moment you call that function. However, that's not what you want.

Your game systems are run by Bevy's parallel scheduler, which means that they could be called at vastly different instants every frame! This will result in inconsistent / jittery timings and make your game misbehave or look stuttery.

Bevy's `Time` gives you timing information that is consistent throughout the frame update cycle. It is intended to be used for game logic.

This is not Bevy-specific, but applies to game development in general. Always get your time from your game engine, not from your programming language or operating system.

UI layout is inverted

In bevy, the Y axis always points *UP*. When working with UI, the origin is at the *bottom left* corner of the screen.

This means that UI is laid out from bottom to top.

This is the opposite of the typical behavior of web pages and other UI toolkits, where layout works from top to bottom.

Bevy uses the Flexbox layout model for UI, but unlike in web pages / CSS, the vertical axis is inverted.

Unintuitively, this means that to build UIs that flow from top to bottom, you need to use

```
FlexDirection::ColumnReverse
```

.

UV coordinates in Bevy

In Bevy, the vertical axis for the pixels of textures / images, and when sampling textures in a shader, points *downwards*, from top to bottom. The origin is at the top left.

This is consistent with how most image file formats store pixel data, and with how most graphics APIs work (including DirectX, Vulkan, Metal, WebGPU, but *not* OpenGL).

This is different from OpenGL (and frameworks based on it). If your prior experience is with these, you may find that the textures on your meshes are flipped vertically. You will have to reexport / regenerate your meshes in the correct UV format.

This is also inconsistent with the [World-coordinate system used everywhere else in Bevy](#), where the Y axis points up.

If the images of your 2D sprites are flipped (for whatever reason), you can correct that using Bevy's sprite-flipping feature:

```
commands.spawn_bundle(SpriteBundle {  
    sprite: Sprite {  
        flip_y: true,  
        flip_x: false,  
        ..Default::default()  
    },  
    ..Default::default()  
});
```

This chapter covers the foundational information about the game engine aspects of Bevy. It serves as an addition to the [Bevy Programming Framework](#) chapter.

Coordinate System

Bevy uses a right-handed Y-up coordinate system.

Bevy uses the same coordinate system for 3D, 2D, and UI, for consistency.

It is easiest to explain in terms of 2D:

- The X axis goes from left to right (+X points right).
- The Y axis goes from bottom to top (+Y points up).
- The Z axis goes from far to near (+Z points towards you, out of the screen).
- For 2D, the origin (X=0.0; Y=0.0) is at the *center of the screen* by default.
 - For UI, the origin is at the *bottom left* corner.

When you are working with 2D sprites, you can put the background on Z=0.0, and place other sprites at increasing positive Z coordinates to layer them on top.

In 3D, the axes are oriented the same way.

This is a right-handed coordinate system. You can use the fingers of your right hand to visualize the 3 axes: thumb=X, index=Y, middle=Z.

It is the same as Godot, Maya, and OpenGL. Compared to Unity, the Z axis is inverted.

Note: In Bevy, the Y axis always points *UP*.

This may feel **unintuitive when working with UI** (as it is the opposite from web pages), or if you are used to working with 2D libraries where the Y axis points down.

Also beware of a common pitfall when working in 2D: **the camera must be positioned at a far away Z coordinate (=999.9 by default), or you might not be able to see your sprites!**

Transforms

Relevant official examples: `transform`, `translation`, `rotation`, `3d_rotation`, `scale`, `move_sprite`, `parenting`, anything that spawns 2D or 3D objects.

First, a quick definition, if you are new to game development:

a Transform is what allows you to place an object in the game world. It is a combination of the object's "translation" (position/coordinates), "rotation", and "scale" (size adjustment).

You move objects around by modifying the translation, rotate them by modifying the rotation, and make them larger or smaller by modifying the scale.

Transform Components

In Bevy, transforms are represented by *two* components: `Transform` and `GlobalTransform`. Any `Entity` that represents an object in the game world needs to have both. All of Bevy's `bundle types` include them. If you are creating a custom entity, you can use `TransformBundle` to ensure you don't miss them.

`Transform` is what you typically work with. It is a `struct` containing the translation, rotation, and scale. To read or manipulate these values, access them from your `systems` using a `query`.

If the entity has a `parent`, the `Transform` component is relative to the parent. This means that the child object will move/rotate/scale along with the parent.

`GlobalTransform` represents the absolute global position in the world. If the entity does not have a parent, then this will have the same value as the `Transform`. The value of `GlobalTransform` is calculated/managed internally by Bevy. You should treat it as read-only; do not mutate it.

Beware: The two components are synchronized by a bevy-internal system (the "transform propagation system"), which runs in the `PostUpdate` stage. This is somewhat finicky and can result in tricky pitfalls if you are trying to do advanced things that rely on both the relative/local and the absolute/global transforms of entities. When you mutate the `Transform`, the `GlobalTransform` is not updated immediately. They will be out-of-sync until the transform propagation system runs.

Time and Timers

Relevant official examples: `timers`, `move_sprite`.

Time

The `Time` resource is your main global source of timing information, that you can access from any system that does anything that needs time. You should derive all timings from it.

Bevy updates these values at the beginning of every frame.

Delta Time

The most common use case is "delta time" – how much time passed between the previous frame update and the current one. This tells you how fast the game is running, so you can scale things like movement and animations, so they can happen smoothly, regardless of the game's frame rate.

```
fn asteroids_fly(
    time: Res<Time>,
    mut q: Query<&mut Transform, With<Asteroid>>,
) {
    for mut transform in q.iter_mut() {
        // move our asteroids along the X axis
        // at a speed of 10.0 units per second
        transform.translation.x += 10.0 * time.delta_seconds();
    }
}
```

Ongoing Time

`Time` can also give you the total running time since startup. Use this if you need a cumulative, increasing, measurement of time.

```
use std::time::Instant;

/// Say, for whatever reason, we want to keep track
/// of when exactly some specific entities were spawned.
#[derive(Component)]
struct SpawnedTime(Instant);

fn spawn_my_stuff(
    mut commands: Commands,
    time: Res<Time>,
) {
    commands.spawn()
        .insert(SpawnedTime(time.startup() + time.time_since_startup()));
}
```

Timers and Stopwatches

There are also facilities to help you track specific intervals or timings: `Timer` and `Stopwatch`. You can create many instances of these, to track whatever you want. You can use them in your own `component` or `resource` types.

Timers and Stopwatches need to be ticked. You need to have some system calling `.tick(delta)`, for it to make progress, or it will be inactive. The delta should come from the `Time` resource.

Timer

`Timer` allows you to detect when a certain interval of time has elapsed. Timers have a set duration. They can be "repeating" or "non-repeating".

Both kinds can be manually "reset" (start counting the time interval from the beginning) and "paused" (they will not progress even if you keep ticking them).

Repeating timers will automatically reset themselves after they reach their set duration.

Use `.finished()` to detect when a timer has reached its set duration. For non-repeating timers, you can also use `.just_finished()`, if you need to respond only on the exact update when the duration was reached.

```
use std::time::Duration;
```

```
#ElapTime(Component)
```

```

#[derive(Component)]
struct FuseTime {
    /// track when the bomb should explode (non-repeating timer)
    timer: Timer,
}

fn explode_bombs(
    mut commands: Commands,
    mut q: Query<(Entity, &mut FuseTime)>,
    time: Res<Time>,
) {
    for (entity, mut fuse_timer) in q.iter_mut() {
        // timers gotta be ticked, to work
        fuse_timer.timer.tick(time.delta());

        // if it finished, despawn the bomb
        if fuse_timer.timer.finished() {
            commands.entity(entity).despawn();
        }
    }
}

struct BombsSpawnConfig {
    /// How often to spawn a new bomb? (repeating timer)
    timer: Timer,
}

/// Spawn a new bomb in set intervals of time
fn spawn_bombs(
    mut commands: Commands,
    time: Res<Time>,
    mut config: ResMut<BombsSpawnConfig>,
) {
    // tick the timer
    config.timer.tick(time.delta());

    if config.timer.finished() {
        commands.spawn()
            .insert(FuseTime {
                // create the non-repeating fuse timer
                timer: Timer::new(Duration::from_secs(5), false),
            });
    }
}

/// Configure our bomb spawning algorithm
fn setup_bomb_spawning(
    mut commands: Commands,
) {
    commands.insert_resource(BombsSpawnConfig {
        // create the repeating timer
        timer: Timer::new(Duration::from_secs(10), true),
    })
}

```

Note that Bevy's timers do *not* work like typical real-life timers (which count downwards toward zero). Bevy's timers start from zero and count *up* towards their set duration. They are basically like stopwatches with extra features: a maximum duration and optional auto-reset.

Stopwatch

`Stopwatch` allow you to track how much time has passed since a certain point.

It will just keep accumulating time, which you can check with `.elapsed()` / `.elapsed_secs()`. You can manually reset it at any time.

```
use bevy::core::Stopwatch;

#[derive(Component)]
struct JumpDuration {
    time: Stopwatch,
}

fn jump_duration(
    time: Res<Time>,
    mut q_player: Query<&mut JumpDuration, With<Player>>,
    kbd: Res<Input<KeyCode>>,
) {
    // assume we have exactly one player that jumps with Spacebar
    let mut jump = q_player.single_mut();

    if kbd.just_pressed(KeyCode::Space) {
        jump.time.reset();
    }

    if kbd.pressed(KeyCode::Space) {
        println!("Jumping for {} seconds.", jump.time.elapsed_secs());
        // stopwatch has to be ticked to progress
        jump.time.tick(time.delta());
    }
}
```

Hierarchical (Parent/Child) Entities

Relevant official examples: [hierarchy](#), [parenting](#).

Technically, the [Entities/Components](#) themselves cannot form a hierarchy (the [ECS](#) is a flat data structure). However, logical hierarchies are a common pattern in games.

Bevy supports creating such a logical link between entities, to form a virtual "hierarchy", by simply adding [Parent](#) and [Children](#) components on the respective entities.

When using [Commands](#) to spawn entities, [Commands](#) has methods for adding children to entities, which automatically add the correct components:

```
// spawn the parent and get its Entity id
let parent = commands.spawn_bundle(MyParentBundle::default())
    .id();

// do the same for the child
let child = commands.spawn_bundle(MyChildBundle::default())
    .id();

// add the child to the parent
commands.entity(parent).push_children(&[child]);

// you can also use `with_children`:
commands.spawn_bundle(MyParentBundle::default())
    .with_children(|parent| {
        parent.spawn_bundle(MyChildBundle::default());
    });
```

You can despawn an entire hierarchy with a single [command](#):

```
fn close_menu(
    mut commands: Commands,
    query: Query<Entity, With<MainMenuUI>>,
) {
    for entity in query.iter() {
        // despawn the entity and its children
        commands.entity(entity).despawn_recursive();
    }
}
```

Accessing the Parent or Children

To make a system that works with the hierarchy, you typically need two [queries](#):

- one with the components you need from the child entities
- one with the components you need from the parent entities

One of the two queries should include the appropriate component, to obtain the entity ids to use with the other one:

- [Parent](#) in the child query, if you want to iterate entities and look up their parents, or
- [Children](#) in the parent query, if you want to iterate entities and look up their children

For example, if we want to get the `Transform` of cameras (`Camera`) that have a parent, and the `GlobalTransform` of their parent:

```
fn camera_with_parent(
    q_child: Query<(&Parent, &Transform), With<Camera>>,
    q_parent: Query<&GlobalTransform>,
) {
    for (parent, child_transform) in q_child.iter() {
        // `parent` contains the Entity ID we can use
        // to query components from the parent:
        let parent_global_transform = q_parent.get(parent.0);

        // do something with the components
    }
}
```

As another example, say we are making a strategy game, and we have Units that are children of a Squad. Say we need to make a system that works on each Squad, and it needs some information about the children:

```
fn process_squad_damage(
    q_parent: Query<(&MySquadDamage, &Children)>,
    q_child: Query<&MyUnitHealth>,
) {
    // get the properties of each squad
    for (squad_dmg, children) in q_parent.iter() {
        // `children` is a collection of Entity IDs
        for &child in children.iter() {
            // get the health of each child unit
            let health = q_child.get(child);

            // do something
        }
    }
}
```

Relative Transforms

If your entities represent "objects in the game world", you probably expect the child to be positioned relative to the parent and move with it.

All [Bundles that come with Bevy](#) provide this behavior automatically. You should at least use the basic `TransformBundle` if you don't need anything else.

For more info, see the [dedicated page about transforms](#).

Fixed Timestep

Relevant official examples: `fixed_timestep`.

Consider using the `iyas_loopless` crate, which provides an alternative implementation that does not suffer from the `usability issues` of the one in Bevy.

If you need something to happen at fixed time intervals (a common use case is Physics updates), you can add the respective `systems` to your `app` using Bevy's `FixedTimestep` `Run Criteria`.

```
use bevy::core::FixedTimestep;

// The timestep says how many times to run the SystemSet every second
// For TIMESTEP_1, it's once every second
// For TIMESTEP_2, it's twice every second

const TIMESTEP_1_PER_SECOND: f64 = 60.0 / 60.0;
const TIMESTEP_2_PER_SECOND: f64 = 30.0 / 60.0;

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_system_set(
            SystemSet::new()
                // This prints out "hello world" once every second
                .with_run_criteria(FixedTimestep::step(TIMESTEP_1_PER_SECOND))
                .with_system(slow_timestep)
        )
        .add_system_set(
            SystemSet::new()
                // This prints out "goodbye world" twice every second
                .with_run_criteria(FixedTimestep::step(TIMESTEP_2_PER_SECOND))
                .with_system(fast_timestep)
        )
        .run();
}

fn slow_timestep() {
    println!("hello world");
}

fn fast_timestep() {
    println!("goodbye world");
}
```

State

You can check the current state of the fixed timestep trackers, by accessing the `FixedTimesteps` `resource`. This lets you know how much time remains until the next time it triggers, or how much it has overstepped. You need to label your fixed timesteps.

See the [official example](#), which illustrates this.

Caveats

The major problem with Bevy's fixed timestep comes from the fact that it is implemented using [Run Criteria](#). It cannot be combined with other run criteria, such as [states](#). This makes it unusable for most projects, which need to rely on states for things like implementing the main menu / loading screen / etc. Consider using [iyes_loopless](#), which does not have this problem.

Also, note that your [systems](#) are still called as part of the regular frame-update cycle, along with all of the normal systems. So, the timing is not exact.

The [FixedTimestep](#) run criteria simply checks how much time passed since the last time your systems were ran, and decides whether to run them during the current frame, or not, or run them multiple times, as needed.

Danger! Lost events!

By default, Bevy's [events](#) are *not reliable*! They only persist for 2 frames, after which they are lost. If your fixed-timestep systems receive events, beware that you may miss some events if the framerate is higher than 2x the fixed timestep.

One way around that is to use [events with manual clearing](#). This gives you control over how long events persist, but can also leak / waste memory if you forget to clear them.

Bevy's own built-in audio support is extremely barebones and limited. It can play sounds and give some control over its volume and playback.

TODO: show how to use Bevy audio, now that its usability has improved.

Kira Audio

Instead, you could try the `bevy_kira_audio` community plugin, which integrates the Kira sound library with bevy. Kira is much more feature-rich, including support for managing many audio tracks (like background music and sound effects), with volume control, stereo panning, playback rate, and streaming.

Using `bevy_kira_audio` in your project requires some extra configuration, because you need to disable Bevy's own audio. Bevy's audio is a cargo feature that is enabled by default, but must be disabled. Cargo does not let you disable individual default features, so you need to disable all default bevy features and re-enable the ones you need. [See here](#) for more info.

You must not include the `bevy_audio` feature, or any of the audio file formats (such as the default `vorbis`). Enable the file formats you care about on `bevy_kira_audio` instead of Bevy.

```
[dependencies.bevy]
version = "0.7"
default-features = false
# These are the remaining default features other than `bevy_audio` and `mp3`
features = [
    "render",
    "animation",
    "bevy_winit",
    "bevy_gilrs",
    "png",
    "hdr",
    "filesystem_watcher",
    "x11"
]

[dependencies.bevy_kira_audio]
version = "0.9.0"
# `ogg` format support is enabled by default, disable if you don't want it
default-features = false
# enable the features you care about
features = [
    "wav",
    "flac",
    "mp3",
    "ogg",
]
```

Bevy Asset Management

Assets are the data that the game engine is working with: all of your images, 3D models, sounds, scenes, game-specific things like item descriptions, and more!

Bevy has a flexible system for loading and managing your game assets asynchronously (in the background, without causing lag spikes in your game).

In your code, you refer to individual assets using [handles](#).

Asset data can be [loaded from files](#) and also [accessed from code](#). [Hot-reloading](#) is supported to help you during development, by reloading asset files if they change while the game is running.

If you want to write some code to do something when assets finish loading, get modified, or are unloaded, you can use [asset events](#).

Handles

Handles are lightweight IDs that refer to a specific asset. You need them to use your assets, for example to [spawn entities](#) like [2D sprites](#) or [3D models](#), or to [access the data of the assets](#).

Handles have the Rust type `Handle<T>`, where `T` is the [asset type](#).

You can store handles in your [entity components](#) or [resources](#).

Handles can refer to not-yet-loaded assets, meaning you can just spawn your entities anyway, using the handles, and the assets will just "pop in" when they become ready.

Obtaining Handles

If you are [loading an asset from a file](#), the `asset_server.load(...)` call will give you the handle. The loading of the data happens in the background, meaning that the handle will initially refer to an unloaded asset, and the actual data will become available later.

If you are [creating your own asset data from code](#), the `assets.add(...)` call will give you the handle.

Reference Counting; Strong and Weak Handles

Bevy keeps track of how many handles to a given asset exist at any time. Bevy will automatically unload unused assets, after the last handle is dropped.

For this reason, creating additional handles to the same asset requires you to call `handle.clone()`. This makes the operation explicit, to ensure you are aware of all the places in your code where you create additional handles. The `.clone()` operation is cheap, so don't worry about performance (in most cases).

There are two kinds of handles: "strong" and "weak". Strong assets are counted, weak handles are not. By default, handles are strong. If you want to create a weak handle, use `.clone_weak()` (instead of `.clone()`) on an existing handle. Bevy can unload the asset after all strong handles are gone, even if you are still holding some weak handles.

Untyped Handles

Bevy also has a `HandleUntyped` type. Use this type of handle if you need to be able to refer to any asset, regardless of the asset type.

This allows you to store a collection (such as `Vec` or `HashMap`) containing assets of mixed types.

Just like regular handles, untyped handles can be strong or weak, and can be used to [access asset data](#).

You can create an untyped handle using `.clone_untyped()` on an existing handle.

You can convert an untyped handle into a typed handle with `typed_from_untyped(<T>())`, specifying the type to

You can convert an untyped handle into a typed handle with `.typed::<T>()`, specifying the type to use.

Load Assets from Files with AssetServer

Relevant official examples: `asset_loading`.

To load assets from files, use the `AssetServer` resource.

```
struct UiFont(Handle<Font>);

fn load_ui_font(
    mut commands: Commands,
    server: Res<AssetServer>
) {
    let handle: Handle<Font> = server.load("font.ttf");

    // we can store the handle in a resource:
    // - to prevent the asset from being unloaded
    // - if we want to use it to access the asset later
    commands.insert_resource(UiFont(handle));
}
```

This queues the asset loading to happen in the background, and return a `handle`. The asset will take some time to become available. You cannot access the actual data immediately in the same `system`, but you can use the handle.

You can spawn entities like your 2D sprites, 3D models, and UI, using the handle, even before the asset has loaded. They will just "pop in" later, when the asset becomes ready.

Note that it is OK to call `asset_server.load(...)` as many times as you want, even if the asset is currently loading, or already loaded. It will just provide you with the same handle. Every time you call it, it will just check the status of the asset, begin loading it if needed, and give you a handle.

Bevy supports loading [a variety of asset file formats](#), and can be extended to support more. The asset loader implementation to use is selected based on the file extension.

Untyped Loading

If you want an `untyped handle`, you can use `asset_server.load_untyped(...)` instead.

You can also load an entire folder of assets, regardless of how many files are inside, using `asset_server.load_folder(...)`. This gives you a `Vec<HandleUntyped>` with all the untyped handles.

```
struct ExtraAssets(Vec<HandleUntyped>);

fn load_extra_assets(
    mut commands: Commands,
    server: Res<AssetServer>,
) {
    if let Ok(handles) = server.load_folder("extra") {
        commands.insert_resource(ExtraAssets(handles));
    }
}
```

Untyped loading is possible, because Bevy always detects the file type from the file extension anyway.

AssetPath and Labels

The asset path you use to identify an asset from the filesystem is actually a special `AssetPath`, which consists of the file path + a label. Labels are used in situations where multiple assets are contained in the same file. An example of this are [GLTF files](#), which can contain meshes, scenes, textures, materials, etc.

Asset paths can be created from a string, with the label (if any) attached after a `#` symbol.

```
fn load_gltf_things(
    mut commands: Commands,
    server: Res<AssetServer>
) {
    // get a specific mesh
    let my_mesh: Handle<Mesh> = server.load("my_scene.gltf#Mesh0/Primitive0");

    // spawn a whole scene
    let my_scene: Handle<Scene> = server.load("my_scene.gltf#Scene0");
    commands.spawn_scene(my_scene);
}
```

See the [GLTF page](#) for more info about working with 3D models.

Where are assets loaded from?

The asset server internally relies on an implementation of the `AssetIo` Rust trait, which is Bevy's way of providing "backends" for fetching data from different types of storage.

Bevy provides its own default built-in I/O backend for desktop platforms and for WebAssembly.

On desktop platforms, it treats asset paths as relative to a folder called `assets`, that must be placed at one of the following locations:

- Alongside the game's executable file, for distribution
- In your Cargo project folder, when running your game using `cargo` during development
 - This is identified by the `CARGO_MANIFEST_DIR` environment variable

On the web, it fetches assets using HTTP URLs pointing within an `assets` folder located alongside the game's `.wasm` file.

There are [unofficial plugins](#) available that provide additional I/O backend implementations, such as for loading assets from inside archive files.

Access the Asset Data

To access the actual asset data from systems, use the `Assets<T>` resource.

You can identify your desired asset using either the `handle` (`untyped handles` can also be used) or the `asset path`:

```
struct SpriteSheets {
    map_tiles: Handle<TextureAtlas>,
}

fn use_sprites(
    handles: Res<SpriteSheets>,
    atlases: Res<Assets<TextureAtlas>>,
    images: Res<Assets<Image>>,
) {
    // Could be `None` if the asset isn't loaded yet
    if let Some(atlas) = atlases.get(&handles.map_tiles) {
        // do something with the texture atlas
    }

    // Can use a path instead of a handle
    if let Some(map_tex) = images.get("map.png") {
        // if "map.png" was loaded, we can use it!
    }
}
```

Creating Assets from Code

You can also add assets to `Assets<T>` manually.

Sometimes you need to create assets from code, rather than `loading them from files`. Some common examples of such use-cases are:

- creating texture atlases
- creating 3D or 2D materials
- procedurally-generating assets like images or 3D meshes

To do this, first create the data for the asset (an instance of the `asset type`), and then add it `.add(...)` to the `Assets<T>` resource, for it to be stored and tracked by Bevy. You will get a `handle` to use to refer to it, just like any other asset.

```
fn add_material(
    mut materials: ResMut<Assets<StandardMaterial>>,
) {
```

```
) {  
  let new_mat = StandardMaterial {  
    base_color: Color::rgba(0.25, 0.50, 0.75, 1.0),  
    unlit: true,  
    ..Default::default()  
  };  
  
  let handle = materials.add(new_mat);  
  
  // do something with the handle  
}
```

React to Changes with Asset Events

If you need to perform specific actions when an asset is created, modified, or removed, you can make a [system](#) that reacts to `AssetEvent` events.

```
struct MyMapImage {
    handle: Handle<Image>,
}

fn fixup_images(
    mut ev_asset: EventReader<AssetEvent<Image>>,
    mut assets: ResMut<Assets<Image>>,
    map_img: Res<MyMapImage>,
) {
    for ev in ev_asset.iter() {
        match ev {
            AssetEvent::Created { handle } |
            AssetEvent::Modified { handle } => {
                // a texture was just loaded or changed!

                let texture = assets.get_mut(handle).unwrap();
                // ^ unwrap is OK, because we know it is loaded now

                if *handle == map_img.handle {
                    // it is our special map image!
                } else {
                    // it is some other image
                }
            }
            AssetEvent::Removed { handle } => {
                // an image was unloaded
            }
        }
    }
}
```

Track Loading Progress

There are good community plugins that can help with this. See [my recommendations for helper crates](#). Otherwise, this page shows you how to do it manually.

If you want to check the status of various [asset files](#), you can poll it from the `AssetServer`. It will tell you whether the asset(s) are loaded, still loading, not loaded, or encountered an error.

To check an individual asset, you can use `asset_server.get_load_state(...)` with a handle or path to refer to the asset.

To check a group of many assets, you can add them to a single collection (such as a `Vec<HandleUntyped>`; [untyped handles](#) are very useful for this) and use `asset_server.get_group_load_state(...)`.

Here is a more complete code example:

```
struct AssetsLoading(Vec<HandleUntyped>);
```

```
fn asset_server(asset_server: &AssetServer) -> AssetsLoading { let mut asset_loading = AssetsLoading::new();
```

```

fn setup(server: Res<AssetServer>, mut loading: ResMut<AssetsLoading>) {
    // we can have different asset types
    let font: Handle<Font> = server.load("my_font.ttf");
    let menu_bg: Handle<Image> = server.load("menu.png");
    let scene: Handle<Scene> = server.load("level01.glTF#Scene0");

    // add them all to our collection for tracking
    loading.0.push(font.clone_untyped());
    loading.0.push(menu_bg.clone_untyped());
    loading.0.push(scene.clone_untyped());
}

fn check_assets_ready(
    mut commands: Commands,
    server: Res<AssetServer>,
    loading: Res<AssetsLoading>
) {
    use bevy::asset::LoadState;

    match server.get_group_load_state(loading.0.iter().map(|h| h.id)) {
        LoadState::Failed => {
            // one of our assets had an error
        }
        LoadState::Loaded => {
            // all assets are now ready

            // this might be a good place to transition into your in-game state

            // remove the resource to drop the tracking handles
            commands.remove_resource::();
            // (note: if you don't have any other handles to the assets
            // elsewhere, they will get unloaded after this)
        }
        _ => {
            // NotLoaded/Loading: not fully ready yet
        }
    }
}

```

Hot-Reloading Assets

Relevant official examples: [hot_asset_reloading](#).

At runtime, if you modify the file of an [asset](#) that is loaded into the game (via the [AssetServer](#)), Bevy can detect that and reload the asset automatically. This is very useful for quick iteration. You can edit your assets while the game is running and see the changes instantly in-game.

Not all [file formats](#) and use cases are supported equally well. Typical asset types like textures / images should work without issues, but complex GLTF or scene files, or assets involving custom logic, might not.

If you need to run custom logic as part of your hot-reloading workflow, you could implement it in a [system](#), using [AssetEvent](#) ([learn more](#)).

Hot reloading is opt-in and has to be enabled in order to work. You can do this in a [startup system](#):

```
asset_server.watch_for_changes().unwrap();
```

Note that this requires the [filesystem_watcher](#) [Bevy cargo feature](#). It is enabled by default, but if you have disabled default features to customize Bevy, be sure to include it if you need it.

Shaders

Bevy also supports hot-reloading for shaders. You can edit your custom shader code and see the changes immediately.

This only works if you are loading your shaders through the bevy asset system (via the [AssetServer](#)).

Shader code that does not come from asset files, such as if you include it as a static string in your source code, cannot be hot-reloaded.

Input Handling

[Click here to download example code.](#)

This is a complete example that you can run. It will print all input activity to the console.

Bevy supports the following inputs:

- [Keyboard](#) (detect when keys are pressed or released)
- [Character](#) (for text input; keyboard layout handled by the OS)
- [Mouse](#) (relative motion, buttons, scrolling)
 - [Motion](#) (moving the mouse, not tied to OS cursor)
 - [Cursor](#) (absolute pointer position)
 - [Buttons](#)
 - [Scrolling](#) (mouse wheel or touchpad gesture)
- [Touchscreen](#) (with multi-touch)
- [Gamepad \(Controller, Joystick\)](#) (via the [gilrs](#) library)

Sensors, like accelerometers and gyroscopes, are not supported yet.

For most input types (where it makes sense), Bevy provides two ways of dealing with them:

- by checking the current state via [resources](#) ([input resources](#)),
- or via [events](#) ([input events](#)).

Some inputs are only provided as events.

Checking state is done using [resources](#) such as [Input](#) (for binary inputs like keys or buttons), [Axis](#) (for analog inputs), [Touches](#) (for fingers on a touchscreen), etc. This way of handling input is very convenient for implementing game logic. In these scenarios, you typically only care about the specific inputs mapped to actions in your game. You can check specific buttons/keys to see when they get pressed/released, or what their current state is.

[Events](#) ([input events](#)) are a lower-level, more all-encompassing approach. Use them if you want to get all activity from that class of input device, rather than only checking for specific inputs.

Input Mapping

Bevy does not yet offer a built-in way to do input mapping (configure key bindings, etc). You need to come up with your own way of translating the inputs into logical actions in your game/app.

There are some community-made plugins that may help with that: [see the input-section on bevy-assets](#). My personal recommendation: [Input Manager plugin by Leafwing Studios](#).

It may be a good idea to build your own abstractions specific to your game. For example, if you need to handle player movement, you might want to have a system for reading inputs and converting them to your own internal "movement intent/action events", and then another system acting on

those internal events, to actually move the player. Make sure to use [explicit system ordering](#) to avoid lag / frame delays.

Relevant official examples: `keyboard_input`, `keyboard_input_events`.

This page shows how to handle keyboard keys being pressed and released.

If you are interested in text input, see the [Character Input](#) page instead.

Note: Command Key on Mac corresponds to the Super/Windows Key on PC.

Checking Key State

Checking the state of specific keys can currently only be done by Key Code, using the `Input<KeyCode>` (`Input`, `KeyCode`) resource:

```
fn keyboard_input(
    keys: Res<Input<KeyCode>>,
) {
    if keys.just_pressed(KeyCode::Space) {
        // Space was pressed
    }
    if keys.just_released(KeyCode::LControl) {
        // Left Ctrl was released
    }
    if keys.pressed(KeyCode::W) {
        // W is being held down
    }
    // we can check multiple at once with `.any_*`
    if keys.any_pressed([KeyCode::LShift, KeyCode::RShift]) {
        // Either the left or right shift are being held down
    }
    if keys.any_just_pressed([KeyCode::Delete, KeyCode::Back]) {
        // Either delete or backspace was just pressed
    }
}
```

Keyboard Events

To get all keyboard activity, you can use `KeyboardInput` events:

```
fn keyboard_events(
    mut key_evr: EventReader<KeyboardInput>,
) {
```

```

) {
    use bevy::input::ElementState;

    for ev in key_evr.iter() {
        match ev.state {
            ElementState::Pressed => {
                println!("Key press: {:?} ({})", ev.key_code, ev.scan_code);
            }
            ElementState::Released => {
                println!("Key release: {:?} ({})", ev.key_code, ev.scan_code);
            }
        }
    }
}

```

These events give you both the Key Code and Scan Code. The Scan Code is represented as an arbitrary `u32` integer ID.

Key Codes and Scan Codes

Keyboard keys can be identified by Key Code or Scan Code.

Key Codes represent the symbol/letter on each key and are dependent on the keyboard layout currently active in the user's OS. Bevy represents them with the `KeyCode` enum.

Scan Codes represent the physical key on the keyboard, regardless of the system layout. Unfortunately, they are just arbitrary integer IDs and platform-dependent. There is no easy way to know what to display in the game's UI for the user, from the scan code.

Additionally, support for using Scan Codes in Bevy is limited. This can be annoying for people with multiple or non-QWERTY keyboard layouts.

See [Bevy Issue #2052](#) for efforts to improve this situation.

Layout-Agnostic Key Bindings

You could try to provide a better experience for players, regardless of these limitations, by:

- Internally recording and storing key bindings as Scan Codes
- Handling input using `events` and using Scan Codes to identify the key
- Storing Key Codes only for displaying the name of a key in the UI

Doing things this way means that users with multiple keyboard layouts in the OS will not have their keybindings break if they accidentally switch their layout mid-game, or start the game with the wrong layout.

Unfortunately, this also means your game UI will display the symbol from the layout that was used when registering the key bindings. This may be wrong or confusing if the user has changed the currently active layout.

Relevant official examples: `mouse_input`, `mouse_input_events`.

Mouse Buttons

Similar to [keyboard input](#), mouse buttons are available as an `Input` state [resource](#), as well as [events](#).

You can check the state of specific mouse buttons using `Input<MouseButton>`:

```
fn mouse_button_input(
    buttons: Res<Input<MouseButton>>,
) {
    if buttons.just_pressed(MouseButton::Left) {
        // Left button was pressed
    }
    if buttons.just_released(MouseButton::Left) {
        // Left Button was released
    }
    if buttons.pressed(MouseButton::Right) {
        // Right Button is being held down
    }
    // we can check multiple at once with `.any_*`
    if buttons.any_just_pressed([MouseButton::Left, MouseButton::Right]) {
        // Either the left or the right button was just pressed
    }
}
```

To get all press/release activity, use `MouseButtonInput` events:

```
fn mouse_button_events(
    mut mousebtn_evr: EventReader<MouseButtonInput>,
) {
    use bevy::input::ElementState;

    for ev in mousebtn_evr.iter() {
        match ev.state {
            ElementState::Pressed => {
                println!("Mouse button press: {:?}", ev.button);
            }
            ElementState::Released => {
                println!("Mouse button release: {:?}", ev.button);
            }
        }
    }
}
```

Mouse Scrolling / Wheel

To detect scrolling input, use `MouseWheel` events:

```
fn scroll_events(
    mut scroll_evr: EventReader<MouseWheel>,
) {
    use bevy::input::mouse::MouseScrollUnit;
    for ev in scroll_evr.iter() {
        match ev.unit {
            MouseScrollUnit::Line => {
                println!("Scroll (line units): vertical: {}, horizontal: {}", ev.y,
ev.x);
            }
            MouseScrollUnit::Pixel => {
                println!("Scroll (pixel units): vertical: {}, horizontal: {}", ev.y,
ev.x);
            }
        }
    }
}
```

The `MouseScrollUnit` enum is important: it tells you the type of scroll input. `Line` is for hardware with fixed steps, like the wheel on desktop mice. `Pixel` is for hardware with smooth (fine-grained) scrolling, like laptop touchpads.

You should probably handle each of these differently (with different sensitivity settings), to provide a good experience on both types of hardware.

Mouse Motion

Use this if you don't care about the exact position of the mouse cursor, but rather you just want to see how much it moved from frame to frame. This is useful for things like controlling a 3D camera.

Use `MouseMotion` events. Whenever the mouse is moved, you will get an event with the delta.

```
fn mouse_motion(
    mut motion_evr: EventReader<MouseMotion>,
) {
    for ev in motion_evr.iter() {
        println!("Mouse moved: X: {} px, Y: {} px", ev.delta.x, ev.delta.y);
    }
}
```

You might want to [grab/lock the mouse inside the game window](#).

Mouse Cursor Position

Use this if you want to accurately track the position pointer / cursor. This is useful for things like clicking and hovering over objects in your game or UI.

You can get the current coordinates of the mouse pointer, from the respective `Window` (if the mouse is currently inside that window):


```
fn cursor_position(
    windows: Res<Windows>,
) {
    // Games typically only have one window (the primary window).
    // For multi-window applications, you need to use a specific window ID here.
    let window = windows.get_primary().unwrap();

    if let Some(_position) = window.cursor_position() {
        // cursor is inside the window, position given
    } else {
        // cursor is not inside the window
    }
}
```

To detect when the pointer is moved, use `CursorMoved` events to get the updated coordinates:

```
fn cursor_events(
    mut cursor_evr: EventReader<CursorMoved>,
) {
    for ev in cursor_evr.iter() {
        println!(
            "New cursor position: X: {}, Y: {}, in Window ID: {:?}" ,
            ev.position.x, ev.position.y, ev.id
        );
    }
}
```

Note that you can only get the position of the mouse inside a window; you cannot get the global position of the mouse in the whole OS Desktop / on the screen as a whole.

To track when the mouse cursor enters and leaves your window(s), use `CursorEntered` and `CursorLeft` events.

Text / Character Input

Relevant official examples: `char_input_events`.

Use this (*not* `keyboard input`) if you want to implement text input in a Bevy app. This way, everything works as the user expects from their operating system, including Unicode support.

Bevy will produce a `ReceivedCharacter` event for every Unicode code point coming from the OS.

This example shows how to let the user input text into a string (here stored as a `local resource`).

```
/// prints every char coming in; press enter to echo the full string
fn text_input(
    mut char_evr: EventReader<ReceivedCharacter>,
    keys: Res<Input<KeyCode>>,
    mut string: Local<String>,
) {
    for ev in char_evr.iter() {
        println!("Got char: '{}'", ev.char);
        string.push(ev.char);
    }

    if keys.just_pressed(KeyCode::Return) {
        println!("Text input: {}", *string);
        string.clear();
    }
}
```

Gamepad (Controller, Joystick)

Relevant official examples: `gamepad_input`, `gamepad_input_events`.

Bevy has support for gamepad input hardware: console controllers, joysticks, etc. Many different kinds of hardware should work, but if your device is not supported, you should file an issue with the [gilrs](#) project.

Gamepad IDs

Bevy assigns a unique ID (`Gamepad`) to each connected gamepad. This lets you associate the device with a specific player and distinguish which one your inputs are coming from.

You can use the `Gamepads` [resource](#) to list the IDs of all the currently connected gamepad devices, or to check the status of a specific one.

To detect when gamepads are connected or disconnected, you can use `GamepadEvent` [events](#).

Example showing how to remember the first connected gamepad ID:

```
/// Simple resource to store the ID of the connected gamepad.
/// We need to know which gamepad to use for player input.
struct MyGamepad(Gamepad);
```

```

struct MyGamepad(Gamepad);

fn gamepad_connections(
    mut commands: Commands,
    my_gamepad: Option<Res<MyGamepad>>,
    mut gamepad_evr: EventReader<GamepadEvent>,
) {
    for GamepadEvent(id, kind) in gamepad_evr.iter() {
        match kind {
            GamepadEventType::Connected => {
                println!("New gamepad connected with ID: {:?}", id);

                // if we don't have any gamepad yet, use this one
                if my_gamepad.is_none() {
                    commands.insert_resource(MyGamepad(*id));
                }
            }
            GamepadEventType::Disconnected => {
                println!("Lost gamepad connection with ID: {:?}", id);

                // if it's the one we previously associated with the player,
                // disassociate it:
                if let Some(MyGamepad(old_id)) = my_gamepad.as_deref() {
                    if old_id == id {
                        commands.remove_resource::();
                    }
                }
            }
            _ => {}
        }
    }
}

```

Handling Gamepad Inputs

You can handle the analog sticks and triggers with `Axis<GamepadAxis>` (`Axis`, `GamepadAxis`). Buttons can be handled with `Input<GamepadButton>` (`Input`, `GamepadButton`), similar to [mouse buttons](#) or [keyboard keys](#).

Notice that the names of buttons in the `GamepadButton` are vendor-neutral (like `South` and `East` instead of `X/O` or `A/B`).

```

fn gamepad_input(
    axes: Res<Axis<GamepadAxis>>,
    buttons: Res<Input<GamepadButton>>,
) {
    for (axis, value) in axes.iter() {
        match axis {
            GamepadAxis::LeftStickX => {
                // ...
            }
            GamepadAxis::LeftStickY => {
                // ...
            }
            GamepadAxis::RightStickX => {
                // ...
            }
            GamepadAxis::RightStickY => {
                // ...
            }
            GamepadAxis::LeftTrigger => {
                // ...
            }
            GamepadAxis::RightTrigger => {
                // ...
            }
        }
    }

    for (button, value) in buttons.iter() {
        match button {
            GamepadButton::South => {
                // ...
            }
            GamepadButton::East => {
                // ...
            }
            GamepadButton::West => {
                // ...
            }
            GamepadButton::North => {
                // ...
            }
            GamepadButton::LeftStickButton1 => {
                // ...
            }
            GamepadButton::LeftStickButton2 => {
                // ...
            }
            GamepadButton::RightStickButton1 => {
                // ...
            }
            GamepadButton::RightStickButton2 => {
                // ...
            }
            GamepadButton::LeftTriggerButton1 => {
                // ...
            }
            GamepadButton::LeftTriggerButton2 => {
                // ...
            }
            GamepadButton::RightTriggerButton1 => {
                // ...
            }
            GamepadButton::RightTriggerButton2 => {
                // ...
            }
        }
    }
}

```

```

buttons: Res<Input<GamepadButton>>,
my_gamepad: Option<Res<MyGamepad>>,
) {
    let gamepad = if let Some(gp) = my_gamepad {
        // a gamepad is connected, we have the id
        gp.0
    } else {
        // no gamepad is connected
        return;
    };

    // The joysticks are represented using a separate axis for X and Y

    let axis_lx = GamepadAxis(gamepad, GamepadAxisType::LeftStickX);
    let axis_ly = GamepadAxis(gamepad, GamepadAxisType::LeftStickY);

    if let (Some(x), Some(y)) = (axes.get(axis_lx), axes.get(axis_ly)) {
        // combine X and Y into one vector
        let left_stick_pos = Vec2::new(x, y);

        // Example: check if the stick is pushed up
        if left_stick_pos.length() > 0.9 && left_stick_pos.y > 0.5 {
            // do something
        }
    }

    // In a real game, the buttons would be configurable, but here we hardcode them
    let jump_button = GamepadButton(gamepad, GamepadButtonType::South);
    let heal_button = GamepadButton(gamepad, GamepadButtonType::East);

    if buttons.just_pressed(jump_button) {
        // button just pressed: make the player jump
    }

    if buttons.pressed(heal_button) {
        // button being held down: heal the player
    }
}

```

You can also handle gamepad inputs using `GamepadEvent` events:

```

fn gamepad_input_events(
    my_gamepad: Option<Res<MyGamepad>>,
    mut gamepad_event: EventReader<GamepadEvent>
) {

```

```

mut gamepad_evr: EventReader<GamepadEvent>,
) {
    let gamepad = if let Some(gp) = my_gamepad {
        // a gamepad is connected, we have the id
        gp.0
    } else {
        // no gamepad is connected
        return;
    };

    for GamepadEvent(id, kind) in gamepad_evr.iter() {
        if id.0 != gamepad.0 {
            // event not from our gamepad
            continue;
        }

        use GamepadEventType::{AxisChanged, ButtonChanged};

        match kind {
            AxisChanged(GamepadAxisType::RightStickX, x) => {
                // Right Stick moved (X)
            }
            AxisChanged(GamepadAxisType::RightStickY, y) => {
                // Right Stick moved (Y)
            }
            ButtonChanged(GamepadButtonType::DPadDown, val) => {
                // buttons are also reported as analog, so use a threshold
                if *val > 0.5 {
                    // button pressed
                }
            }
            _ => {} // don't care about other inputs
        }
    }
}

```

Gamepad Settings

You can use the [GamepadSettings resource](#) to configure dead-zones and other parameters of the various axes and buttons. You can set the global defaults, as well as individually per-axis/button.

Here is an example showing how to configure gamepads with custom settings (not necessarily *good* settings, please don't copy these blindly):

```

// this should be run once, when the game is starting
// (transition entering your in-game state might be a good place to put it)
fn configure_gamepads()

```

```

fn configure_gamepads
    my_gamepad: Option<Res<MyGamepad>>,
    mut settings: ResMut<GamepadSettings>,
) {
    let gamepad = if let Some(gp) = my_gamepad {
        // a gamepad is connected, we have the id
        gp.0
    } else {
        // no gamepad is connected
        return;
    };

    // add a larger default dead-zone to all axes (ignore small inputs, round to zero)
    settings.default_axis_settings.negative_low = -0.1;
    settings.default_axis_settings.positive_low = 0.1;

    // make the right stick "binary", squash higher values to 1.0 and lower values to
    0.0
    let right_stick_settings = AxisSettings {
        positive_high: 0.5, // values 0.5 to 1.0, become 1.0
        positive_low: 0.5, // values 0.0 to 0.5, become 0.0
        negative_low: -0.5, // values -0.5 to 0.0, become 0.0
        negative_high: -0.5, // values -1.0 to -0.5, become -1.0
        // the raw value should change by at least this much,
        // for Bevy to register an input event:
        threshold: 0.01,
    };

    // make the triggers work in big/coarse steps, to get fewer events
    // reduces noise and precision
    let trigger_settings = AxisSettings {
        threshold: 0.2,
        // also set some conservative deadzones
        positive_high: 0.8,
        positive_low: 0.2,
        negative_high: -0.8,
        negative_low: -0.2,
    };

    // set these settings for the gamepad we use for our player
    settings.axis_settings.insert(
        GamepadAxis(gamepad, GamepadAxisType::RightStickX),
        right_stick_settings.clone()
    );
    settings.axis_settings.insert(
        GamepadAxis(gamepad, GamepadAxisType::RightStickY),
        right_stick_settings.clone()
    );
    settings.axis_settings.insert(
        GamepadAxis(gamepad, GamepadAxisType::LeftZ),
        trigger_settings.clone()
    );
    settings.axis_settings.insert(
        GamepadAxis(gamepad, GamepadAxisType::RightZ),
        trigger_settings.clone()
    );

    // for buttons (or axes treated as buttons), make them less sensitive
    let button_settings = ButtonSettings {

        // require them to be pressed almost all the way, to count
        press: 0.9,
        // require them to be released almost all the way, to count

```

```
    release: 0.1,  
  };  
  
  settings.default_button_settings = button_settings;  
}
```


Relevant official examples: `touch_input`, `touch_input_events`.

Multi-touch touchscreens are supported. You can track multiple fingers on the screen, with position and pressure/force information. Bevy does not offer gesture recognition.

The `Touches` resource allows you to track any fingers currently on the screen:

```
fn touches(
    touches: Res<Touches>,
) {
    // There is a lot more information available, see the API docs.
    // This example only shows some very basic things.

    for finger in touches.iter() {
        if touches.just_pressed(finger.id()) {
            println!("A new touch with ID {} just began.", finger.id());
        }
        println!(
            "Finger {} is at position ({}), started from ({}).",
            finger.id(),
            finger.position().x,
            finger.position().y,
            finger.start_position().x,
            finger.start_position().y,
        );
    }
}
```

Alternatively, you can use `TouchInput` events:

```
fn touch_events(
    mut touch_evr: EventReader<TouchInput>,
) {
    use bevy::input::touch::TouchPhase;
    for ev in touch_evr.iter() {
        // in real apps you probably want to store and track touch ids somewhere
        match ev.phase {
            TouchPhase::Started => {
                println!("Touch {} started at: {:?}", ev.id, ev.position);
            }
            TouchPhase::Moved => {
                println!("Touch {} moved to: {:?}", ev.id, ev.position);
            }
            TouchPhase::Ended => {
                println!("Touch {} ended at: {:?}", ev.id, ev.position);
            }
            TouchPhase::Cancelled => {
                println!("Touch {} cancelled at: {:?}", ev.id, ev.position);
            }
        }
    }
}
```

Drag-and-Drop (Files)

Relevant official examples: [drag_and_drop](#).

Bevy supports the Drag-and-Drop gesture common on most desktop operating systems, but only for files, not arbitrary data / objects.

If you drag a file (say, from the file manager app) into a Bevy app, Bevy will produce a [FileDragAndDrop](#) event, containing the path of the file that was dropped in.

Usually, in a graphical app, you may want to do different things depending on where it was dropped. For this, you can [check the mouse cursor position](#), or use a Bevy UI [Interaction](#).

For example, here is how to detect if a file was dropped onto a special UI widget/element (which we identify with a custom marker [component](#)):

```
#[derive(Component)]
struct MyDropTarget;

fn file_drop(
    mut dnd_evr: EventReader<FileDragAndDrop>,
    query_ui_droptarget: Query<&Interaction, With<MyDropTarget>>,
) {
    for ev in dnd_evr.iter() {
        println!("{:?}", ev);
        if let FileDragAndDrop::DroppedFile { id, path_buf } = ev {
            println!("Dropped file with path: {:?}", path_buf);

            if id.is_primary() {
                // it was dropped over the main window
            }

            for interaction in query_ui_droptarget.iter() {
                if *interaction == Interaction::Hovered {
                    // it was dropped over our UI element
                    // (our UI element is being hovered over)
                }
            }
        }
    }
}
```

MIDI (Musical Instrument)

Bevy does not yet have this built-in, but there is a [3rd-party plugin](#) available: `bevy_midi`.

This chapter covers topics related to working with the application's OS window.

Page coming soon...

In the meantime, you can learn from Bevy's [examples](#).

See the `window_settings` example.

Relevant official examples: `clear_color`.

[Click here for the full example code.](#)

Use the `ClearColor` resource to choose the background color.

```
fn main() {
    App::new()
        .insert_resource(ClearColor(Color::rgb(0.4, 0.4, 0.4)))
        .add_plugins(DefaultPlugins)
        .run();
}
```

Grabbing the Mouse

[Click here for the full example code.](#)

Relevant official examples: `mouse_grab`.

You can lock/release the mouse cursor using bevy's [window settings API](#).

Here is an example that locks and hides the cursor in the primary window on [mouse click](#) and releases it when [pressing Esc](#):

```
fn cursor_grab_system(
    mut windows: ResMut<Windows>,
    btn: Res<Input<MouseButton>>,
    key: Res<Input<KeyCode>>,
) {
    let window = windows.get_primary_mut().unwrap();

    if btn.just_pressed(MouseButton::Left) {
        window.set_cursor_lock_mode(true);
        window.set_cursor_visibility(false);
    }

    if key.just_pressed(KeyCode::Escape) {
        window.set_cursor_lock_mode(false);
        window.set_cursor_visibility(true);
    }
}
```

Setting the Window Icon

[Click here for the full example code.](#)

You might want to set a custom Window Icon. On Windows and Linux, this is the icon image shown in the window title bar (if any) and task bar (if any).

Unfortunately, Bevy does not yet provide an easy and ergonomic built-in way to do this. However, it can be done via the `winit` APIs.

The way shown here is quite hacky. To save on code complexity, instead of using Bevy's asset system to load the image in the background, we bypass the assets system and directly load the file using the `image` library.

There is some WIP on adding a proper API for this to Bevy; see [PR #2268](#) and [Issue #1031](#).

This example shows how to set the icon for the primary/main window, from a Bevy startup system.

```
use bevy::window::WindowId;
use bevy::winit::WinitWindows;
use winit::window::Icon;

fn set_window_icon(
    // we have to use `NonSend` here
    windows: NonSend<WinitWindows>,
) {
    let primary = windows.get_window(WindowId::primary()).unwrap();

    // here we use the `image` crate to load our icon data from a png file
    // this is not a very bevy-native solution, but it will do
    let (icon_rgba, icon_width, icon_height) = {
        let image = image::open("my_icon.png")
            .expect("Failed to open icon path")
            .into_rgba8();
        let (width, height) = image.dimensions();
        let rgba = image.into_raw();
        (rgba, width, height)
    };

    let icon = Icon::from_rgba(icon_rgba, icon_width, icon_height).unwrap();

    primary.set_window_icon(Some(icon));
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_startup_system(set_window_icon)
        .run();
}
```

Note: that `WinitWindows` is a [non-send resource](#).

Note: you need to add `winit` to your project's dependencies, and it must be the same version as the one used by Bevy. You can use `cargo tree` or check `Cargo.lock` to see which is the correct version. As of Bevy 0.7, that should be `winit = "0.26"`.

This chapter covers topics relevant to making 2D games with Bevy.

2D Camera Setup

Page coming soon...

In the meantime, you can learn from Bevy's [examples](#).

Page coming soon...

In the meantime, you can learn from Bevy's [examples](#).

This chapter covers topics relevant to making 3D games with Bevy.

3D Camera Setup

Page coming soon...

In the meantime, you can learn from Bevy's [examples](#).

Relevant official examples: `load_gltf`, `update_gltf_scene`.

Bevy uses the GLTF 2.0 file format for 3D assets.

(other formats such as Wavefront OBJ may be unofficially available via 3rd-party plugins)

Quick-Start: Spawning 3D Models into your World

The simplest use case is to just load a "3D model" and spawn it into the game world.

"3D models" can often be complex, consisting of multiple parts. Think of a house: the windows, roof, doors, etc., are separate pieces, that are likely made of multiple meshes, materials, and textures. Bevy would technically need multiple ECS Entities to represent and render the whole thing.

This is why your GLTF "model" is represented by Bevy as a Scene. This way, you can easily spawn it, and Bevy will create all the relevant [child entities](#) and configure them correctly.

So that you can treat the whole thing as "a single object" and position it in the world, you can just [spawn it under a parent entity](#), and use its [transform](#).

```
fn spawn_gltf(
    mut commands: Commands,
    ass: Res<AssetServer>,
) {
    // note that we have to include the `Scene0` label
    let my_gltf = ass.load("my.glb#Scene0");

    // to be able to position our 3d model:
    // spawn a parent entity with a TransformBundle
    // and spawn our gltf as a scene under it
    commands.spawn_bundle(TransformBundle {
        local: Transform::from_xyz(2.0, 0.0, -5.0),
        global: GlobalTransform::identity(),
    }).with_children(|parent| {
        parent.spawn_scene(my_gltf);
    });
}
```

If your GLTF Scene represents "a whole level/map", rather than "an individual 3d model", and you don't need to move it around, you can just spawn the scene directly, without creating a parent entity.

Also, this example assumes that you have a simple GLTF file containing only one "default scene". GLTF is a very flexible file format. A single file can contain many "models" or more complex "scenes". To get a better understanding of GLTF and possible workflows, read the rest of this page. :)

Introduction to GLTF

GLTF is a modern open standard for exchanging 3D assets between different 3D software applications, like game engines and 3D modeling software.

The GLTF file format has two variants: human-readable ascii/text (`*.gltf`) and binary (`*.glb`). The binary format is more compact and preferable for packaging the assets with your game. The text format may be useful for development, as it can be easier to manually inspect using a text editor.

A GLTF file can contain many objects (sub-assets): meshes, materials, textures, scenes. When loading a GLTF file, Bevy will load all of the assets contained inside. They will be mapped to the [appropriate Bevy-internal asset types](#).

The GLTF sub-assets

GLTF terminology can be confusing, as it sometimes uses the same words to refer to different things, compared to Bevy. This section will try explain the various GLTF terms.

To understand everything, it helps to mentally consider how these concepts are represented in different places: in your 3D modeling software (like Blender), in the GLTF file itself, and in Bevy.

GLTF **Scenes** are what you spawn into your game world. This is typically what you see on the screen in your 3D modeling software. Scenes combine all of the data needed for the game engine to create all the needed entities to represent what you want. Conceptually, think of a scene as one "unit". Depending on your use case, this could be one "3d model", or even a whole map or game level. In Bevy, these are represented as Bevy Scenes with all the child ECS entities.

GLTF Scenes are composed of GLTF **Nodes**. These describe the "objects" in the scene, typically GLTF Meshes, but can also be other things like Cameras and Lights. Each GLTF Node has a transform for positioning it in the scene. GLTF Nodes do not have a core Bevy equivalent; Bevy just uses this data to create the ECS Entities inside of a Scene. Bevy has a special `GltfNode` asset type, if you need access to this data.

GLTF **Meshes** represent one conceptual "3D object". These correspond to the "objects" in your 3D modeling software. GLTF Meshes may be complex and composed of multiple smaller pieces, called GLTF Primitives, each of which may use a different Material. GLTF Meshes do not have a core Bevy equivalent, but there is a special `GltfMesh` asset type, which describes the primitives.

GLTF **Primitives** are individual "units of 3D geometry", for the purposes of rendering. They contain the actual geometry / vertex data, and reference the Material to be used when drawing. In Bevy, each GLTF Primitive is represented as a Bevy `Mesh` asset, and must be spawned as a separate ECS Entity to be rendered.

GLTF **Materials** describe the shading parameters for the surfaces of your 3D models. They have full support for Physically-Based Rendering (PBR). They also reference the textures to use. In Bevy, they are represented as `StandardMaterial` assets, as used by the Bevy PBR 3D renderer.

GLTF **Textures** (images) can be embedded inside the GLTF file, or stored externally in separate image files alongside it. For example, you can have your textures as separate PNG or JPEG files for

ease of development, or package them all inside the GLTF file for ease of distribution. In Bevy, GLTF textures are loaded as Bevy `Image` assets.

GLTF **Samplers** describe the settings for how the GPU should use a given Texture. Bevy does not keep these separate; this data is stored inside the Bevy `Image` asset (the `sampler` field of type `SamplerDescriptor`).

GLTF **Animations** describe animations that interpolate various values, such as transforms or mesh skeletons, over time. In Bevy, these are loaded as `AnimationClip` assets.

GLTF Usage Patterns

A single GLTF file can contain any number of sub-assets of any of the above types, referring to each other however they like.

Because GLTF is so flexible, it is up to you how to structure your assets.

A single GLTF file might be used:

- To represent a single "3D model", containing a single GLTF Scene with the model, so you can spawn it into your game.
- To represent a whole level, as a GLTF Scene, possibly also including the camera. This lets you load and spawn a whole level/map at once.
- To represent sections of a level/map, such as a rooms, as separate GLTF Scenes. They can share meshes and textures if needed.
- To contain a set of many different "3D models", each as a separate GLTF Scene. This lets you load and manage the whole collection at once and spawn them individually as needed.
- ... others?

Tools for Creating GLTF Assets

If you are using a recent version of Blender (2.8+) for 3D modeling, GLTF is supported out of the box. Just export and choose GLTF as the format.

For other tools, you can try these exporter plugins:

- [Old Blender \(2.79\)](#)
- [3DSMax](#)
- [Autodesk Maya](#)
 - (or this [alternative](#))

Using GLTF Sub-Assets in Bevy

The various sub-assets contained in a GLTF file can be addressed in two ways:

- by index (integer id, in the order they appear in the file)
- by name (text string, the names you set in your 3D modeling software when creating the asset,

which can be exported into the GLTF)

To get handles to the respective assets in Bevy, you can use the `Gltf` "master asset", or alternatively, [AssetPath with Labels](#).

`Gltf` master asset

If you have a complex GLTF file, this is likely the most flexible and useful way of navigating its contents and using the different things inside.

You have to wait for the GLTF file to load, and then use the `Gltf` asset.

```
use bevy::gltf::Gltf;

/// Helper resource for tracking our asset
struct MyAssetPack(Handle<Gltf>);

fn load_gltf(
    mut commands: Commands,
    ass: Res<AssetServer>,
) {
    let gltf = ass.load("my_asset_pack.glb");
    commands.insert_resource(MyAssetPack(gltf));
}

fn spawn_gltf_objects(
    mut commands: Commands,
    my: Res<MyAssetPack>,
    assets_gltf: Res<Assets<Gltf>>,
) {
    // if the GLTF has loaded, we can navigate its contents
    if let Some(gltf) = assets_gltf.get(&my.0) {
        // spawn the first scene in the file
        commands.spawn_scene(gltf.scenes[0].clone());

        // spawn the scene named "YellowCar"
        // do it under a parent entity, to position it in the world
        commands.spawn_bundle(TransformBundle {
            local: Transform::from_xyz(1.0, 2.0, 3.0),
            global: GlobalTransform::identity(),
        }).with_children(|parent| {
            parent.spawn_scene(gltf.named_scenes["YellowCar"].clone());
        });

        // PERF: the `.clone()`s are just for asset handles, don't worry :)
    }
}
```

For a more convoluted example, say we want to directly create a 3D PBR entity, for whatever reason. (This is not recommended; you should probably just use scenes)

```
use bevy::gltf::GltfMesh;
```

```
fn gltf_mesh_entity(<
```

```
fn gltf_manual_entity(
    mut commands: Commands,
    my: Res<MyAssetPack>,
    assets_gltf: Res<Assets<Gltf>>,
    assets_gltfmesh: Res<Assets<GltfMesh>>,
) {
    if let Some(gltf) = assets_gltf.get(&my.0) {
        // Get the GLTF Mesh named "CarWheel"
        // (unwrap safety: we know the GLTF has loaded already)
        let carwheel = assets_gltfmesh.get(&gltf.named_meshes["CarWheel"]).unwrap();

        // Spawn a PBR entity with the mesh and material of the first GLTF Primitive
        commands.spawn_bundle(PbrBundle {
            mesh: carwheel.primitives[0].mesh.clone(),
            // (unwrap: material is optional, we assume this primitive has one)
            material: carwheel.primitives[0].material.clone().unwrap(),
            ..Default::default()
        });
    }
}
```

AssetPath with Labels

This is another way to access specific sub-assets. It is less reliable, but may be easier to use in some cases.

Use the `AssetServer` to convert a path string into a `Handle`.

The advantage is that you can get handles to your sub-assets immediately, even if your GLTF file hasn't loaded yet.

The disadvantage is that it is more error-prone. If you specify a sub-asset that doesn't actually exist in the file, or mis-type the label, or use the wrong label, it will just silently not work. Also, currently only using a numerical index is supported. You cannot address sub-assets by name.

```
fn use_gltf_things(
    mut commands: Commands,
    ass: Res<AssetServer>,
) {
    // spawn the first scene in the file
    let scene0 = ass.load("my_asset_pack.glb#Scene0");
    commands.spawn_scene(scene0);

    // spawn the second scene under a parent entity
    // (to move it)
    let scene1 = ass.load("my_asset_pack.glb#Scene1");
    commands.spawn_bundle(TransformBundle {
        local: Transform::from_xyz(1.0, 2.0, 3.0),
        global: GlobalTransform::identity(),
    }).with_children(|parent| {
        parent.spawn_scene(scene1);
    });
}
```

The following asset labels are supported (`{}` is the numerical index):

- `Scene{}` : GLTF Scene as Bevy `Scene`
- `Node{}` : GLTF Node as `GltfNode`
- `Mesh{}` : GLTF Mesh as `GltfMesh`
- `Mesh{}/Primitive{}` : GLTF Primitive as Bevy `Mesh`
- `Texture{}` : GLTF Texture as Bevy `Image`
- `Material{}` : GLTF Material as Bevy `StandardMaterial`
- `DefaultMaterial` : as above, if the GLTF file contains a default material with no index
- `Animation{}` : GLTF Animation as Bevy `AnimationClip`
- `Skin{}` : GLTF mesh skin as Bevy `SkinnedMeshInverseBindposes`

The `GltfNode` and `GltfMesh` asset types are only useful to help you navigate the contents of your GLTF file. They are not core Bevy renderer types, and not used by Bevy in any other way. The Bevy renderer expects Entities with `MaterialMeshBundle`; for that you need the `Mesh` and `StandardMaterial`.

Bevy Limitations

Bevy does not fully support all features of the GLTF format and has some specific requirements about the data. Not all GLTF files can be loaded and rendered in Bevy. Unfortunately, in many of these cases, you will not get any error or diagnostic message.

Commonly-encountered limitations:

- Textures embedded in ascii (`*.gltf`) files (base64 encoding) cannot be loaded. Put your textures in external files, or use the binary (`*.glb`) format.
- Mipmaps are only supported if the texture files (in KTX2 or DDS format) contain them. The GLTF spec requires missing mipmap data to be generated by the game engine, but Bevy does not support this. If your assets are missing mipmaps, textures will look grainy/noisy.
- Bevy's renderer requires all meshes/primitives to have per-vertex positions, UVs, and normals. Make sure all of this data is included.
- Meshes/primitives without textures (if the material is just a solid color) must still include UVs regardless. Bevy will not render meshes without UVs.
- When using normal maps in your material, tangents must also be included in the mesh. Assets with normal maps but without tangents are valid; other software would typically autogenerate the tangents if they are missing, but Bevy does not support this yet. Be sure to tick the checkbox for including tangents when exporting.
- Spot lights are not supported. Bevy currently only has Point lights and Directional lights.

This list is not exhaustive. There may be other unsupported scenarios that I did not know of or forgot to include here. :)

Bevy Programming Framework

This chapter presents the features of the Bevy core programming framework. This covers the ECS (Entity Component System), App and Scheduling.

For examples of programming patterns and idioms, see the [Programming Patterns](#) chapter.

All the knowledge of this chapter is useful even if you want to use Bevy as something other than a game engine. For example: using just the ECS for a scientific simulation.

Hence, this chapter does not cover the game-engine parts of Bevy. Those features are covered in other chapters of the book, like the [Bevy Game Engine Core](#) chapter.

Includes concise explanations of each core concept, with code snippets to show how it might be used in a game. Care is taken to point out any important considerations for using each feature and to recommend known good practices.

ECS as a Data Structure

Relevant official examples: `ecs_guide`.

Also check out the complete game examples: `alien_cake_addict`, `breakout`.

Bevy stores and manages all your data for you, using the Bevy ECS (Entity-Component System).

Conceptually, you can think of it by analogy with tables, like in a database or spreadsheet. Your different data types (Components) are like the "columns" of a table, and there can be arbitrarily many "rows" (Entities) containing values / instances of each component.

For example, you could create a `Health` component for your game. You could then have many entities representing different things in your game, such as the player, NPCs, or monsters, all of which can have a `Health` value (as well as other relevant components).

This makes it easy to write game logic (`Systems`) that can operate on any entity with the necessary components (such as a health/damage system for anything that has `Health`), regardless of whether that's the player, an NPC, or a monster (or anything else). This makes your game logic very flexible and reusable.

The set / combination of components that a given entity has, is called the entity's Archetype.

Note that entities aren't limited to just "objects in the game world". The ECS is a general-purpose data structure. You can create entities and components to store any data.

Performance

Bevy has a smart scheduling algorithm that runs your systems in parallel as much as possible. It does that automatically, when your functions don't require conflicting access to the same data. Your game will scale to run on multiple CPU cores "for free"; that is, without requiring extra development effort from you.

To improve the chances for parallelism, you can make your data and code more granular. Split your data into smaller types / `struct`s. Split your logic into multiple smaller systems / functions. Have each system access only the data that is relevant to it. The fewer access conflicts, the faster your game will run.

The general rule of thumb for Bevy performance is: more granular is better.

Note for Programmers coming from Object-Oriented Languages

You may be used to thinking in terms of "object classes". For example, you might be tempted to define a big monolithic `struct Player` containing all the fields / properties of the player.

In Bevy, this is considered bad practice, because doing it that way can make it more difficult to work with your data, and limit performance.

Instead, you should make things granular, when different pieces of data may be accessed independently.

For example, represent the Player in your game as an entity, composed of separate component types (separate `struct`s) for things like the health, XP, or whatever is relevant to your game. You can also attach standard Bevy components like `Transform` ([transforms explained](#)) to it.

This will make it easier for you to develop your systems (game logic / behaviors), as well as make your game's runtime performance better.

However, something like a `Transform`, or a set of coordinates, still makes sense as a single `struct`, because its fields are not likely to be useful independently.

Entities

Entities are just a [simple integer ID](#), that identifies a particular set of component values.

To create ("spawn") new entities, use `Commands`.

Components

Components are the data associated with entities.

To create a new component type, simply define a Rust `struct` or `enum`, and derive the `Component` trait.

```
#[derive(Component)]
struct Health {
    hp: f32,
    extra: f32,
}
```

Types must be unique -- an entity can only have one component per Rust type.

Use wrapper (newtype) structs to make unique components out of simpler types:

```
#[derive(Component)]
struct PlayerXp(u32);

#[derive(Component)]
struct PlayerName(String);
```

You can use empty structs to help you identify specific entities. These are known as "marker components". Useful with [query filters](#).

```
/// Add this to all menu ui entities to help identify them
#[derive(Component)]
struct MainMenuUI;

/// Marker for hostile game units
#[derive(Component)]
struct Enemy;

/// This will be used to identify the main player entity
#[derive(Component)]
struct Player;
```

Components can be accessed from [systems](#), using [queries](#).

You can add/remove components on existing entities, using `Commands`.

Component Bundles

Bundles are like "templates", to make it easy to create entities with a common set of components.


```
#[derive(Bundle)]
struct PlayerBundle {
    xp: PlayerXp,
    name: PlayerName,
    health: Health,
    _p: Player,

    // We can nest/include another bundle.
    // Add the components for a standard Bevy Sprite:
    #[bundle]
    sprite: SpriteSheetBundle,
}
```

Bevy also considers arbitrary tuples of components as bundles:

```
(ComponentA, ComponentB, ComponentC)
```

Note that you cannot [query](#) for a whole bundle. Bundles are just a convenience when creating the entities. Query for the individual component types that your [system](#) needs to access.

Resources

Relevant official examples: `ecs_guide`.

Resources allow you to store a single global instance of some data type, independently of entities.

Use them for data that is truly global for your app, such as configuration / settings.

Any Rust type (`struct` or `enum`) can be used as a resource. Currently, no special trait or derive is required, but that may change in future Bevy versions (similar to how `Components` require it).

Types must be unique; there can only be one instance of a given type.

```
struct GoalsReached {
    main_goal: bool,
    bonus: bool,
}
```

Resources can be accessed from `systems`, using `Res` / `ResMut`.

Resource Initialization

Implement `Default` for simple resources:

```
#[derive(Default)]
struct StartingLevel(usize);
```

For resources that need complex initialization, implement `FromWorld`:

```
struct MyFancyResource { /* stuff */ }

impl FromWorld for MyFancyResource {
    fn from_world(world: &mut World) -> Self {
        // You have full access to anything in the ECS from here.
        // For instance, you can mutate other resources:
        let mut x = world.get_resource_mut::().unwrap();
        x.do_mut_stuff();

        MyFancyResource { /* stuff */ }
    }
}
```

You can initialize your resources at `App` creation:

```
fn main() {
    App::new()
```

```
// ...  
  
// if it implements `Default` or `FromWorld`  
.init_resource::()  
// if not, or if you want to set a specific value  
.insert_resource(StartingLevel(3))  
  
// ...  
.run();  
}
```

Commands can be used to create/remove resources from inside a system:

```
commands.insert_resource(GoalsReached { main_goal: false, bonus: false });  
commands.remove_resource::();
```

If you insert a resource of a type that already exists, it will be overwritten.

Usage Advice

The choice of when to use entities/components vs. resources is typically about how you want to access the data: globally from anywhere (resources), or using ECS patterns (entities/components).

Even if there is only one of a certain thing in your game (such as the player in a single-player game), it can be a good fit to use an entity instead of resources, because entities are composed of multiple components, some of which can be common with other entities. This can make your game logic more flexible. For example, you could have a "health/damage system" that works with both the player and enemies.

Systems

Relevant official examples: `ecs_guide`, `startup_system`, `system_param`.

Systems are functions you write, which are run by Bevy.

This is where you implement all your game logic.

These functions can only take [special parameter types](#), to specify what you need access to. If you use [unsupported parameter types](#) in your function, you will get confusing compiler errors!

Some of the options are:

- accessing [resources](#) using `Res` / `ResMut`
- accessing [components of entities](#) using [queries](#) (`Query`)
- creating/destroying entities, components, and resources using [Commands](#) (`Commands`)
- sending/receiving [events](#) using `EventWriter` / `EventReader`

```
fn debug_start(
    // access resource
    start: Res<StartingLevel>
) {
    eprintln!("Starting on level {:?}", *start);
}
```

System parameters can be grouped into tuples (which can be nested). This is useful for organization.

```
fn complex_system(
    (a, mut b): (Res<ResourceA>, ResMut<ResourceB>),
    // this resource might not exist, so wrap it in an Option
    mut c: Option<ResMut<ResourceC>>,
) {
    if let Some(mut c) = c {
        // do something
    }
}
```

Your function can have a maximum of 16 total parameters. If you need more, group them into tuples to work around the limit. Tuples can contain up to 16 members, but can be nested indefinitely.

Runtime

To run your systems, you need to add them to Bevy via the [app builder](#):

```
fn main() {
    App::new()
```

```
// ...

// run it only once at launch
.add_startup_system(init_menu)
.add_startup_system(debug_start)

// run it every frame update
.add_system(move_player)
.add_system(enemies_ai)

// ...
.run();
}
```

The above is enough for simple projects.

As your project grows more complex, you might want to enhance your app builder with some of the powerful tools that Bevy offers for managing when/how your systems run, such as: [explicit ordering](#) with [labels](#), [system sets](#), [states](#), [run criteria](#), and [stages](#).

Relevant official examples: `ecs_guide`.

Queries let you access [components of entities](#).

```
fn check_zero_health(
    // access entities that have `Health` and `Transform` components
    // get read-only access to `Health` and mutable access to `Transform`
    // optional component: get access to `Player` if it exists
    mut query: Query<(&Health, &mut Transform, Option<&Player>>>,
) {
    // get all matching entities
    for (health, mut transform, player) in query.iter_mut() {
        eprintln!("Entity at {} has {} HP.", transform.translation, health.hp);

        // center if hp is zero
        if health.hp <= 0.0 {
            transform.translation = Vec3::ZERO;
        }

        if let Some(player) = player {
            // the current entity is the player!
            // do something special!
        }
    }
}
```

Get the [components](#) associated with a specific [entity](#):

```
if let Ok((health, mut transform)) = query.get_mut(entity) {
    // do something with the components
} else {
    // the entity does not have the components from the query
}
```

Get the IDs ([Entity](#)) of the entities you access with your queries:

```
// add `Entity` to `Query` to get Entity IDs
fn query_entities(q: Query<(Entity, /* ... */>>) {
    for (e, /* ... */) in q.iter() {
        // `e` is the Entity ID of the entity we are accessing
    }
}
```

If you know that the query should only ever match a single entity, you can use `single` / `single_mut`, instead of iterating:

```
fn query_player(mut q: Query<(&Player, &mut Transform)>) {
    let (player, mut transform) = q.single_mut();

    // do something with the player and its transform
}
```

(this will panic if the query matches more than one entity)

Bundles

Queries work with individual components. If you created an entity using a [bundle](#), you need to query for the specific components from that bundle that you care about.

A common beginner mistake is to query for the bundle type!

Query Filters

Add query filters to narrow down the entities you get from the query.

Use `With` / `Without` to only get entities that have specific components.

```
fn debug_player_hp(
    // access the health, only for friendly players, optionally with name
    query: Query<(&Health, Option<&PlayerName>), (With<Player>, Without<Enemy>)>,
) {
    // get all matching entities
    for (health, name) in query.iter() {
        if let Some(name) = name {
            eprintln!("Player {} has {} HP.", name.0, health.hp);
        } else {
            eprintln!("Unknown player has {} HP.", health.hp);
        }
    }
}
```

Multiple filters can be combined:

- in a tuple to apply all of them (AND logic)
- using the `or<(...)>` wrapper to detect any of them (OR logic).
 - (note the tuple inside)

Commands

Relevant official examples: [ecs_guide](#).

Use [Commands](#) to spawn/despawn entities, add/remove components on existing entities, manage resources.

These actions do not take effect immediately; they are queued to be performed later when it is safe to do so. See: [stages](#).

(if you are not using stages, that means your other [systems](#) will see them on the next frame update)

```
fn spawn_player(  
    mut commands: Commands,  
    f
```



```

    } {
        // manage resources
        commands.insert_resource(GoalsReached { main_goal: false, bonus: false });
        commands.remove_resource::();

        // create a new entity using `spawn`
        let entity_id = commands.spawn()
            // add a component
            .insert(ComponentA)
            // add a bundle
            .insert_bundle(MyBundle::default())
            // get the Entity ID
            .id();

        // shorthand for creating an entity with a bundle
        commands.spawn_bundle(PlayerBundle {
            name: PlayerName("Henry".into()),
            xp: PlayerXp(1000),
            health: Health {
                hp: 100.0, extra: 20.0
            },
            _p: Player,
            sprite: Default::default(),
        });

        // spawn another entity
        // NOTE: tuples of arbitrary components are valid bundles
        let other = commands.spawn_bundle((
            ComponentA::default(),
            ComponentB::default(),
            ComponentC::default(),
        )).id();

        // add/remove components of an existing entity
        commands.entity(entity_id)
            .insert(ComponentB)
            .remove::()
            .remove_bundle::();

        // despawn an entity
        commands.entity(other).despawn();
    }

fn make_all_players_hostile(
    mut commands: Commands,
    query: Query<Entity, With<Player>>,
) {
    for entity in query.iter() {
        // add an `Enemy` component to the entity
        commands.entity(entity).insert(Enemy);
    }
}

```

Events

Relevant official examples: [event](#).

Send data between systems! Let your [systems](#) communicate with each other!

To send events, use an [EventWriter<T>](#). To receive events, use an [EventReader<T>](#).

Every reader tracks the events it has read independently, so you can handle the same events from multiple [systems](#).

```
struct LevelUpEvent(Entity);

fn player_level_up(
    mut ev_levelup: EventWriter<LevelUpEvent>,
    query: Query<(Entity, &PlayerXp)>,
) {
    for (entity, xp) in query.iter() {
        if xp.0 > 1000 {
            ev_levelup.send(LevelUpEvent(entity));
        }
    }
}

fn debug_levelups(
    mut ev_levelup: EventReader<LevelUpEvent>,
) {
    for ev in ev_levelup.iter() {
        eprintln!("Entity {:?} leveled up!", ev.0);
    }
}
```

You need to add your custom event types via the [app builder](#):

```
fn main() {
    App::new()
        // ...
        .add_event::<LevelUpEvent>()
        .add_system(player_level_up)
        .add_system(debug_levelups)
        // ...
        .run();
}
```

Events should be your go-to data flow tool. As events can be sent from any system and received by multiple systems, they are *extremely* versatile.

Possible Pitfalls

Beware of frame delay / 1-frame-lag. This can occur if Bevy runs the receiving system before the sending system. The receiving system will only get a chance to receive the events on the next frame

update. If you need to ensure that events are handled immediately / during the same frame, you can use [explicit system ordering](#).

Events don't persist. They are stored until the end of the next frame, after which they are lost. If your systems do not handle events every frame, you could miss some.

The advantage of this design is that you don't have to worry about excessive memory use from unhandled events.

If you don't like this, [you can have manual control over when events are cleared](#) (at the risk of leaking / wasting memory if you forget to clear them).

App Builder (main function)

Relevant official examples: All of them ;)

In particular, check out the complete game examples: `alien_cake_addict`, `breakout`.

To enter the bevy runtime, you need to configure an `App`. The app is how you define the structure of all the things that make up your project: `plugins`, `systems`, `event` types, `states`, `stages`...

Technically, the `App` contains the ECS World(s) (where all the data is stored) and Schedule(s) (where all the `systems` to run are stored). For advanced use-cases, `Sub-apps` are a way to have more than one ECS World and Schedule.

`Local resources` do not need to be registered. They are part of their respective `systems`.

`Component` types do not need to be registered.

Schedules cannot (yet) be modified at runtime; all `systems` you want to run must be added/configured in the `App` ahead of time.

The data in the ECS World can be modified at any time; create/destroy your `entities` and `resources`, from `systems` using `Commands`, or `exclusive systems` using `direct World access`.

`Resources` can also be initialized ahead of time, here in the `App` builder.

You also need to add the `plugin group` with Bevy's built-in functionality: either `DefaultPlugins` if you are making a full game/app, or `MinimalPlugins` for something like a headless server.

Note that there are some special `configuration resources` that must be added first, if you would like to use them, to take effect.

```
fn main() {  
    App::new()
```

```

// make sure to add any config resources first, before Bevy:
.insert_resource(WindowDescriptor {
    // ...
    ..Default::default()
}) // etc...

// Bevy itself:
.add_plugins(DefaultPlugins)

// resources:
.insert_resource(StartingLevel(3))
// if it implements `Default` or `FromWorld`
.init_resource::()

// events:
.add_event::()

// systems to run once at startup:
.add_startup_system(spawn_player)

// systems to run each frame:
.add_system(player_level_up)
.add_system(debug_levelups)
.add_system(debug_stats_change)
// ...

// launch the app!
.run();
}

```

Quitting the App

To cleanly shut down bevy, send an `AppExit` event from any `system`:

```
use bevy::app::AppExit;

fn exit_system(mut exit: EventWriter<AppExit>) {
    exit.send(AppExit);
}
```

For prototyping, bevy provides a system you can [add to your `App`](#), to exit on pressing the `Esc` key:

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_system(bevy::input::system::exit_on_esc_system)
        .run();
}
```

Relevant official examples: [ecs_guide](#).

Local resources allow you to have per-[system](#) data.

[Local<T>](#) is a system parameter similar to [ResMut<T>](#), which gives you full mutable access to an instance of some data type, that is independent from entities and components.

[Res<T>](#) / [ResMut<T>](#) refer to a single global instance of the type, shared between all systems. On the other hand, every [Local<T>](#) parameter is a separate instance, exclusively for that system.

```
#[derive(Default)]
struct MyState;

fn my_system1(mut local: Local<MyState>) {
    // you can do anything you want with the local here
}

fn my_system2(mut local: Local<MyState>) {
    // the local in this system is a different instance
}
```

The type must implement [Default](#) or [FromWorld](#). It is automatically initialized.

A system can have multiple [Local](#)s of the same type.

Specify an initial value

[Local<T>](#) is always automatically initialized using the default value for the type.

If you need specific data, you can use a closure instead. Rust closures that take system parameters are valid Bevy systems, just like standalone functions. Using a closure allows you to "move data into the function".

This example shows how to initialize some data to configure a system, without using [Local<T>](#):

```
#[derive(Default)]
struct MyConfig {
```

```

    magic: usize,
}

fn my_system(
    mut cmd: Commands,
    my_res: Res<MyStuff>,
    // note this isn't a valid system parameter
    config: &MyConfig,
) {
    // TODO: do stuff
}

fn main() {
    let config = MyConfig {
        magic: 420,
    };

    App::new()
        // create a "move closure", so we can use the `config`
        // variable that we created above
        .add_system(move |cmd: Commands, res: Res<MyStuff>| {
            // call our function from inside the closure
            my_system(cmd, res, &config);
        })
        .run();
}

```


Relevant official examples: `plugin`, `plugin_group`.

As your project grows, it can be useful to make it more modular. You can split it into "plugins".

Plugins are simply collections of things to be added to the [App Builder](#).

```
struct MyPlugin;

impl Plugin for MyPlugin {
    fn build(&self, app: &mut App) {
        app
            .init_resource::()
            .add_event::()
            .add_startup_system(plugin_init)
            .add_system(my_system);
    }
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_plugin(MyPlugin)
        .run();
}
```

For internal organization in your own project, the main value of plugins comes from not having to declare all your Rust types and functions as `pub`, just so they can be accessible from `fn main` to be added to the app builder. Plugins let you add things to your [app](#) from multiple different places, like separate Rust files / modules.

You can decide how plugins fit into the architecture of your game.

Some suggestions:

- Create plugins for different [states](#).
- Create plugins for various sub-systems, like physics or input handling.

Plugin groups

Plugin groups register multiple plugins at once. Bevy's `DefaultPlugins` and `MinimalPlugins` are examples of this. To create your own plugin group:

```
struct MyPluginGroup;
```

```

impl PluginGroup for MyPluginGroup {
    fn build(&mut self, group: &mut PluginGroupBuilder) {
        group
            .add(FooPlugin)
            .add(BarPlugin);
    }
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_plugins(MyPluginGroup)
        .run();
}

```

When adding a plugin group to the [app](#), you can disable some plugins while keeping the rest.

For example, if you want to manually set up logging (with your own `tracing` subscriber), you can disable Bevy's `LogPlugin`:

```

App::new()
    .add_plugins_with(DefaultPlugins, |plugins| {
        plugins.disable::<LogPlugin>()
    })
    .run();

```

Note that this simply disables the functionality, but it cannot actually remove the code to avoid binary bloat. The disabled plugins still have to be compiled into your program.

If you want to slim down your build, you should look at disabling Bevy's default [cargo features](#), or depending on the various Bevy sub-crates individually.

Publishing Crates

Plugins give you a nice way to publish Bevy-based libraries for other people to easily include into their projects.

If you intend to publish plugins as crates for public use, you should read [the official guidelines for plugin authors](#).

Don't forget to submit an entry to [Bevy Assets](#) on the official website, so that people can find your plugin more easily. You can do this by making a PR in [the Github repo](#).

If you are interested in supporting bleeding-edge Bevy (main), [see here for advice](#).

System Order of Execution

Bevy's scheduling algorithm is designed to deliver maximum performance by running as many systems as possible in parallel across the available CPU threads.

This is possible when the systems do not conflict over the data they need to access. However, when a system needs to have mutable (exclusive) access to a piece of data, other systems that need to access the same data cannot be run at the same time. Bevy determines all of this information from the system's function signature (the types of the parameters it takes).

In such situations, the order is *nondeterministic* by default. Bevy takes no regard for when each system will run, and the order could even change every frame!

Does it even matter?

In many cases, you don't need to worry about this.

However, sometimes you need to rely on specific systems to run in a particular order. For example:

- Maybe the logic you wrote in one of your systems needs any modifications done to that data by another system to always happen first?
- One system needs to receive [events](#) sent by another system.
- You are using [change detection](#).

In such situations, systems running in the wrong order typically causes their behavior to be delayed until the next frame. In rare cases, depending on your game logic, it may even result in more serious logic bugs!

It is up to you to decide if this is important.

With many things in typical games, such as juicy visual effects, it probably doesn't matter if they get delayed by a frame. It might not be worthwhile to bother with it. If you don't care, leaving the order ambiguous may also result in better performance.

On the other hand, for things like handling the player input controls, this would result in annoying lag, so you should probably fix it.

Explicit System Ordering

If a specific system must always run before or after some other systems, you can add ordering constraints:

```
fn main() {  
    App::new()  
        .add_plugins(DefaultPlugins)
```

```

.add_plugins(DefaultPlugins)

// order doesn't matter for these systems:
.add_system(particle_effects)
.add_system(npc_behaviors)
.add_system(enemy_movement)

.add_system(input_handling)

.add_system(
    player_movement
    // `player_movement` must always run before `enemy_movement`
    .before(enemy_movement)
    // `player_movement` must always run after `input_handling`
    .after(input_handling)
)
.run();
}

```

`.before` / `.after` may be used as many times as you need on one system.

Labels

For more advanced use cases, you can use [labels](#). Labels can either be strings, or custom types (like `enum`s) that derive `SystemLabel`.

This allows you to affect multiple systems at once, with the same constraints. You can place multiple labels on one system. You can also use the same label on multiple systems.

Each label is a reference point that other systems can be ordered around.

```

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
#EnumDiscriminators(6, transparent)

```

```
#[derive(SystemLabel)]
enum MyLabel {
    Input,
    Player,
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // create labels, because we want to have multiple affected systems
        .add_system(input_joystick.label(MyLabel::Input))
        .add_system(input_keyboard.label(MyLabel::Input))
        .add_system(input_touch.label(MyLabel::Input))

        // this will always run before anything labeled "input"
        .add_system(input_parameters.before(MyLabel::Input))

        // this will always run after anything labeled "input" and "map"
        // also give it a few labels it just in case
        .add_system(
            player_movement
            // can also just use strings
            .label("player_movement")
            .label(MyLabel::Player)
            .after(MyLabel::Input)
        )
        .run();
}
```

When you have multiple systems with common labels or ordering, it may be convenient to use [system sets](#).

Circular Dependencies

If you have multiple systems mutually depending on each other, then it is clearly impossible to resolve the situation completely like that.

You should try to redesign your game to avoid such situations, or just accept the consequences. You can at least make it behave predictably, using explicit ordering to specify the order you prefer.

System Sets

System Sets allow you to easily apply common properties to multiple systems, for purposes such as [labeling](#), [ordering](#), [run criteria](#), and [states](#).

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

    // group our input handling systems into a set
    .add_system_set(
        SystemSet::new()
            .label("input")
            .with_system(keyboard_input)
            .with_system(gamepad_input)
    )

    // our "net" systems should run before "input"
    .add_system_set(
        SystemSet::new()
            .label("net")
            .before("input")
            // individual systems can still have
            // their own labels (and ordering)
            .with_system(server_session.label("session"))
            .with_system(server_updates.after("session"))
    )

    // some ungrouped systems
    .add_system(player_movement.after("input"))
    .add_system(session_ui.after("session"))
    .add_system(smoke_particles)

    .run();
}
```

Change Detection

Relevant official examples: `component_change_detection`.

Bevy allows you to easily detect when data is changed. You can use this to perform actions in response to changes.

One of the main use cases is optimization – avoiding unnecessary work by only doing it if the relevant data has changed. Another use case is triggering special actions to occur on changes, like configuring something or sending the data somewhere.

Components

Filtering

You can make a [query](#) that only yields entities if specific [components](#) on them have been modified.

Use [query filters](#):

- `Added<T>` : detect new component instances
 - if the component was added to an existing entity
 - if a new entity with the component was spawned
- `Changed<T>` : detect component instances that have been changed
 - triggers when the component is accessed mutably
 - also triggers if the component is newly-added (as per `Added`)

(If you want to react to removals, see the page on [removal detection](#). It works differently and is much trickier to use.)

```
/// Print the stats of friendly players when they change
fn debug_stats_change(
    mut stats,
```

```

    query: Query<
        // components
        (&Health, &PlayerXp),
        // filters
        (Without<Enemy>, Or<(Changed<Health>, Changed<PlayerXp>>>)),
    >,
) {
    for (health, xp) in query.iter() {
        eprintln!(
            "hp: {}+{}, xp: {}",
            health.hp, health.extra, xp.0
        );
    }
}

/// detect new enemies and print their health
fn debug_new_hostiles(
    query: Query<(Entity, &Health), Added<Enemy>>,
) {
    for (entity, health) in query.iter() {
        eprintln!("Entity {:?} is now an enemy! HP: {}", entity, health.hp);
    }
}

```

Checking

If you want to access all the entities, as normal, regardless of if they have been modified, but you just want to check the status, you can use the special `ChangeTrackers<T>` query parameter.

```

/// Make sprites flash red on frames when the Health changes
fn debug_damage(
    mut query: Query<(&mut Sprite, ChangeTrackers<Health>>),
) {
    for (mut sprite, tracker) in query.iter_mut() {
        // detect if the Health changed this frame
        if tracker.is_changed() {
            sprite.color = Color::RED;
        } else {
            // extra check so we don't mutate on every frame without changes
            if sprite.color != Color::WHITE {
                sprite.color = Color::WHITE;
            }
        }
    }
}

```

This is useful for processing all entities, but doing different things depending on if they have been modified.

Resources

For [resources](#), change detection is provided via methods on the `Res` / `ResMut` system parameters.

```

fn check_res_changed(
    my_res: Res<MyResource>,
) {

```



```

    } {
        if my_res.is_changed() {
            // do something
        }
    }

fn check_res_added(
    // use Option, not to panic if the resource doesn't exist yet
    my_res: Option<Res<MyResource>>,
) {
    if let Some(my_res) = my_res {
        // the resource exists

        if my_res.is_added() {
            // it was just added
            // do something
        }
    }
}

```

Note that change detection cannot currently be used to detect [states](#) changes (via the `State` resource) (bug).

What gets detected?

`Changed` detection is triggered by `DerefMut`. Simply accessing components via a mutable query, without actually performing a `&mut` access, will *not* trigger it.

This makes change detection quite accurate. You can rely on it to optimize your game's performance, or to otherwise trigger things to happen.

Also note that when you mutate a component, Bevy does not track if the new value is actually different from the old value. It will always trigger the change detection. If you want to avoid that, simply check it yourself:

```

fn update_player_xp(
    mut query: Query<&mut PlayerXp>,
) {
    for mut xp in query.iter_mut() {
        let new_xp = maybe_lvl_up(&xp);

        // avoid triggering change detection if the value is the same
        if new_xp != *xp {
            *xp = new_xp;
        }
    }
}

```

Change detection works on a per-[system](#) granularity, and is reliable. A system will not detect changes that it made itself, only those done by other systems, and only if it has not seen them before (the changes happened since the last time it ran). If your system only runs sometimes (such as with [states](#) or [run criteria](#)), you do *not* have to worry about missing changes.

Possible Pitfalls

Beware of frame delay / 1-frame-lag. This can occur if Bevy runs the detecting system before the changing system. The detecting system will see the change the next time it runs, typically on the next frame update.

If you need to ensure that changes are handled immediately / during the same frame, you can use [explicit system ordering](#).

However, when detecting component additions with `Added<T>` (which are typically done using `Commands`), this is not enough; you need [stages](#).

Relevant official examples: [state](#).

Consider using the [iyes_loopless](#) crate, which provides an alternative implementation that does not suffer from the [usability issues](#) of the one in Bevy.

States allow you to structure the runtime "flow" of your app.

This is how you can implement things like:

- A menu screen or a loading screen
- Pausing / unpausing the game
- Different game modes
- ...

In every state, you can have different [systems](#) running. You can also add one-shot setup and cleanup systems to run when entering or exiting a state.

To use states, define an enum type and add [system sets](#) to your [app builder](#):

```
#[derive(Debug, Clone, Eq, PartialEq, Hash)]
enum AppState {
    MainMenu
```

```

MainMenu,
InGame,
Paused,
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // add the app state type
        .add_state(AppState::MainMenu)

        // add systems to run regardless of state, as usual
        .add_system(play_music)

        // systems to run only in the main menu
        .add_system_set(
            SystemSet::on_update(AppState::MainMenu)
                .with_system(handle_ui_buttons)
        )

        // setup when entering the state
        .add_system_set(
            SystemSet::on_enter(AppState::MainMenu)
                .with_system(setup_menu)
        )

        // cleanup when exiting the state
        .add_system_set(
            SystemSet::on_exit(AppState::MainMenu)
                .with_system(close_menu)
        )
        .run();
}

```

It is OK to have multiple system sets for the same state.

This is useful when you want to place [labels](#) and use [explicit system ordering](#).

This can also be useful with [Plugins](#). Each plugin can add its own set of systems to the same state.

States are implemented using [run criteria](#) under the hood. These special system set constructors are really just helpers to automatically add the state management run criteria.

Controlling States

Inside of systems, you can check and control the state using the `State<T>` resource:

```

fn play_music(
    app_state: Res<State<AppState>>,
    //

```

```
// ...
) {
    match app_state.current() {
        AppState::MainMenu => {
            // TODO: play menu music
        }
        AppState::InGame => {
            // TODO: play game music
        }
        AppState::Paused => {
            // TODO: play pause screen music
        }
    }
}
```

To change to another state:

```
fn enter_game(mut app_state: ResMut<State<AppState>>) {
    app_state.set(AppState::InGame).unwrap();
    // ^ this can fail if we are already in the target state
    // or if another state change is already queued
}
```

After the systems of the current state complete, Bevy will transition to the next state you set.

You can do arbitrarily many state transitions in a single frame update. Bevy will handle all of them and execute all the relevant systems (before moving on to the next [stage](#)).

State Stack

Instead of completely transitioning from one state to another, you can also overlay states, forming a stack.

This is how you can implement things like a "game paused" screen, or an overlay menu, with the game world still visible / running in the background.

You can have some systems that are still running even when the state is "inactive" (that is, in the background, with other states running on top). You can also add one-shot systems to run when "pausing" or "resuming" the state.

In your [app builder](#):

```
// player movement only when actively playing
.add_system_set(
    SystemSet::on_update(AppState::InGame)
```

```

        SystemSet::on_update(AppState::InGame)
            .with_system(player_movement)
    )
    // player idle animation while paused
    .add_system_set(
        SystemSet::on_inactive_update(AppState::InGame)
            .with_system(player_idle)
    )
    // animations both while paused and while active
    .add_system_set(
        SystemSet::on_in_stack_update(AppState::InGame)
            .with_system(animate_trees)
            .with_system(animate_water)
    )
    // things to do when becoming inactive
    .add_system_set(
        SystemSet::on_pause(AppState::InGame)
            .with_system(hide_enemies)
    )
    // things to do when becoming active again
    .add_system_set(
        SystemSet::on_resume(AppState::InGame)
            .with_system(reset_player)
    )
    // setup when first entering the game
    .add_system_set(
        SystemSet::on_enter(AppState::InGame)
            .with_system(setup_player)
            .with_system(setup_map)
    )
    // cleanup when finally exiting the game
    .add_system_set(
        SystemSet::on_exit(AppState::InGame)
            .with_system(despawn_player)
            .with_system(despawn_map)
    )
)

```

To manage states like this, use `push / pop`:

```

// to go into the pause screen
app_state.push(AppState::Paused).unwrap();
// to go back into the game
app_state.pop().unwrap();

```

(using `.set` as shown before replaces the active state at the top of the stack)

Known Pitfalls and Limitations

Combining with Other Run Criteria

Because states are implemented using [run criteria](#), they cannot be combined with other uses of run criteria, such as [fixed timestep](#).

If you try to add another run criteria to your system set, it would replace Bevy's state-management run criteria! This would make the system set no longer constrained to run as part of a state!

Consider using `iyes_loopless`, which does not have such limitations.

Multiple Stages

Bevy states cannot work across multiple `stages`. Workarounds are available, but they are broken and buggy.

This is a huge limitation in practice, as it greatly limits how you can use `commands`. Not being able to use Commands is a big deal, as you cannot do things like spawn entities and operate on them during the same frame, among other important use cases.

Consider using `iyes_loopless`, which does not have such limitations.

With Input

If you want to use `Input<T>` to trigger state transitions using a button/key press, you need to clear the input manually by calling `.reset`:

```
fn esc_to_menu(
    mut keys: ResMut<Input<KeyCode>>,
    mut app_state: ResMut<State<AppState>>,
) {
    if keys.just_pressed(KeyCode::Escape) {
        app_state.set(AppState::MainMenu).unwrap();
        keys.reset(KeyCode::Escape);
    }
}
```

(note that this requires `ResMut`)

Not doing this can cause `issues`.

`iyes_loopless` does not have this issue.

Events

When receiving `events` in systems that don't run all the time, such as during a pause state, you will miss any events that are sent during the frames when the receiving systems are not running!

To mitigate this, you could implement a `custom cleanup strategy`, to manually manage the lifetime of the relevant event types.

Run Criteria

Consider using the `iyesh_loopless` crate, which provides an alternative implementation that does not suffer from the `usability issues` of the one in Bevy.

Run Criteria are a mechanism for controlling if Bevy should run specific `systems`, at runtime. This is how you can make functionality that only runs under certain conditions.

Run Criteria can be applied to individual `systems`, `system sets`, and `stages`.

Run Criteria are Bevy systems that return a value of type `enum ShouldRun`. They can accept any `system parameters`, like a normal system.

This example shows how run criteria might be used to implement different multiplayer modes:

```
use bevy::ecs::schedule::ShouldRun;
```

```
#Example: (Placeholder for actual code)
```



```

#[derive(Debug, PartialEq, Eq)]
enum MultiplayerKind {
    Client,
    Host,
    Local,
}

fn run_if_connected(
    mode: Res<MultiplayerKind>,
    session: Res<MyNetworkSession>,
) -> ShouldRun
{
    if *mode == MultiplayerKind::Client && session.is_connected() {
        ShouldRun::Yes
    } else {
        ShouldRun::No
    }
}

fn run_if_host(
    mode: Res<MultiplayerKind>,
) -> ShouldRun
{
    if *mode == MultiplayerKind::Host || *mode == MultiplayerKind::Local {
        ShouldRun::Yes
    } else {
        ShouldRun::No
    }
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // if we are currently connected to a server,
        // activate our client systems
        .add_system_set(
            SystemSet::new()
                .with_run_criteria(run_if_connected)
                .before("input")
                .with_system(server_session)
                .with_system(fetch_server_updates)
        )

        // if we are hosting the game,
        // activate our game hosting systems
        .add_system_set(
            SystemSet::new()
                .with_run_criteria(run_if_host)
                .before("input")
                .with_system(host_session)
                .with_system(host_player_movement)
                .with_system(host_enemy_ai)
        )

        // other systems in our game
        .add_system(smoke_particles)
        .add_system(water_animation)
        .add_system_set(
            SystemSet::new()
                .label("input")
                .with_system(keyboard_input)

```

```
        .with_system(gamepad_input),
    )
    .run();
}
```

Known Pitfalls

Combining Multiple Run Criteria

It is not possible to make a system that is conditional on multiple run criteria. Bevy has a `.pipe` method that allows you to "chain" run criteria, which could let you modify the output of a run criteria, but this is very limiting in practice.

Consider using `!yes_loopless`. It allows you to use any number of run conditions to control your systems, and does not prevent you from using `states` or `fixed timestep`.

Events

When receiving `events` in systems that don't run every frame, you will miss any events that are sent during the frames when the receiving systems are not running!

To mitigate this, you could implement a `custom cleanup strategy`, to manually manage the lifetime of the relevant event types.

Labels

You need labels to name various things in your `app`, such as `systems` (for `order control`), `run criteria`, `stages`, and ambiguity sets.

Bevy uses some clever Rust type system magic, to allow you to use strings as well as your own custom types for labels, and even mix them!

Using strings for labels is quick and easy for prototyping. However, they are easy to mistype and are unstructured. The compiler cannot validate them for you, to catch mistakes.

You can also use custom types (usually `enum`s) to define your labels. This allows the compiler to check them, and helps you stay organized in larger projects.

You need to derive the appropriate trait, depending on what they will be used for: `StageLabel`, `SystemLabel`, `RunCriteriaLabel`, or `AmbiguitySetLabel`.

Any Rust type is suitable, as long as it has the following standard Rust traits: `Clone` + `PartialEq` + `Eq` + `Hash` + `Debug` (and the implied `Send`

- `Sync` + `'static`).

```
#[derive(Debug, Clone, PartialEq, Eq, Hash)]
#[derive(SystemLabel)]
```

```

enum MySystems {
    InputSet,
    Movement,
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
#[derive(StageLabel)]
enum MyStages {
    Prepare,
    Cleanup,
}

#[derive(Debug, Clone, PartialEq, Eq, Hash)]
#[derive(StageLabel)]
struct DebugStage;

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // Add our game systems:
        .add_system_set(
            SystemSet::new()
                .label(MySystems::InputSet)
                .with_system(keyboard_input)
                .with_system(gamepad_input)
            )
        .add_system(player_movement.label(MySystems::Movement))

        // temporary debug system, let's just use a string label
        .add_system(debug_movement.label("temp-debug"))

        // Add our custom stages:
        // note that Bevy's `CoreStage` is an enum just like ours!
        .add_stage_before(CoreStage::Update, MyStages::Prepare,
SystemStage::parallel())
        .add_stage_after(CoreStage::Update, MyStages::Cleanup, SystemStage::parallel())

        .add_stage_after(CoreStage::Update, DebugStage, SystemStage::parallel())

        // we can just use a string for this one:
        .add_stage_before(CoreStage::PostUpdate, "temp-debug-hack",
SystemStage::parallel())

        .run();
}

```

For quick prototyping, it is convenient to just use strings as labels.

However, by defining your labels as custom types, the Rust compiler can check them for you, and your IDE can auto-complete them. It is the recommended way, as it prevents mistakes, and helps you stay organized in larger projects.

Stages

All [systems](#) to be run by Bevy are contained in stages. Every frame update, Bevy executes each stage, in order. Within each stage, Bevy's scheduling algorithm can run many systems in parallel, using multiple CPU cores for good performance.

The boundaries between stages are effectively hard synchronization points. They ensure that all systems of the previous stage have completed before any systems of the next stage begin, and that there is a moment in time when no systems are in-progress.

This makes it possible/safe to apply [Commands](#). Any operations performed by systems using [Commands](#) are applied at the end of that stage.

Internally, Bevy has at least these built-in [stages](#):

- In the [main app](#) ([CoreStage](#)): [First](#), [PreUpdate](#), [Update](#), [PostUpdate](#), [Last](#)
- In the render [sub-app](#) ([RenderStage](#)): [Extract](#), [Prepare](#), [Queue](#), [PhaseSort](#), [Render](#), [Cleanup](#)

By default, when you add your systems, they are added to [CoreStage::Update](#).

Bevy's internal systems are in the other stages, to ensure they are ordered correctly relative to your game logic.

If you want to add your own systems to any of Bevy's internal stages, you need to beware of potential unexpected interactions with Bevy's own internal systems. Remember: Bevy's internals are implemented using ordinary systems and ECS, just like your own stuff!

You can add your own additional stages. For example, if we want our debug systems to run after our game logic:

```
fn main() {
    // label for our debug stage
    static DEBUG: &str = "debug";

    App::new()
        .add_plugins(DefaultPlugins)

        // add DEBUG stage after Bevy's Update
        // also make it single-threaded
        .add_stage_after(CoreStage::Update, DEBUG, SystemStage::single_threaded())

        // systems are added to the `CoreStage::Update` stage by default
        .add_system(player_gather_xp)
        .add_system(player_take_damage)

        // add our debug systems
        .add_system_to_stage(DEBUG, debug_player_hp)
        .add_system_to_stage(DEBUG, debug_stats_change)
        .add_system_to_stage(DEBUG, debug_new_hostiles)

        .run();
}
```

If you need to manage when your systems run, relative to one another, it is generally preferable to avoid using stages, and to use [explicit system ordering](#) instead. Stages limit parallel execution and

the performance of your game.

However, stages can make it easier to organize things, when you really want to be sure that all previous systems have completed. Stages are also the only way to apply [Commands](#).

If you have systems that need to rely on the actions that other systems have performed by using [Commands](#), and need to do so during the same frame, placing those systems into separate stages is the only way to accomplish that.

Removal Detection

Relevant official examples: `removal_detection`.

Removal detection is special. This is because, unlike with `change detection`, the data does not exist in the ECS anymore (obviously), so Bevy cannot keep tracking metadata for it.

Nevertheless, being able to respond to removals is important for some applications, so Bevy offers a limited form of it.

Components

You can check for `components` that have been removed during the current frame. The data is cleared at the end of every frame update. Note that this makes this feature tricky to use, and requires you to use multiple `stages`.

When you remove a component (using `Commands (Commands)`), the operation is applied at the end of the `stage`. The `system` that checks for the removal must run in a later stage during the same frame update. Otherwise, it will not detect the removal.

Use the `RemovedComponents<T>` special system parameter type, to get an iterator for the `Entity` IDs of all the entities that had a component of type `T` that was removed earlier this frame.

```
/// Some component type for the sake of this example.  
#[derive(Component)]  
struct Foo;
```

```
struct Seen;
```

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        // we could add our system to Bevy's `PreUpdate` stage
        // (alternatively, you could create your own stage)
        .add_system_to_stage(CoreStage::PreUpdate, remove_components)
        // our detection system runs in a later stage
        // (in this case: Bevy's default `Update` stage)
        .add_system(detect_removals)
        .run();
}

fn remove_components(
    mut commands: Commands,
    q: Query<(Entity, &Transform), With<Seen>>,
) {
    for (e, transform) in q.iter() {
        if transform.translation.y < -10.0 {
            // remove the `Seen` component from the entity
            commands.entity(e)
                .remove::();
        }
    }
}

fn detect_removals(
    removals: RemovedComponents<Seen>,
    // ... (maybe Commands or a Query ?) ...
) {
    for entity in removals.iter() {
        // do something with the entity
    }
}
```

(To do things with these entities, you can just use the `Entity` IDs with `Commands::entity()` or `Query::get()`.)

Resources

Bevy does not provide any API for detecting when [resources](#) are removed.

You can work around this using `Option` and a separate `Local` system parameter, effectively implementing your own detection.

```
fn detect_removed_res(
    my_res: Option<Res<MyResource>>,
    mut my_res: Local<MyResource>
) {
    if my_res.is_none() {
        my_res = Res::new(MyResource::new());
    }
}
```



```

mut my_res_existed: Local<bool>,
) {
  if let Some(my_res) = my_res {
    // the resource exists!

    // remember that!
    *my_res_existed = true;

    // (... you can do something with the resource here if you want ...)
  } else if *my_res_existed {
    // the resource does not exist, but we remember it existed!
    // (it was removed)

    // forget about it!
    *my_res_existed = false;

    // ... do something now that it is gone ...
  }
}

```

Note that, since this detection is local to your system, it does not have to happen during the same frame update.

Param Sets

For safety reasons, a [system](#) cannot have multiple parameters whose data access might have a chance of mutability conflicts over the same data.

Some examples:

- Multiple incompatible [queries](#).
- Using `&World` while also having other system parameters to access specific data.
- ...

Bevy provides a solution: wrap them in a `ParamSet`:

```
fn reset_health(
    // access the health of enemies and the health of players
    // (note: some entities could be both!)
    mut set: ParamSet<(
        Query<&mut Health, With<Enemy>>,
        Query<&mut Health, With<Player>>,
        // also access the whole world ... why not
        &World,
    )>,
) {
    // set health of enemies (use the 1st param in the set)
    for mut health in set.p0().iter_mut() {
        health.hp = 50.0;
    }

    // set health of players (use the 2nd param in the set)
    for mut health in set.p1().iter_mut() {
        health.hp = 100.0;
    }

    // read some data from the world (use the 3rd param in the set)
    let my_resource = set.p2().resource::();

    // since we only used the conflicting system params one at a time,
    // everything is safe and our code can compile; ParamSet guarantees this
}
```

This ensures only one of the conflicting parameters can be used at the same time.

The maximum number of parameters in a param set is 8.

System Chaining

Relevant official examples: [system_chaining](#).

You can compose a single Bevy [system](#) from multiple Rust functions.

You can make functions that can take an input and produce an output, and be connected together to run as a single larger system.

This is called "system chaining", but beware that the term is somewhat misleading – you are *not* creating a chain of multiple systems to run in order; you are creating a single large Bevy system consisting of multiple Rust functions.

Note that system chaining is *not* a way of communicating between systems. If you want to pass data between systems, you should use [Events](#) instead.

One useful application is to be able to return errors from your system code (allowing the use of Rust's `?` operator) and then have a separate function for handling them:

```
fn net_receive(mut netcode: ResMut<MyNetProto>) -> std::io::Result<()> {
    netcode.receive_updates()?;

    Ok(())
}

fn handle_io_errors(In(result): In<std::io::Result<()>>) {
    if let Err(e) = result {
        eprintln!("I/O error occurred: {}", e);
    }
}
```

Such functions cannot be [registered](#) individually as systems (Bevy doesn't know what to do with the input/output). You have to connect them in a chain:

```
fn main() {
    App::new()
        // ...
        .add_system(net_receive.chain(handle_io_errors))
        // ...
        .run();
}
```

Performance Warning

Beware that Bevy treats the whole chain as if it was a single big system, with all the combined resources and queries. This implies that parallelism could be limited, affecting performance.

Avoid adding a system that requires mutable access to anything, as part of multiple chains. It would block all affected chains (and other systems accessing the same data) from running in parallel.

Direct World Access

The `World` is where Bevy ECS stores all data and associated metadata. It keeps track of `resources`, `entities` and `components`.

Typically, the `App`'s schedule runner will run all `stages` (which, in turn, run their `systems`) on the main world. Regular `systems` are limited in what data they can access from the world, by their `system parameter types`. Operations that manipulate the world itself are only done indirectly using `Commands`. This is how most typical Bevy user code behaves.

However, there are also ways you can get full direct access to the world, which gives you full control and freedom to do anything with any data stored in the Bevy ECS:

- `Exclusive systems`
- `FromWorld` impls
- Via the `App` builder
- Manually created `World`s for purposes like `tests` or scenes
- Custom `Commands`
- Custom `Stage` impls (not recommended, prefer exclusive systems)

Direct world access lets you do things like:

- Freely spawn/despawn entities, insert/remove resources, etc., taking effect immediately (no delay like when using `[Commands]` from a regular `system`)
- Access any component, entities, and resources you want
- Manually run systems or stages

Working with the `World`

Here are some ways that you can make use of the direct world access APIs.

`SystemState`

The easiest way to do things is using a `SystemState`.

This is a type that "imitates a system", behaving the same way as a `system` with various parameters would. All the same behaviors like `queries`, `change detection`, and even `Commands` are available. You can use any `system params`.

It also tracks any persistent state, used for things like `change detection` or caching to improve performance. Therefore, if you plan on reusing the same `SystemState` multiple times, you should store it somewhere, rather than creating a new one every time. Every time you call `.get(world)`, it behaves like another "run" of a system.

If you are using `Commands`, you can choose when you want to apply them to the world. You need to manually call `.apply(world)` on the `SystemState`, to apply them.

```
// TODO: write code example
```

Running a Stage

If you want to run some systems (a common use-case is [testing](#)), the easiest way is to construct an impromptu `SystemStage` ([stages](#)). This way you reuse all the scheduling logic that Bevy normally does when running systems.

```
// TODO: write code example
```

Navigating by Metadata

The world contains a lot of metadata that allows navigating all the data efficiently, such as information about all the stored components, entities, archetypes.

```
// TODO: write code example
```

Exclusive Systems

Exclusive systems are [systems](#) that Bevy will not run in parallel with any other system. They can have full unrestricted access to the whole ECS `World`, by taking a `&mut World` parameter.

Inside of an exclusive system, you have full control over all data stored in the ECS. You can do whatever you want.

Note that exclusive systems can limit performance, as they prevent multi-threading (nothing else runs at the same time).

Some example situations where exclusive systems are useful:

- Dump various entities and components to a file, to implement things like saving and loading of game save files, or scene export from an editor
- Directly spawn/despawn [entities](#), or create/remove [resources](#), immediately with no delay (unlike when using [Commands](#) from a regular system)
- Run arbitrary systems with your own scheduling algorithm
- ...

See the [direct World access page](#) to learn more about how to do such things.

```
fn do_crazy_things(world: &mut World) {  
    // we can do anything with any data in the Bevy ECS here!  
}
```

You need to add exclusive systems to the [App](#), just like regular systems, but you must call `.exclusive_system()` on them.

They cannot be ordered in-between regular parallel systems. Exclusive systems always run at one of the following places:

- `.at_start()` : at the beginning of a [stage](#)
- `.at_end()` : at the end of a [stage](#), after [commands](#) from regular systems have been applied
- `.before_commands()` : after all the regular systems in a [stage](#), but before [commands](#) are applied

(if you don't specify anything, the default is assumed `.at_start()`)

```
fn main() {  
    App::new()  
        .add_plugins(DefaultPlugins)
```

```
.add_plugins(DefaultPlugins)

// this will run at the start of CoreStage::Update (the default stage)
.add_system(do_crazy_things.exclusive_system())

// this will run at the end of CoreStage::PostUpdate
.add_system_to_stage(
    CoreStage::PostUpdate,
    some_more_things
    .exclusive_system()
    .at_end()
)

.run();
}
```

This page has not been written yet...

"Non-send" refers to data types that must only be accessed from the "main thread" of the application. Such data is marked by Rust as `!Send` (lacking the `Send` trait).

Some (often system) libraries have interfaces that cannot be safely used from other threads. A common example of this are various low-level OS interfaces for things like windowing, graphics, or audio. If you are doing advanced things like creating a Bevy plugin for interfacing with such things, you may encounter the need for this.

Normally, Bevy works by running all your `systems` on a thread-pool, making use of many CPU cores. However, you might need to ensure that some code always runs on the "main thread", or access data that is not safe to access in a multithreaded way.

Non-Send Systems and Data Access

To do this, you can use a `NonSend<T>` / `NonSendMut<T>` system parameter. This behaves just like `Res<T>` / `ResMut<T>`, letting you access an ECS `resource` (single global instance of some data), except that the presence of such a parameter forces the Bevy scheduler to always run the `system` on the main thread. This ensures that data never has to be sent between threads or accessed from different threads.

One example of such a resource is `WinitWindows` in Bevy. This is the low-level version of `Windows` that gives you more direct access to OS window management functionality.

```
fn setup_raw_window(mut windows: NonSend<WinitWindows>) {
    let raw_window = windows.get_window(WindowId::primary()).unwrap();
    // do some special things
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        // just add it as a normal system;
        // Bevy will notice the NonSend parameter
        // and ensure it runs on the main thread
        .add_startup_system(setup_raw_window)
        .run();
}
```

Custom Non-Send Resources

Normally, to insert `resources`, their types must be `Send`.

Bevy tracks non-Send resources separately, to ensure that they can only be accessed using `NonSend<T>` / `NonSendMut<T>`.

It is not possible to insert non-send resources using `Commands`, only using `direct World access`. This means that you have to initialize them in an `exclusive system`, `FromWorld` impl, or custom stage.

```
fn setup_platform_audio(world: &mut World) {  
    // assuming `OSAudioMagic` is some primitive that is not thread-safe  
    let instance = OSAudioMagic::init();  
  
    world.insert_non_send_resource(instance);  
}  
  
fn main() {  
    App::new()  
        .add_plugins(DefaultPlugins)  
        .add_startup_system(setup_platform_audio.exclusive_system())  
        .run();  
}
```

Writing Tests for Systems

You might want to write and run automated tests for your [systems](#).

You can use the regular Rust testing features (`cargo test`) with Bevy.

To do this, you can create an empty ECS `World` in your tests, and then, using [direct World access](#), insert whatever [entities](#) and [resources](#) you need for testing. Create a standalone [stage](#) with the [systems](#) you want to run, and manually run it on the `World` .

Bevy's official repository has a fantastic [example of how to do this](#).

This chapter is about any non-obvious tricks, programming techniques, patterns and idioms, that may be useful when programming with Bevy.

These topics are an extension of the topics covered in the [Bevy Programming Framework](#) chapter. See that chapter to learn the foundational concepts.

Bevy [systems](#) are just plain rust functions, which means they can be generic. You can add the same system multiple times, parametrized to work on different Rust types or values.

Generic over Component types

You can use the generic type parameter to specify what [component](#) types (and hence what [entities](#)) your [system](#) should operate on.

This can be useful when combined with Bevy [states](#). You can do the same thing to different sets of entities depending on state.

Example: Cleanup

One straightforward use-case is for cleanup. We can make a generic cleanup system that just despawns all entities that have a certain component type. Then, trivially run it on exiting different states.

```
use bevy::ecs::component::Component;

fn cleanup_system<T: Component>(
    mut commands: Commands,
    q: Query<Entity, With<T>>,
) {
    for e in q.iter() {
        commands.entity(e).despawn_recursive();
    }
}
```

Menu entities can be tagged with `cleanup::MenuExit`, entities from the game map can be tagged with `cleanup::LevelUnload`.

We can add the generic cleanup system to our state transitions, to take care of the respective entities:

```
/// Marker components to group entities for cleanup
mod cleanup {
```

```

use bevy::prelude::*;
#[derive(Component)]
pub struct LevelUnload;
#[derive(Component)]
pub struct MenuClose;
}

#[derive(Debug, Clone, Eq, PartialEq, Hash)]
enum AppState {
    MainMenu,
    InGame,
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_state(AppState::MainMenu)
        // add the cleanup systems
        .add_system_set(SystemSet::on_exit(AppState::MainMenu)
            .with_system(cleanup_system::<cleanup::MenuClose>))
        .add_system_set(SystemSet::on_exit(AppState::InGame)
            .with_system(cleanup_system::<cleanup::LevelUnload>))
        .run();
}

```

Using Traits

You can use this in combination with Traits, for when you need some sort of varying implementation/functionality for each type.

Example: Bevy's Camera Projections

(this is a use-case within Bevy itself)

Bevy has a `CameraProjection` trait. Different projection types like `PerspectiveProjection` and `OrthographicProjection` implement that trait, providing the correct logic for how to respond to resizing the window, calculating the projection matrix, etc.

There is a generic system `fn camera_system::<T: CameraProjection + Component>`, which handles all the cameras with a given projection type. It will call the trait methods when appropriate (like on window resize events).

The [Bevy Cookbook Custom Camera Projection Example](#) shows this API in action.

Using Const Generics

Now that Rust has support for Const Generics, functions can also be parametrized by values, not just types.

```
fn process_layer<const LAYER_ID: usize>(
    // system params
) {
    // do something for this `LAYER_ID`
}

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_system(process_layer::<1>)
        .add_system(process_layer::<2>)
        .add_system(process_layer::<3>)
        .run();
}
```

Note that these values are static / constant at compile-time. This can be a severe limitation. In some cases, when you might suspect that you could use const generics, you might realize that you actually want a runtime value.

If you need to "configure" your system by passing in some data, you could, instead, use a [Resource](#) or [Local](#).

Note: As of Rust 1.59, support for using `enum` values as const generics is not yet stable. To use `enum`s, you need Rust Nightly, and to enable the experimental/unstable feature (put this at the top of your `main.rs` or `lib.rs`):

```
#![feature(adt_const_params)]
```

Component Storage (Table/Sparse-Set)

Bevy ECS provides two different ways of storing data: tables and sparse sets. The two storage kinds offer different performance characteristics.

The kind of storage to be used can be chosen per [component](#) type. When you derive the `Component` trait, you can specify it. The default, if unspecified, is table storage.

You can have components with a mixture of different storage kinds on the same entity.

The rest of this page is dedicated to explaining the performance trade-offs and why you might want to choose one storage kind vs. the other.

```
/// Component for entities that can cast magic spells
#[derive(Component)] // Use the default table storage
struct Mana {
    mana: f32,
}

/// Component for enemies that currently "see" the player
/// Every frame, add/remove to entities based on visibility
/// (use sparse-set storage due to frequent add/remove)
#[derive(Component)]
#[component(storage = "SparseSet")]
struct CanSeePlayer;

/// Component for entities that are currently taking bleed damage
/// Add to entities to apply bleed effect, remove when done
/// (use sparse-set storage to not fragment tables,
/// as this is a "temporary effect")
#[derive(Component)]
#[component(storage = "SparseSet")]
struct Bleeding {
    damage_rate: f32,
}
```

Table Storage

Table storage is optimized for fast [query](#) iteration. If the way you usually use a specific component type is to access its data across many entities, this will offer the best performance.

However, adding/removing table components to existing entities is a relatively slow operation. It requires copying the data of all the other table components for the entity, to a different location in memory.

It's OK if you have to do this sometimes, but if you are likely to add/remove a component very frequently, you might want to switch that component type to sparse-set storage.

You can see why table storage was chosen as Bevy's default. Most component types are rarely added/removed in practice. You typically spawn entities with all the components they should have, and then access the data via queries, usually every frame. Sometimes you might add or remove a component to change an entity's behavior, but probably not nearly as often, or every frame.

Sparse-Set Storage

Sparse-Set storage is optimized for fast adding/removing of a component to existing entities, at the cost of slower querying. It can be more efficient for components that you would like to add/remove very frequently.

An example of this might be a [marker component](#) indicating whether an enemy can currently see the player. You might want to have such a component type, so that you can easily use a [query filter](#) to find all the enemies that are currently tracking the player. However, this is something that can change every frame, as enemies or the player move around the game level. If you add/remove this component every time the visibility status changed, that's a lot of additions and removals.

You can see that situations like these are more niche and do not apply to most component types. Treat sparse-set storage as a potential optimization you could try in specific circumstances.

Table Fragmentation

Furthermore, the actual memory layout of the "tables" depends on the set of all table components that each of your entities has.

ECS queries perform best when many of the entities they match have the same overall set of components.

Having a large number of entities, that all have the same component types, is very efficient in terms of data access performance. Having diverse entities with a varied mixture of different component types, means that their data will be fragmented in memory and be less efficient to access.

Sparse-Set components do not affect the memory layout of tables. Hence, components that are only used on a few entities or as a "temporary effect", might also be good candidates for sparse-set storage. That way they don't fragment the memory of the other (table) components.

Overall Advice

While this page describes the general performance characteristics and gives some guidelines, you often cannot know if something improves performance without benchmarking.

You could use sparse-set storage just in obvious situations like the "visibility marker" example given earlier, and otherwise leave the default table storage.

When your game grows complex enough and you have something to benchmark, you could try to apply it in more places and see how it affects your results.

Manual Event Clearing

[Click here to download a full example file with the code from this page.](#)

The [event](#) queue needs to be cleared periodically, so that it does not grow indefinitely and waste unbounded memory.

Bevy's default cleanup strategy is to clear events every frame, but with double buffering, so that events from the previous frame update stay available. This means that you can handle the events only until the end of the next frame after the one when they are sent.

This default works well for systems that run every frame and check for events every time, which is the typical usage pattern.

However, if you have systems that do not read events every frame, they might miss some events. Some common scenarios where this occurs are:

- systems with an early-return, that don't read events every time they run
- when using [fixed timestep](#)
- systems that only run in specific [states](#), such as if your game has a pause state
- when using custom [run criteria](#) to control your systems

To be able to reliably manage events in such circumstances, you might want to have manual control over how long the events are held in memory.

You can replace Bevy's default cleanup strategy with your own.

To do this, simply add your event type (wrapped as `Events<T>`) to the [app builder](#) using `.init_resource`, instead of `.add_event`.

(`.add_event` is actually just a convenience method that initializes the [resource](#) and adds Bevy's built-in system ([generic](#) over your event type) for the default cleanup strategy)

You must then clear the events at your discretion. If you don't do this often enough, your events might pile up and waste memory.

Example

We can create [generic systems](#) for this. Implement the custom cleanup strategy, and then add that [system](#) to your `App` as many times as you need, for each [event](#) type where you want to use your custom behavior.

```
use bevy::ecs::event::Events;
```

```
fn main() {
```

```

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)

        // add the `Events<T>` resource manually
        // these events will not have automatic cleanup
        .init_resource::

```

Bevy Render (GPU) Framework

NOTE: This chapter of the book is an early *Work in Progress*! Many links are still broken!

This chapter covers Bevy's rendering framework and how to work with the GPU.

Make sure you are well familiar with [Bevy's Core Programming Framework](#). Everything here builds on top of it.

Here you will learn how to write custom rendering code. If you are simply interested in using the existing graphical features provided by Bevy, check out the chapters about [2D](#) and [3D](#).

Render Architecture Overview

NOTE: This chapter of the book is an early *Work in Progress*! Many links are still broken!

The current Bevy render architecture premiered in Bevy 0.6. The [news blog post](#) is another place you can learn about it. :)

It was inspired by the Destiny Render Architecture (from the Destiny game).

Pipelined Rendering

Bevy's renderer is architected in a way that operates independently from all the normal app logic. It operates in its own separate [ECS World](#) and has its own [schedule](#), with [stages](#) and [systems](#).

The plan is that, in a future Bevy version, the renderer will run in parallel with all the normal app logic, allowing for greater performance. This is called "pipelined rendering": rendering the previous frame at the same time as the app is processing the next frame update.

Every frame, the two parts are synchronized in a special [stage](#) called "Extract". The Extract stage has access to both [ECS Worlds](#), allowing it to copy data from the main World into the render World.

From then on, the renderer only has access to the render World, and can only use data that is stored there.

Every frame, all [entities](#) in the render World are erased, but [resources](#) are kept. If you need to persist data from frame to frame, store it in resources. Dynamic data that could change every frame should be copied into the render world in the Extract stage, and typically stored using entities and components.

 Diagram of pipelined rendering timings in app-bound and render-bound cases

Core Architecture

The renderer operates in multiple [render stages](#). This is how the work that needs to be performed on the CPU is managed.

- **Extract** : quickly copy the minimal data you need from the main World to the render World
- **Prepare** : send data to the GPU (buffers, textures, bind groups)
- **Queue** : generate the render jobs to be run (usually [phase items](#))
- **PhaseSort** : sort and batch [phase items](#) for efficient rendering
- **Render** : execute the [render graph](#) to produce actual GPU commands and do the work
- **Cleanup** : clear any data from the render World that should not persist to the next frame

The ordering of the workloads to be performed on the GPU is controlled using the [render graph](#). The graph consists of [nodes](#), each representing a workload for the GPU, typically a [render pass](#). The nodes are connected using [edges](#), representing their ordering/dependencies with regard to one another.

Layers of Abstraction

The Bevy rendering framework can accomodate you working at various different levels of abstraction, depending on how much you want to integrate with the Bevy ecosystem and built-in features, vs. have more direct control over the GPU.

For most things, you would be best served by the "high-level" or "mid-level" APIs.

Low-Level

Bevy works directly with `wgpu`, a Rust-based cross-platform graphics API. It is the abstraction layer over the GPU APIs of the underlying `platform`. This way, the same GPU code can work on all supported platforms. The API design of `wgpu` is based on the WebGPU standard, but with extensions to support native platform features, going beyond the limitations of the web platform.

`wgpu` (and hence Bevy) supports the following backends for each platform:

- Vulkan (Linux/Windows/Android)
- DirectX 12 (Windows)
- Metal (Apple)
- WebGL2 (Web)
- WebGPU (Web; experimental)
- GLES3 (Linux/Android; legacy)
- DirectX 11 (Windows; legacy; WIP (not yet ready for use))

`wgpu` forms the "lowest level" of Bevy rendering. If you really need the most direct control over the GPU, you can pretty much use `wgpu` directly, from within the Bevy render framework.

Mid-Level

On top of `wgpu`, Bevy provides some abstractions that can help you, and integrate better with the rest of Bevy.

The first is `pipeline caching` and `specialization`. If you create your `render pipelines` via this interface, Bevy can manage them efficiently for you, creating them when they are first used, and then caching and reusing them, for optimal performance.

Caching and specialization are, analogously, also available for `GPU Compute pipelines`.

Similar to the `pipeline cache`, there is a `texture cache`. This is what you use for rendering-internal `textures` (for example: shadow maps, reflection maps, ...), that do not originate from `assets`. It will manage and reuse the GPU memory allocation, and free it when it becomes unused.

For using data from `assets`, Bevy provides the `Render Asset` abstraction to help with extracting the data from different `asset types`.

Bevy can manage all the "objects to draw" using [phases](#), which sort and draw [phase items](#). This way, Bevy can sort each object to render, relative to everything else in the scene, for optimal performance and correct transparency (if any).

Phase Items are defined using [render commands](#) and/or [draw functions](#). These are, conceptually, the rendering equivalents of ECS [systems](#) and [exclusive systems](#), fetching data from the ECS World and generating [draw calls](#) for the GPU.

All of these things fit into the core architecture of the Bevy [render graph](#) and [render stages](#). During the Render stage, [graph nodes](#) will execute [render passes](#) with the [render phases](#), to draw everything as it was set up in the Prepare/Queue/PhaseSort stages.

The `bevy_core_pipeline` crate defines a set of [standard phase/item](#) and main pass types. If you can, you should work with them, for best compatibility with the Bevy ecosystem.

High-Level

On top of all the mid-level APIs, Bevy provides abstractions to make many common kinds of workloads easier.

The most notable higher-level features are [meshes](#) and [materials](#).

Meshes are the source of per-vertex data ([vertex attributes](#)) to be fed into your [shaders](#). The material specifies what [shaders](#) to use and any other data that needs to be fed into it, like [textures](#).

Render Stages

Everything on the CPU side (the whole process of driving the GPU workloads) is structured in a sequence of "render stages":

- **Extract** : quickly copy the minimal data you need from the main World to the render World
- **Prepare** : send data to the GPU (buffers, textures, bind groups)
- **Queue** : generate the render jobs to be run (usually [phase items](#))
- **PhaseSort** : sort and batch [phase items](#) for efficient rendering
- **Render** : execute the [render graph](#) to produce actual GPU commands and do the work
- **Cleanup** : clear any data from the render World that should not persist to the next frame

Timings

Note: Pipelined rendering is not yet actually enabled in Bevy 0.7. This section explains the intended behavior, which will land in a future Bevy version. You have to understand it, because any custom rendering code you write will have to work with it in mind.

Diagram of pipelined rendering timings in app-bound and render-bound cases

Every frame, [Extract](#) serves as the synchronization point.

When the Render Schedule completes, it will start again, but Extract will wait for the App Schedule, if it has not completed yet. The App Schedule will start again as soon as Extract has completed.

Therefore:

- in an App-bound scenario (if app takes longer than render):
 - The start of Extract is waiting for App to finish
- in a Render-bound scenario (if render takes longer than app):
 - The start of App is waiting for Extract to finish

If [vsync](#) is enabled, the wait for the next refresh of the screen will happen in the [Prepare](#) stage. This has the effect of prolonging the Prepare stage in the Render schedule. Therefore, in practice, your game will behave like the "Render-bound" scenario shown above.

The final render (the framebuffer with the pixels to show in the [window](#)) is *presented* to the OS/driver at the end of the [Render](#) stage.

Currently Bevy updates its [timing information](#) (in [Res<Time>](#)) at the start of the First stage in the main App schedule. Note that this is not linked to rendering in any way. This is a [known issue](#).

Adding Systems to Render Stages

If you are implementing custom rendering functionality in Bevy, you will likely need to add some of your own systems to at least some of the render stages:

- Anything that needs data from your main App World will need a system in [Extract](#) to copy that data. In practice, this is almost everything, unless it is fully contained on the GPU / in VRAM.

- Most use cases will need to do some setup of GPU resources in [Prepare](#) and/or [Queue](#).
- In [Cleanup](#), all [entities](#) are cleared automatically. If you have some custom data stored in [resources](#), you can let it stay for the next frame, or add a system to clear it, if you want.

The way Bevy is set up, you shouldn't need to do anything in [Render](#) or [PhaseSort](#). If your custom rendering is part of the Bevy [render graph](#), it will just be handled automatically when Bevy executes the render graph in the [Render](#) stage. If you are implementing custom [phase items](#), the Main Pass render graph node will render them together with everything else.

You can add your rendering systems to the respective stages, using the render [sub-app](#):

```
// TODO: code example
```

Extract

Extract is a very important and special stage. It is the synchronization point that links the two ECS Worlds. This is where the data required for rendering is copied ("extracted") from the main App World into the Render World, allowing for pipelined rendering.

During the Extract stage, nothing else can run in parallel, on either the main App World or the Render World. Hence, Extract should be kept minimal and complete its work as quickly as possible.

It is recommended that you avoid doing any computations in Extract, if possible. Just copy data.

It is recommended that you only copy the data you actually need for rendering. Create new [component types](#) and [resources](#) just for use within the render World, with only the data you need.

For example, Bevy's 2D sprites uses a `struct ExtractedSprite`, where it copies the relevant data from the "user-facing" components of sprite and spritesheet entities in the main World.

Bevy **reserves Entity IDs** in the render World, matching all the Entities existing in the main World. In most cases, you do not need to *spawn* new entities in the render World. You can just [insert components with Commands](#) on the same Entity IDs as from the main World.

```
// TODO: code example
```

Prepare

Prepare is the stage to use if you need to set up any data on the GPU. This is where you can create GPU [buffers](#), [textures](#), and [bind groups](#).

// TODO: elaborate on different ways Bevy is using it internally

```
// TODO: code example
```

Queue

Queue is the stage where you can set up the "rendering jobs" you will need to execute.

Typically, this means creating [phase items](#) with the correct [render pipeline](#) and [draw function](#), for everything that you need to draw.

For other things, analogously, Queue is where you would set up the workloads (like compute or draw calls) that the GPU would need to perform.

// TODO: elaborate on different ways Bevy is using it internally

```
// TODO: code example
```

PhaseSort

This stage exists for Bevy to sort all of the [phase items](#) that were set up during the [Queue](#) stage, before rendering in the [Render](#) stage.

It is unlikely that you will need to add anything custom here. I'm not aware of use cases. [Let me know](#) if you know of any.

Render

Render is the stage where Bevy executes the [Render Graph](#).

If you are using any of the [standard render phases](#), you don't need to do anything. Your custom [phase items](#) will be rendered automatically as part of the Main Pass built-in render graph [nodes](#), alongside everything else.

If you are implementing a rendering feature that needs a separate step, you can add it as a [render graph node](#), and it will be rendered automatically.

The only time you might need to do something custom here is if you really want to sidestep Bevy's frameworks and reach for low-level `wgpu` access. You could place it in the Render stage.

Cleanup

Bevy has a built-in system in Cleanup that clears all [entities](#) in the render World. Therefore, all data stored in components will be lost. It is expected that fresh data will be obtained in the next frame's [Extract](#) stage.

To persist rendering data over multiple frames, you should store it in [resources](#). That way you have

To persist rendering data over multiple frames, you should store it in [resources](#). That way you have control over it.

If you need to clear some data from your resources sometimes, you could add a custom system to the Cleanup stage to do it.

```
// TODO: code example
```

This chapter shows you how to do various practical things using Bevy.

Indended as a supplement to Bevy's [official examples](#).

The examples are written to only focus on the relevant information for the task at hand.

Only the relevant parts of the code are shown. Full compilable example files are available and linked on each page.

It is assumed that you are already familiar with [Bevy Programming](#).

Show Framerate in Console

[Click here for the full example code.](#)

You can use bevy's builtin diagnostics system to print framerate (FPS) to the console, for monitoring performance.

```
use bevy::diagnostic::{FrameTimeDiagnosticsPlugin, LogDiagnosticsPlugin};

fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_plugin(LogDiagnosticsPlugin::default())
        .add_plugin(FrameTimeDiagnosticsPlugin::default())
        .run();
}
```

Convert cursor to world coordinates

[Click here for the full example code.](#)

Bevy does not yet provide built-in functions to help with finding out what the cursor is pointing at.

3D games

There is a good (unofficial) plugin: `bevy_mod_picking`.

2D games

This code should work for 2D games with orthographic projections. It is "undoing" the projection and camera transformations.

(there will likely be *slight* inaccuracy from floating-point calculations)

This should work regardless of the scaling settings of your projection, and camera [transform](#).

```
/// Used to help identify our main camera
#[derive(Component)]
struct MainCamera
```

```
struct MainCamera;
```

```
fn setup(mut commands: Commands) {
    commands.spawn()
        .insert_bundle(OrthographicCameraBundle::new_2d())
        .insert(MainCamera);
}

fn my_cursor_system(
    // need to get window dimensions
    wnds: Res<Windows>,
    // query to get camera transform
    q_camera: Query<(&Camera, &GlobalTransform), With<MainCamera>>
) {
    // get the camera info and transform
    // assuming there is exactly one main camera entity, so query::single() is OK
    let (camera, camera_transform) = q_camera.single();

    // get the window that the camera is displaying to (or the primary window)
    let wnd = if let RenderTarget::Window(id) = camera.target {
        wnds.get(id).unwrap()
    } else {
        wnds.get_primary().unwrap()
    };

    // check if the cursor is inside the window and get its position
    if let Some(screen_pos) = wnd.cursor_position() {
        // get the size of the window
        let window_size = Vec2::new(wnd.width() as f32, wnd.height() as f32);

        // convert screen position [0..resolution] to ndc [-1..1] (gpu coordinates)
        let ndc = (screen_pos / window_size) * 2.0 - Vec2::ONE;

        // matrix for undoing the projection and camera transform
        let ndc_to_world = camera_transform.compute_matrix() *
camera.projection_matrix.inverse();

        // use it to convert ndc to world-space coordinates
        let world_pos = ndc_to_world.project_point3(ndc.extend(-1.0));

        // reduce it to a 2D value
        let world_pos: Vec2 = world_pos.truncate();

        eprintln!("World coords: {}/{}", world_pos.x, world_pos.y);
    }
}
```

Custom Camera Projection

[Click here for the full example code.](#)

Camera with a custom projection (not using one of Bevy's standard perspective or orthographic projections).

You could also use this to change the coordinate system, if you insist on using something other than [Bevy's default coordinate system](#), for whatever reason.

Here we implement a simple orthographic projection that maps `-1.0` to `1.0` to the vertical axis of the window, and respects the window's aspect ratio for the horizontal axis:

See how Bevy constructs its camera bundles, for reference: [orthographic](#), [perspective](#).

This example is based on the setup for a 2D camera:

```
use bevy::render::primitives::Frustum;
use bevy::render::camera::{Camera, Camera2d, CameraProjection, DepthCalculation};
use bevy::render::camera::VisibleFrustum;
```



```
use bevy::render::view::visible_entities;
```

```
#[derive(Component)]
struct SimpleOrthoProjection {
    near: f32,
    far: f32,
    aspect: f32,
}

impl CameraProjection for SimpleOrthoProjection {
    fn get_projection_matrix(&self) -> Mat4 {
        Mat4::orthographic_rh(
            -self.aspect, self.aspect, -1.0, 1.0, self.near, self.far
        )
    }

    // what to do on window resize
    fn update(&mut self, width: f32, height: f32) {
        self.aspect = width / height;
    }

    fn depth_calculation(&self) -> DepthCalculation {
        // for 2D (camera doesn't rotate)
        DepthCalculation::ZDifference

        // otherwise
        //DepthCalculation::Distance
    }

    fn far(&self) -> f32 {
        self.far
    }
}

impl Default for SimpleOrthoProjection {
    fn default() -> Self {
        Self { near: 0.0, far: 1000.0, aspect: 1.0 }
    }
}

fn setup(mut commands: Commands) {
    // We need all the components that Bevy's built-in camera bundles would add

    let projection = SimpleOrthoProjection::default();
    let camera = Camera {
        near: projection.near,
        far: projection.far,
        ..default()
    };

    // position the camera like bevy would do by default for 2D:
    let transform = Transform::from_xyz(0.0, 0.0, projection.far - 0.1);
    // frustum construction code copied from Bevy
    let view_projection =
        projection.get_projection_matrix() * transform.compute_matrix().inverse();
    let frustum = Frustum::from_view_projection(
        &view_projection,
        &transform.translation,
        &transform.back(),
        projection.far,
    );

    commands.spawn bundle((
```

```

        camera,
        projection,
        frustum,
        VisibleEntities::default(),
        transform,
        GlobalTransform::default(),
        Camera2d,
    ));
}

fn main() {
    // need to add a bevy-internal camera system to update
    // the projection on window resizing

    use bevy::render::camera::camera_system;

    App::new()
        .add_plugins(DefaultPlugins)
        .add_startup_system(setup)
        .add_system_to_stage(
            CoreStage::PostUpdate,
            camera_system::<SimpleOrthoProjection>,
        )
        .run();
}

```

Pan + Orbit Camera

[Click here for the full example code.](#)

This code is a community contribution.

Current version developed by [@mirenbharta](#). Initial work by [@skairunner](#).

This is a camera controller similar to the ones in 3D editors like Blender.

Use the right mouse button to rotate, middle button to pan, scroll wheel to move inwards/outwards.

This is largely shown for illustrative purposes, as an example to learn from. In your projects, you may want to try the `bevy_config_cam` plugin.

```
/// Tags an entity as capable of panning and orbiting.  
#[derive(Component)]  
struct PanOrbitCamera {
```

```

struct PanOrbitCamera {
    /// The "focus point" to orbit around. It is automatically updated when panning the
    camera
    pub focus: Vec3,
    pub radius: f32,
    pub upside_down: bool,
}

impl Default for PanOrbitCamera {
    fn default() -> Self {
        PanOrbitCamera {
            focus: Vec3::ZERO,
            radius: 5.0,
            upside_down: false,
        }
    }
}

/// Pan the camera with middle mouse click, zoom with scroll wheel, orbit with right
mouse click.
fn pan_orbit_camera(
    windows: Res<Windows>,
    mut ev_motion: EventReader<MouseMotion>,
    mut ev_scroll: EventReader<MouseWheel>,
    input_mouse: Res<Input<MouseButton>>,
    mut query: Query<(&mut PanOrbitCamera, &mut Transform, &PerspectiveProjection)>,
) {
    // change input mapping for orbit and panning here
    let orbit_button = MouseButton::Right;
    let pan_button = MouseButton::Middle;

    let mut pan = Vec2::ZERO;
    let mut rotation_move = Vec2::ZERO;
    let mut scroll = 0.0;
    let mut orbit_button_changed = false;

    if input_mouse.pressed(orbit_button) {
        for ev in ev_motion.iter() {
            rotation_move += ev.delta;
        }
    } else if input_mouse.pressed(pan_button) {
        // Pan only if we're not rotating at the moment
        for ev in ev_motion.iter() {
            pan += ev.delta;
        }
    }
    for ev in ev_scroll.iter() {
        scroll += ev.y;
    }
    if input_mouse.just_released(orbit_button) ||
input_mouse.just_pressed(orbit_button) {
        orbit_button_changed = true;
    }

    for (mut pan_orbit, mut transform, projection) in query.iter_mut() {
        if orbit_button_changed {
            // only check for upside down when orbiting started or ended this frame
            // if the camera is "upside" down, panning horizontally would be inverted,
            so invert the input to make it correct

            let up = transform.rotation * Vec3::Y;
            pan_orbit.upside_down = up.y <= 0.0;
        }
    }
}

```

```

let mut any = false;
if rotation_move.length_squared() > 0.0 {
    any = true;
    let window = get_primary_window_size(&windows);
    let delta_x = {
        let delta = rotation_move.x / window.x * std::f32::consts::PI * 2.0;
        if pan_orbit.upside_down { -delta } else { delta }
    };
    let delta_y = rotation_move.y / window.y * std::f32::consts::PI;
    let yaw = Quat::from_rotation_y(-delta_x);
    let pitch = Quat::from_rotation_x(-delta_y);
    transform.rotation = yaw * transform.rotation; // rotate around global y
axis
    transform.rotation = transform.rotation * pitch; // rotate around local x
axis
} else if pan.length_squared() > 0.0 {
    any = true;
    // make panning distance independent of resolution and FOV,
    let window = get_primary_window_size(&windows);
    pan *= Vec2::new(projection.fov * projection.aspect_ratio, projection.fov)
/ window;
    // translate by local axes
    let right = transform.rotation * Vec3::X * -pan.x;
    let up = transform.rotation * Vec3::Y * pan.y;
    // make panning proportional to distance away from focus point
    let translation = (right + up) * pan_orbit.radius;
    pan_orbit.focus += translation;
} else if scroll.abs() > 0.0 {
    any = true;
    pan_orbit.radius -= scroll * pan_orbit.radius * 0.2;
    // dont allow zoom to reach zero or you get stuck
    pan_orbit.radius = f32::max(pan_orbit.radius, 0.05);
}

if any {
    // emulating parent/child to make the yaw/y-axis rotation behave like a
turntable
    // parent = x and y rotation
    // child = z-offset
    let rot_matrix = Mat3::from_quat(transform.rotation);
    transform.translation = pan_orbit.focus +
rot_matrix.mul_vec3(Vec3::new(0.0, 0.0, pan_orbit.radius));
}
}

fn get_primary_window_size(windows: &Res<Windows>) -> Vec2 {
    let window = windows.get_primary().unwrap();
    let window = Vec2::new(window.width() as f32, window.height() as f32);
    window
}

/// Spawn a camera like this
fn spawn_camera(mut commands: Commands) {
    let translation = Vec3::new(-2.0, 2.5, 5.0);
    let radius = translation.length();

    commands.spawn_bundle(PerspectiveCameraBundle {
        transform: Transform::from_translation(translation)
            .looking_at(Vec3::ZERO, Vec3::Y),
        ..Default::default()
    });
}

```

```
}).insert(PanOrbitCamera {  
    radius,  
    ..Default::default()  
});  
}
```

List All Resource Types

[Click here for the full example code.](#)

This example shows how to print a list of all types that have been added as [resources](#).

```
fn print_resources(archetypes: &Archetypes, components: &Components) {
    let mut r: Vec<String> = archetypes
        .resource()
        .components()
        .map(|id| components.get_info(id).unwrap())
        // get_short_name removes the path information
        // i.e. `bevy_audio::audio::Audio` -> `Audio`
        // if you want to see the path info replace
        // `TypeRegistration::get_short_name` with `String::from`
        .map(|info| TypeRegistration::get_short_name(info.name()))
        .collect();

    // sort list alphabetically
    r.sort();
    r.iter().for_each(|name| println!("{}", name));
}
```

Note that this does *not* give you a comprehensive list of every Bevy-provided type that is useful as a resource. It lists the types of all the resources *currently added* to the app (by all registered plugins, your own, etc.).

[See here for a more useful list types provided in Bevy.](#)

Bevy on Different Platforms

This chapter is a collection of platform-specific information, about using Bevy with different operating systems or environments.

Feel free to suggest things to add.

Bevy trivially works out-of-the-box on the major desktop operating systems: Linux, macOS, Windows. No special configuration is required.

See the following pages for specific tips/advice when developing for the desktop platforms:

- [Linux](#)
- [macOS](#)
- [Windows](#)

Bevy aims to also make it easy to target other platforms, such as web browsers (via WebAssembly), mobile (Android and iOS), and game consoles. Your Bevy code can be the same for all platforms, with differences only in the build process and environment setup.

However, that vision is not fully met yet. Currently, support for non-desktop platforms is limited, and requires more complex configuration:

- [Web Browsers](#): Bevy works quite well on web, but with some limitations.
- Mobile: support is minimal and broken. It will build, but may or may not run. Expect to immediately encounter major issues.
- Game consoles: support is still completely non-existent yet.

If you are interested in these other platforms and you'd like to help improve Bevy's cross-platform support, your contributions would be greatly welcomed!

If you have any additional Linux-specific knowledge, please help improve this page!

Create Issues or PRs on [GitHub](#).

Desktop Linux is one of the best-supported platforms by Bevy.

There are some development dependencies you may need to setup, depending on your distribution. [See instructions in official Bevy repo](#).

[See here](#) if you also want to build Windows EXEs from Linux.

GPU Drivers

Support for the Vulkan graphics API is required to run Bevy apps. You (and your users) must ensure that you have compatible hardware and drivers installed.

On most modern distributions and computers, this should be no problem.

If Bevy apps refuse to run and print an error to the console about not being able to find a compatible GPU, the problem is most likely with the Vulkan components of your graphics driver not being installed correctly. You may need to install some extra packages or reinstall your graphics drivers. Check with your Linux distribution for what to do.

X11 and Wayland

As of the year 2021, the Linux desktop ecosystem is fragmented between the legacy X11 stack and the modern Wayland stack. Many distributions are switching to Wayland-based desktop environments by default.

Bevy supports both, but only X11 support is enabled by default. If you are running a Wayland-based desktop, this means your Bevy app will run in the XWayland compatibility layer.

To enable native Wayland support for Bevy, enable the `wayland` cargo feature:

```
[dependencies]
bevy = { version = "0.7", features = ["wayland"] }
```

Now your app will be built with support for both X11 and Wayland.

If you want to remove X11 support for whatever reason, disable the `x11` cargo default feature.

You can override which display protocol to use at runtime, using an environment variable:

```
export WINIT_UNIX_BACKEND=x11
```

(to run using X11/XWayland on a Wayland desktop)

or

or

```
export WINIT_UNIX_BACKEND=wayland
```

(to require the use of Wayland)

If you have any additional macOS-specific knowledge, please help improve this page!

Create Issues or PRs on [GitHub](#).

(this page is currently empty; please help!)

If you have any additional Windows-specific knowledge, please help improve this page!

Create Issues or PRs on [GitHub](#).

Windows is one of the best-supported platforms by Bevy.

Both the MSVC and the GNU compiler toolchains should work.

You can also build Windows EXEs while working in Linux.

Distributing Your App

The EXE built with `cargo build` can work standalone without any extra files or DLLs.

Your `assets` folder needs be distributed alongside it. Bevy will search for it in the same directory as the EXE on the user's computer.

The easiest way to give your game to other people to play is to put them together in a ZIP file. If you use some other method of installation, install the `assets` folder and the EXE to the same path.

Disabling the Windows Console

By default, when you run a Bevy app (or any Rust program for that matter) on Windows, a Console window also shows up. To disable this, place this Rust attribute at the top of your `main.rs`:

```
#![windows_subsystem = "windows"]
```

This tells Windows that your executable is a graphical application, not a command-line program. Windows will know not display a console.

However, the console can be useful for development, to see log messages. You can disable it only for release builds, and leave it enabled in debug builds, like this:

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

Creating an icon for your app

There are two places where you might want to put your application icon:

- The EXE file (how it looks in the file explorer)
- The window at runtime (how it looks in the taskbar and the window title bar)

Setting the EXE icon

(adapted from [here](#))

The EXE icon can be set using a cargo build script.

Add a build dependency of `embed_resources` to your `Cargo.toml` allow embedding assets into your compiled executables

```
[build-dependencies]
embed-resource = "1.6.3"
```

Create a `build.rs` file in your project folder:

```
extern crate embed_resource;

fn main() {
    let target = std::env::var("TARGET").unwrap();
    if target.contains("windows") {
        embed_resource::compile("icon.rc");
    }
}
```

Create a `icon.rc` file in your project folder:

```
app_icon ICON "icon.ico"
```

Create your icon as `icon.ico` in your project folder.

Setting the Window Icon

See [Bevy Cookbook: Setting the Window Icon](#).

Cross-Compilation

This sub-chapter covers how to make builds of your Bevy apps to run on a different Operating System than the one you are working on.

- [Create Windows EXEs from Linux](#)

(also check out the [Windows Platform page](#) for info about developing for Windows generally)

Rust offers two different toolchains for building for Windows:

- **MSVC**: the default when working in Windows, requires downloading Microsoft SDKs
- **GNU**: alternative MINGW-based build, may be easier to setup

First-Time Setup (MSVC)

Rust Toolchain (MSVC)

You can actually use the same MSVC-based Rust toolchain, that is the standard when working on Windows, from Linux.

Add the target to your Rust installation (assuming you use `rustup`):

```
rustup target add x86_64-pc-windows-msvc
```

This installs the files Rust needs to compile for Windows, including the Rust standard library.

Microsoft Windows SDKs

You need to install the Microsoft Windows SDKs, just like when working on Windows. On Linux, this can be done with an easy script called `xwin`. You need to accept Microsoft's proprietary license.

Install `xwin`:

```
cargo install xwin
```

Now, use `xwin` to accept the Microsoft license, download all the files from Microsoft servers, and install them to a directory of your choosing.

For example, to install to `/opt/xwin/`:

```
xwin --accept-license 1 splat --output /opt/xwin
```

Linking (MSVC)

Rust needs to know how to link the final EXE file.

The default Microsoft linker (`link.exe`) is not available on Linux. Instead, we need to use the LLD linker (this is also recommended when working on Windows anyway). Just install the `lld` package

from your Linux distro.

We also need to tell Rust the location of the Microsoft Windows SDK libraries (that were installed with `xwin` in [the previous step](#)).

Add this to `.cargo/config.toml` (in your home folder or in your bevy project):

```
[target.x86_64-pc-windows-msvc]
linker = "lld"
rustflags = [
    "-Lnative=/opt/xwin/crt/lib/x86_64",
    "-Lnative=/opt/xwin/sdk/lib/um/x86_64",
    "-Lnative=/opt/xwin/sdk/lib/ucrt/x86_64"
]
```

First-Time Setup (GNU)

Rust Toolchain (GNU)

You can also use the alternative GNU-based Windows toolchain.

Add the target to your Rust installation (assuming you use `rustup`):

```
rustup target add x86_64-pc-windows-gnu
```

This installs the files Rust needs to compile for Windows, including the Rust standard library.

MINGW

The GNU toolchain requires the MINGW environment to be installed. Your distro likely provides a package for it. Search your distro for a cross-compilation mingw package.

It might be called something like: `cross-x86_64-w64-mingw32`, but that varies in different distros.

You don't need any files from Microsoft.

Building Your Project

Finally, with all the setup done, you can just build your Rust/Bevy projects for Windows:

```
cargo build --target=x86_64-pc-windows-msvc --release
```

```
cargo build --target=x86_64-pc-windows-gnu --release
```

Browser (WebAssembly)

Introduction

You can make web browser games using Bevy. This chapter will help you with the things you need to know to do it. This page gives an overview of Bevy's Web support.

Your Bevy app will be compiled for WebAssembly (WASM), which allows it to be embedded in a web page and run inside the browser.

Performance will be limited, as WebAssembly is slower than native code and does not currently support multithreading.

Not all 3rd-party plugins are compatible. If you need extra unofficial plugins, you will have to check if they are compatible with WASM.

Project Setup

The same Bevy project, without any special code modifications, can be built for either web or desktop/native.

However, you will need a "website" with some HTML and JavaScript to load and run your game. For development and testing, this could be just a minimal shim. It can be easily autogenerated.

To deploy, you will need a server to host your website for other people to access. You could use GitHub's hosting service: [GitHub Pages](#).

Additional Caveats

When users load your site to play your game, their browser will need to download the files. [Optimizing for size](#) is important, so that your game can load fast and not waste data bandwidth.

Some minor extra configuration is needed to be able to:

- [See Rust panic messages](#)

Quick Start

First, add WASM support to your Rust installation. Using Rustup:

```
rustup target install wasm32-unknown-unknown
```

Now, to run your Bevy project in the browser.

wasm-server-runner

The easiest and most automatic way to get started is the `wasm-server-runner` tool.

Install it:

```
cargo install wasm-server-runner
```

Set up `cargo` to use it, in `.cargo/config.toml` (in your project folder, or globally in your user home folder):

```
[target.wasm32-unknown-unknown]  
runner = "wasm-server-runner"
```

Now you can just run your game with:

```
cargo run --target wasm32-unknown-unknown
```

It will automatically run a minimal local webserver and open your game in your browser.

wasm-bindgen

`wasm-bindgen` is the tool to generate all the files needed to put the game on your website.

Run:

```
cargo build --release --target wasm32-unknown-unknown  
wasm-bindgen --out-dir ./out/ --target web ./target/
```

`./out/` is the directory where it will place the output files.

You need to put these on your web server.

Higher-level Tools

Here are some higher-level alternatives. These tools can do more for you and automate more of your workflow, but are more opinionated in how they work.

- [Trunk](#)
- `wasm-pack`

Panic Messages

Unless we do something about it, you will not be able to see Rust panic messages when running in a web browser. This means that, if your game crashes, you will not know why.

To fix this, we can set up a panic hook that will cause the messages to appear in the browser console, using the [console_error_panic_hook](#) crate.

Add the crate to your dependencies in `Cargo.toml`:

```
[dependencies]
console_error_panic_hook = "0.1"
```

At the start of your main function, before doing anything else, add this:

```
// When building for WASM, print panics to the browser console
#[cfg(target_arch = "wasm32")]
console_error_panic_hook::set_once();
```

Optimize for Size

When serving a WASM binary, the smaller it is, the faster the browser can download it. Faster downloads means faster page load times and less data bandwidth use, and that means happier users.

This page gives some suggestions for how to make your WASM files smaller.

Do not prematurely optimize! You probably don't need small WASM files during development, and many of these techniques can get in the way of your workflow! They may come at the cost of longer compile times or less debuggability.

Depending on the nature of your application, your mileage may vary, and performing measurements of binary size and execution speed is recommended.

[Twiggy](#) is a code size profiler for WASM binaries, which you can use to make measurements.

For additional information and more techniques, refer to the Code Size chapter in the [Rust WASM book](#).

Compiling for size instead of speed

You can change the optimization profile of the compiler, to tell it to prioritize small output size, rather than performance.

(although in some rare cases, optimizing for size can actually improve speed)

In `Cargo.toml`, add one of the following:

```
[profile.release]
opt-level = 's'
```

```
[profile.release]
opt-level = 'z'
```

These are two different profiles for size optimization. Usually, `z` produces smaller files than `s`, but sometimes it can be the opposite. Measure to confirm which one works better for you.

Link-Time Optimization (LTO)

In `Cargo.toml`, add the following:

```
[profile.release]
lto = "thin"
```

LTO tells the compiler to optimize all code together, considering all crates as if they were one. It may be able to inline and prune functions much more aggressively.

This typically results in smaller size *and* better performance, but do measure to confirm. Sometimes, the size can actually be larger.

The downside here is that compilation will take much longer. Do this only for release builds you publish for other users.

Use the `wasm-opt` tool

The [binaryen](#) toolkit is a set of extra tools for working with WASM. One of them is `wasm-opt`. It goes much further than what the compiler can do, and can be used to further optimize for either speed or size:

```
# Optimize for size (s profile).
wasm-opt -Os -o output.wasm input.wasm

# Optimize for size (z profile).
wasm-opt -Oz -o output.wasm input.wasm

# Optimize aggressively for speed.
wasm-opt -O3 -o output.wasm input.wasm

# Optimize aggressively for both size and speed.
wasm-opt -O -ol 100 -s 100 -o output.wasm input.wasm
```

Do you know of more WASM size-optimization techniques? Post about them in the [GitHub Issue Tracker](#) so that they can be added to this page!

Hosting on GitHub Pages

GitHub Pages is a hosting service that allows you to publish your website on GitHub's servers.

For more details, visit the official [GitHub Pages documentation](#).

Deploying a website (like your WASM game) to GitHub pages is done by putting the files in a special branch in a GitHub repository. You could create a separate repository for this, but you could also do it from the same repository as your source code.

You will need the final website files for deployment.

Create an empty branch in your git repository:

```
git checkout --orphan web
git reset --hard
```

You should now be in an empty working directory.

Put all files necessary for hosting, including your HTML, WASM, JavaScript, and `assets` files, and commit them into git:

```
git add *
git commit
```

(or better, manually list your files in the above command, in place of the `*` wildcard)

Push your new branch to GitHub:

```
git push -u origin web --force
```

In the GitHub Web UI, go to the repository settings, go to the "GitHub Pages" section, then under "Source" pick the branch "web" and the `/` (root) folder. Then click "Save".

Wait a little bit, and your site should become available at `https://your-name.github.io/your-repo`.

While the majority of this book was authored by me, Ida Iyes ([@inodentry](#)), a number of folks have made significant contributions to help! Thank you all so much! ❤️

- Alice I. Cecile [@alice-i-cecile](#) : review, advice, reporting lots of good suggestions
- Nile [@TheRawMeatball](#) : review, useful issue reports
- [@Zaszi](#) : writing the initial draft of the WASM chapter
- [@skairunner](#) and [@mirenbharta](#) : developing the Pan+Orbit camera example
- [@billyb2](#) : the Fixed Timestep example

Thanks to everyone who has submitted [GitHub issues](#) with suggestions!

Big thanks to all [sponsors](#)! ❤️ You all are making this pay off, literally!

Thanks to you, I can actually keep working on this book, improving and maintaining it!

And of course, the biggest thanks goes to the [Bevy project](#) itself and its founder, [@cart](#) , for creating this awesome community and game engine in the first place! It makes all of this possible. You literally changed my life! ❤️

Contact Me

You can find me in the following places:

- Discord: `Ida Iyes#0981`
- Twitter: `@IyesGames`
- GitHub: `@inodentry`
- Reddit: `iyesgames`

For improvements or fixes to this book, please file an issue on [GitHub](#).

If you want to help out the Bevy Game Engine project, check out Bevy's [official contributing guide](#).

Be civil. If you need a code of conduct, have a look at Bevy's.

If you have any suggestions for the book, such as ideas for new content, or if you notice anything that is incorrect or misleading, please file issues in [the GitHub repository](#)!

Contributing Code

If you simply want to contribute code examples to the book, feel free to make a PR. I can take care of writing the book text / page that your code will be displayed on.

Cookbook Examples

The code for cookbook examples should be provided as a full, runnable, example file, under `src/code/examples`. The book page will only show the relevant parts of the code, without unnecessary boilerplate.

Always use [mdbook anchor syntax](#), not line numbers, to denote the parts of the code to be shown on the page.

Credits

If you contribute a cookbook example, I will credit you in the book by your github username with a link to the PR. Please let me know if you prefer not to be credited, or if you would like to be credited in another way (but no commercial self-promotion allowed).

Contributing Book Text

I do not directly merge book text written by other people. This is because I want the book to follow a consistent editorial style.

If you would like to write new content for the book, feel free to make a PR with the content to be included, but note that it will likely not be preserved exactly as you wrote it.

I will likely merge it into a temporary branch and then edit or rewrite it as I see fit, for publishing into the book.

Licensing

To avoid complications with copyright and licensing, you agree to provide any contributions you make to the project under the [MIT-0 No Attribution License](#).

Note that this allows your work to be relicensed without preserving your copyright.

As described previously, the actual published content in the book will be my own derivative work based on your contributions. I will license it consistently with the rest of the book; see: [License](#).

Bevy version

Content written for the current Bevy release, is accepted for the `main` branch of the book.

Content written for new developments in Bevy's main branch, is accepted for the `next` branch of the book, in preparation for the next upcoming Bevy release.

Style Guidelines

Aim for simplicity and minimalism. Do not include things irrelevant to getting the point across.

"Perfection is achieved not when there is nothing more to add, but when there is nothing more to remove."

Don't forget to point out potential gotchas and other relevant practical considerations.

Try to use the most common/standard terminology and keywords, to make things easy to find. Don't come up with new/extra terminology of your own.

Avoid repeating information found elsewhere in the book, prefer linking to it instead.

Code Style

Avoid long lines of code, to keep it readable on small screens.

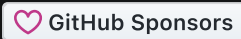
Use reasonable formatting that does not deviate much from the common conventions used by the Rust language community. I don't enforce it strictly; there is no need to use `rustfmt`. If deviating from those standards allows for the code to be presented better in the context of the book, then doing so is preferable.

Text Style

Make it easy to read.

- Be brief. Try to cover all important information without verbose explanations.
- Prefer simple English with short sentences.
- Avoid information overload:
 - Split things into short paragraphs.
 - Avoid introducing many (even if related) topics at the same time.

- Cover advanced usage separately from the basics.



Donate to help me work on this book. Thanks!

I offer professional Bevy tutoring and consulting services. [Contact me](#) if interested!