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 ☆ Star 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.

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

GitHub Sponsors

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

Maintenance Policy

To ease the maintenance burden, the policy of the project is that the book may contain a mixture of content for different versions of Bevy. However, mixing outdated and updated content on the same page is not allowed.

Every page in the book must clearly state what version of Bevy it was written for, at the top of the page, above the main heading. All content on that page must be relevant for the stated Bevy version.

Current Bevy Version

The current Bevy release is 0.10.

The aim is to try to maintain the book up to date with the current latest release of Bevy. New content should be written for this version. Outdated pages will be updated on a best-effort basis.

Old Bevy Version

Content for old releases of Bevy is only allowed in the book if it already exists from before and has not been updated yet.

Unreleased (Bevy's git main branch)

Pages that cover new yet-unreleased features of Bevy are allowed. If there is an exciting new feature, and I want to document it, I will write a page, without waiting for the next release of Bevy to come out. :)

For functionality that exists in the current release in some form, but has changed for the next release, the current version is the one that should be documented. The page can be updated in the future. I might sometimes make exceptions to this rule.

License

- Anything publicly visible on the bevyengine.org website

The MIT-0 license applies as soon as your contribution has been accepted upstream.

GitHub Forks and Pull Requests created for the purposes of contributing to the Official Bevy Project are given the following license exception: the Attribution requirements of CC BY-NC-SA 4.0 are waived for as long as the work is pending upstream review (Pull Request Open). If upstream rejects your contribution, you are given a period of 1 month to comply with the full terms of the CC BY-NC-SA 4.0 license or delete your work. If upstream accepts your contribution, the MIT-0 license applies.

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 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, 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 Cookbook: various code examples beyond the ones in Bevy official repos
- Bevy on Different Platforms: information about working with specific plaforms / OSs

To learn how to program in Bevy, see these chapters:

- Bevy Programming Framework: the ECS+App frameworks, the foundation of everything
- Programming Patterns: opinionated advice, patterns, idioms
- [WIP] Bevy Render (GPU) Framework: working with the GPU and Bevy's rendering

The following chapters cover various Bevy feature areas:

- General Game Engine Features: general features that don't belong in any other chapter
- Asset Management: working with data assets
- Input Handling: working with different input devices
- Window Management: working with the OS window
- Working with 2D: Bevy's features for 2D games
- Working with 3D: Bevy's features for 3D games

Bevy Version:	0.9	(outdated!)

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 join us to 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
- General Game Engine Features: Basic features of Bevy, needed for making any game
    - Coordinate System
    - Transforms
    - Time and Timers
    - Parent/Child Hierarchies
- 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
    - Stages
    - Change Detection
    - Removal Detection
    - Param Sets

- 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
    - System Piping
    - 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

Bevy Version:	0.9	(outdated!)

List of Bevy Builtins

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

SystemParams

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

(List in API Docs)

In regular systems:

- `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
- `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
- `ParallelCommands` : Abstraction to help use `Commands` when you will do your own parallelism
- `SystemName` : The name (string) of the system, may be useful for debugging
- `&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
- `StaticSystemParam` : Helper for generic system abstractions, to avoid lifetime annotations
- tuples containing any of these types, with up to 16 members

In exclusive systems:

- `&mut World` : Full direct access to the ECS World
- `Local<T>` : Data local to the system
- `SystemState<P>` : Emulates a regular system, allowing you to easily access data from the World. `P` are the system parameters.
- `QueryState<Q, F = ()>` : Allows you to perform queries on the World, similar to a `Query` in regular systems.

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.

Systems running during the Extract stage can also use `Extract<T>` , to access data from the Main World instead of the Render World. `T` can be any other system parameter type.

Assets

(more info about working with 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`

Format	Cargo feature	Default?	Filename extensions
BMP	`"bmp"`	No	`.bmp`
DDS	`"dds"`	No	`.dds`
KTX2	`"ktx2"`	No	`.ktx2`
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`, `.comp`

Font formats (loaded as `Font` assets):

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

Bevy Scenes:

Format	Filename extensions
RON-serialized scene	`.scn`, `.scn.ron`

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:

- `SpatialBundle` : Contains the required transform and visibility components that must be included on *all* entities that need rendering or hierarchy
- `TransformBundle` : Contains only the transform types, subset of `SpatialBundle`
- `VisibilityBundle` : Contains only the visibility types, subset of `SpatialBundle`

Scenes:

- `SceneBundle` : Used for spawning scenes
- `DynamicSceneBundle` : Used for spawning dynamic scenes

Bevy 3D:

- `Camera3dBundle` : 3D camera, can use perspective (default) or orthographic projection
- `MaterialMeshBundle` : 3D Object/Primitive: a Mesh and the Material to draw it with
- `PbrBundle` : `MaterialMeshBundle` with the standard Physically-Based Material `StandardMaterial`
- `DirectionalLightBundle` : 3D directional light (like the sun)
- `PointLightBundle` : 3D point light (like a lamp or candle)
- `SpotLightBundle` : 3D spot light (like a projector or flashlight)

Bevy 2D:

- `Camera2dBundle` : 2D camera, uses orthographic projection + other special configuration for 2D
- `SpriteBundle` : 2D sprite (`Image` asset type)
- `SpriteSheetBundle` : 2D sprite (`TextureAtlas` asset type)
- `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:

- `NodeBundle` : Empty node element (like HTML `<div>`)
- `ButtonBundle` : Button element
- `ImageBundle` : Image element
- `TextBundle` : Text element

Resources

(more info about working with resources)

Configuration Resources

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

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
- `Msaa` : 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

- `WinitSettings` : Settings for the OS Windowing backend, including update loop / power-management settings

Settings that are not modifiable at runtime are not represented using resources. Instead, they are configured via the respective plugins.

In Bevy 0.9, there is an exception to this rule:

- `WgpuSettings` : Low-level settings for the GPU API and backends

These settings must be inserted as a resource to the app, at the top, before adding `DefaultPlugins` .

This API inconsistency will be addressed in future versions of Bevy. These settings will be configurable using the plugin, instead of a resource, just like other settings that are only used during engine startup.

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. These resources are in the Main World. See here for the resources in the Render World.

- `Time` : Global time-related information (current frame delta time, time since startup, etc.)
- `AssetServer` : Control the asset system: Load assets, check load status, etc.
- `Assets<T>` : Contains the actual data of the loaded assets of a given type
- `State<T>` : Control over app states
- `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): Raw state of the `winit` backend for each window
- `Audio` : Use this to play sounds via `bevy_audio`
- `SceneSpawner` : Direct control over spawning Scenes into the main app World
- `AppTypeRegistry` : Access to the Reflection Type Registry
- `FixedTimesteps` : The state of all registered `FixedTimestep` drivers
- `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)

- `FrameCount` : Global time-related information (current frame delta time, time since startup, etc.)
- `Diagnostics` : Diagnostic data collected by the engine (like frame times)
- `NonSendMarker` : Dummy resource to ensure a system always runs on the main thread

Render World Resources

These resources are present in the Render World. They can be accessed from rendering systems (that run during render stages).

- `RenderGraph` : The Bevy Render Graph
- `PipelineCache` : Bevy's manager of render pipelines. Used to store render pipelines used by the app, to avoid recreating them more than once.
- `TextureCache` : Bevy's manager of temporary textures. Useful when you need textures to use internally during rendering.
- `RenderAssets<T>` : Contains handles to the GPU representations of currently loaded asset data
- `DefaultImageSampler` : The default sampler for `Image` asset textures
- `FallbackImage` : Dummy 1x1 pixel white texture. Useful for shaders that normally need a texture, when you don't have one available.

There are many other resources in the Render World, which are not mentioned here, either because they are internal to Bevy's rendering algorithms, or because they are just extracted copies of the equivalent resources in the Main World.

Low-Level `wgpu` Resources

Using these resources, you can have direct access to the `wgpu` APIs for controlling the GPU. These are available in both the Main World and the Render World.

- `RenderDevice` : The GPU device, used for creating hardware resources for rendering/compute
- `RenderQueue` : The GPU queue for submitting work to the hardware
- `RenderAdapter` : Handle to the physical GPU hardware
- `RenderAdapterInfo` : 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.

- `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
- `Gamepads` : Registry of all the connected `Gamepad` IDs

Events

(more info about working with 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`

Engine Events

Events related to various internal things happening during the normal runtime of a Bevy app.

- `AssetEvent<T>` : Sent by Bevy when asset data has been added/modified/removed; can be used to detect changes to assets

- `HierarchyEvent` : Sent by Bevy when entity parents/children change
- `AppExit` : Tell Bevy to shut down

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
- `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)
- `ComputedVisibility` : Check if an entity should be rendered (is it hidden? is it culled?)
- `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
- `UiCameraConfig` : Can be used to disable or configure UI rendering for a specific camera
- `Camera2d` : Configuration parameters for 2D cameras
- `Camera3d` : Configuration parameters for 3D cameras
- `OrthographicProjection` : Orthographic projection for a 2D camera
- `Projection` : Projection for a 3D camera (perspective or orthographic)
- `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
- `SpotLight` : (3D) Properties of a spot 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
- `BackgroundColor` : (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 (`StartupStage` , run once at app startup): `PreStartup` , `Startup` , `PostStartup`
- In the main app (`CoreStage` , run every frame update): `First` , `PreUpdate` , `Update` , `PostUpdate` , `Last`
- In the render sub-app (`RenderStage`): `Extract` , `Prepare` , `Queue` , `PhaseSort` , `Render` , `Cleanup`

The Render Stages are each intended for a specific purpose:

- `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

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

Bevy Version:	0.10	(current)

Getting Started

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.

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"
edition = "2021"

[dependencies]
bevy = "0.10"
```

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.

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.

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 should work as a fallback, for systems that do not support other APIs, but might not be in such a good state as other APIs. DirectX 11 support for legacy systems is planned, but not available yet.

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

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

Bevy Version:	main	(development)

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 freshest 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.

You may want to consider forking the repositories of any plugins you use. This allows you to easily apply fixes if needed, or edit their `Cargo.toml` for any special configuration to make your project work.

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.

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 your own forks, check that the dependencies are correctly and consistently specified everywhere.

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"] # their bevy commit
bevy = { git = "https://github.com/bevyengine/bevy?rev=a420beb0" } # ours

# 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. Cargo patches only apply when you build your project directly, not as a dependency, so they do not affect your users and can be safely kept in your `Cargo.toml`.

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.

```yaml
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
```

Bevy Version:	0.10	(current)

Text Editor / IDE

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 `BEVY_ASSET_ROOT` or `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.

If this is not set, Bevy will search for `assets` alongside the executable binary, in the same folder where it is located. This makes things easy for distribution. However, during development, since your executable is located in the `target` directory where `cargo` placed it, Bevy will be unable to find the `assets`.

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}",
    }
}
```

IntelliJ

When using queries, type information gets lost due to Bevy relying on procedural macros. You can fix this by enabling procedural macro support in the IDE.

1. type `Experimental feature` in the dialog of the `Help | Find Action` action
2. enable the features `org.rust.cargo.evaluate.build.scripts` and `org.rust.macros.proc`

Bevy Version:	0.10	(current)

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 Schedules (all of the registered systems with their ordering dependencies), and the Bevy Render Graph.

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.

Bevy Version:	0.10	(current)

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.

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.

Bevy Version:	0.10	(current)

Configuring Bevy

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 = "0.10"
# 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:
  "bevy_asset",         # Assets management
  "bevy_audio",         # Builtin audio
  "bevy_gilrs",         # Gamepad input support
  "bevy_scene",         # Scenes management
  "bevy_winit",         # Window management
  "bevy_render",        # Rendering framework core
  "bevy_core_pipeline", # Common 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
  "animation",          # Animation support
  "tonemapping_luts",   # Support different camera Tonemapping modes (embeds
extra data)
  "filesystem_watcher", # Asset hot-reloading
  "x11",                # Linux: Support X11 windowing system
  "android_shared_stdcxx", # For Android builds, use shared C++ library

  # File formats:
  "png",    # PNG image format for simple 2D images
  "hdr",    # HDR images
  "ktx2",   # Preferred format for GPU textures
  "zstd",   # ZSTD compression support in KTX2 files
  "vorbis", # Audio: OGG 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
  "bevy_dynamic_plugin",  # Support for loading of `DynamicPlugin`s
  "accesskit_unix",       # AccessKit integration for UI Accessibility

  # File formats:
  "dds",  # Alternative DirectX format for GPU textures, instead of KTX2
  "jpeg", # JPEG lossy format for 2D photos
  "bmp",  # Uncompressed BMP image format
  "tga",  # Truevision Targa image format
  "exr",  # OpenEXR advanced image format
  "basis-universal", # Basis Universal GPU texture compression format
```

```
    "flac", # Audio: FLAC lossless format
    "mp3",  # Audio: MP3 format (not recommended)
    "wav",  # Audio: Uncompressed WAV
    "symphonia-all", # All Audio formats supported by the Symphonia library

    # Development/Debug features:
    "dynamic_linking", # Dynamic linking for faster compile-times
    "trace",           # Enable tracing for performance measurement
    "detailed_trace",  # Make traces more verbose
    "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 `bevy_winit` and your selection of Rendering features. For Linux support, you need at least one of `x11` or `wayland`.

`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. If you want something with more features, you can use the `bevy_kira_audio` plugin instead. Disable `bevy_audio` and `vorbis`.

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.

Development Features

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

Asset hot-reloading

The `filesystem_watcher` feature controls support for hot-reloading of assets, supported on desktop platforms.

Dynamic Linking

`dynamic_linking` 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 and macOS are also supported, but are less tested and have had issues in the past.

Do not enable this for release builds you intend to publish to other people, unless you have a very good special reason to and you know what you are doing. 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_linking
```

You could also add this to your IDE/editor configuration.

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.

See Bevy's official docs on profiling to learn more.

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.

Bevy Version:	0.10	(current)

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"
```

Bevy Version:	0.10	(current)

Performance

Unoptimized debug builds

You can partially enable compiler optimizations in debug/dev mode!

You can enable higher optimizations for dependencies (incl. Bevy), but not your own code, to keep recompilations fast!

In `Cargo.toml` or `.cargo/config.toml`:

```
# Enable max optimizations for dependencies, but not for our code:
[profile.dev.package."*"]
opt-level = 3
```

The above is enough to make Bevy run fast. It will only slow down clean builds, without affecting recompilation times for your project.

If your own code does CPU-intensive work, you might want to also enable some optimization for it. However, this might greatly affect compile times in some projects (similar to a full release build), so it is not generally recommended.

```
# Enable only a small amount of optimization in debug mode
[profile.dev]
opt-level = 1
```

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 squeeze out even more performance at the cost of very slow compile times:

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

Bevy Version:	0.10	(current)

Obscure Rust compiler errors

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

Common beginner mistakes

- Using `commands: &mut Commands` instead of `mut commands: 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.

Error adding function as system

The errors can look like this:

```
error[E0277]: the trait bound `for<'a, 'b, 'c> fn(...) {system}: IntoSystem<(),
(), _>` is not satisfied
   --> src/main.rs:5:21
    |
5   |         .add_system(my_system)
    |          ---------- ^^^^^^^^^ the trait `IntoSystem<(), (), _>` is not
implemented for fn item `for<'a, 'b, 'c> fn(...) {system}`
    |          |
    |          required by a bound introduced by this call
    |
    = help: the trait `IntoSystemAppConfig<()>` is implemented for
`SystemAppConfig`
    = note: required for `for<'a, 'b, 'c> fn(...) {system}` to implement
`IntoSystemConfig<_>`
    = note: required for `for<'a, 'b, 'c> fn(...) {system}` to implement
`IntoSystemAppConfig<_>`
```

The error (confusingly) points to the place in your code where you try to add the system, but in reality, the problem is actually in the `fn` function definition!

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

Error on malformed queries

You might also errors that look like this:

```
error[E0277]: the trait bound `Transform: WorldQuery` is not satisfied
   --> src/main.rs:10:12
    |
10  |     query: Query<Transform>,
    |            ^^^^^^^^^^^^^^^ the trait `WorldQuery` is not implemented for
`Transform`
    |
    = help: the following other types implement trait `WorldQuery`:
              &'__w mut T
              &T
              ()
              (F0, F1)
              (F0, F1, F2)
              (F0, F1, F2, F3)
              (F0, F1, F2, F3, F4)
              (F0, F1, F2, F3, F4, F5)
            and 54 others
note: required by a bound in `bevy::prelude::Query`
   --> ~/.cargo/registry/src/index.crates.io-6f17d22bba15001f/bevy_ecs-0.10.0
/src/system/query.rs:276:37
    |
276 | pub struct Query<'world, 'state, Q: WorldQuery, F: ReadOnlyWorldQuery =
()> {
    |                                     ^^^^^^^^^^ required by this bound in
`Query`
```

To access your components, you need to use reference syntax (`&` or `&mut`).

```
error[E0107]: struct takes at most 2 generic arguments but 3 generic arguments
were supplied
   --> src/main.rs:10:12
    |
10  |     query: Query<&Transform, &Camera, &GlobalTransform>,
    |            ^^^^^                      ---------------- help: remove this
generic argument
    |            |
    |            expected at most 2 generic arguments
    |
note: struct defined here, with at most 2 generic parameters: `Q`, `F`
   --> ~/.cargo/registry/src/index.crates.io-6f17d22bba15001f/bevy_ecs-0.10.0
/src/system/query.rs:276:12
    |
276 | pub struct Query<'world, 'state, Q: WorldQuery, F: ReadOnlyWorldQuery =
()> {
    |            ^^^^^                -
--------------------------
```

When you want to query for multiple components, you need to put them in a tuple: `(&Transform, &Camera, &GlobalTransform)`.

Bevy Version:	0.10	(current)

2D objects not displaying

Bevy's 2D coordinate space is set up so that your background can be at Z=0.0, and other 2D objects can be layered 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 (`Camera2dBundle::default()`).

By default, when you create a 2D camera using Bevy's built-in Bundle constructor, 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.

```
// ✗ INCORRECT: we are creating a new transform with the given rotation
// it does not have the correct Z,
// because it overrides the Transform from the bundle default
commands.spawn(Camera2dBundle {
    transform:
Transform::from_rotation(Quat::from_rotation_z(30.0f32.to_radians())),
    ..Default::default()
});

// ✓ OKAY: we can set the XYZ position ourselves.
// Z=999.9 is what Bevy's default uses.
commands.spawn(Camera2dBundle {
    transform: Transform::from_xyz(0.0, 0.0, 999.9)
        .with_rotation(Quat::from_rotation_z(30.0f32.to_radians())),
    ..Default::default()
});

// ✓ OKAY: we can create the bundle in a `mut` variable,
// and then modify only what we care about
// This way, Bevy's defaults remain intact
let mut my_camera = Camera2dBundle::default();
my_camera.transform.rotation = Quat::from_rotation_z(30.0f32.to_radians());
commands.spawn(my_camera);
```

Bevy Version:	0.9	(outdated!)

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

Some others that may be required:

- UVs (if using textures in the material)
- Tangents (only if using normal maps, otherwise not required)

If you are generating your own mesh data, make sure to provide everything you need.

If you are loading meshes from asset files, make sure they include everything that is needed (check your export settings).

If you need Tangents for normal maps, it is recommended that you include them in your GLTF files. This avoids Bevy having to autogenerate them at runtime. Many 3D editors (like Blender) do not enable this option by default.

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 (`*.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 yet. If your assets are missing mipmaps, textures will look grainy/noisy.

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.

Missing Visibility component on parent

If your entity is in a hierarchy, all its parents need to have a `Visibility` component. It is required even if those parent entities are not supposed to render anything.

Fix it by inserting a `VisibilityBundle`:

```
commands.entity(parent)
    .insert(VisibilityBundle::default());
```

Or better, make sure to spawn the parent entities correctly in the first place. You can use a `VisibilityBundle` or `SpatialBundle` (with transforms) if you are not using a bundle that already includes these components.

Bevy Version:	0.10	(current)

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 Version:	0.10	(current)

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.

Bevy Version:	0.10	(current)

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 inconsistent with the World-coordinate system used everywhere else in Bevy, where the Y axis points up.

It is, however, 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).

OpenGL (and frameworks based on it) is different. If your prior experience is with that, you may find that your textures appear flipped vertically.

If you are using a mesh, make sure it has the correct UV values. If it was created with other software, be sure to select the correct settings.

If you are writing a custom shader, make sure your UV arithmetic is correct.

Sprites

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

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

Bevy Programming Framework

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

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 General Game Engine Features chapter.

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

For additional of programming patterns and idioms, see the Programming Patterns chapter.

Bevy Version:	0.10	(current)

ECS Programming Introduction

This page will try to teach you the general ECS mindset/paradigm.

Relevant official examples: `ecs_guide`.

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

ECS is a programming paradigm that separates data and behavior. Bevy will store all of your data and manage all of your individual pieces of functionality for you. The code will run when appropriate. Your code can get access to whatever data it needs to do its thing.

This makes it easy to write game logic (Systems) in a way that is flexible and reusable. For example, you can implement:

- health and damage that works the same way for anything in the game, regardless of whether that's the player, an NPC, or a monster, or a vehicle
- gravity and collisions for anything that should have physics
- an animation or sound effect for all buttons in your UI

Of course, when you need specialized behavior only for specific entities (say, player movement, which only applies to the player), that is naturally easy to express, too.

Read more about how to represent your data.

Read more about how to represent your functionality.

Bevy Version:	0.10	(current)

Intro: Your Data

This page is an overview, to give you an idea of the big picture of how Bevy works. Click on the various links to be taken to dedicated pages where you can learn more about each concept.

As mentioned in the ECS Intro, Bevy stores all your data for you and allows you easy and flexible access to whatever you need, wherever you need it.

The ECS's data structure is called the `World`. That is what stores and manages all of the data. For advanced scenarios, is possible to have multiple worlds, and then each one will behave as its own separate ECS. However, normally, you just work with the main World that Bevy sets up for your App.

You can represent your data in two different ways: Entities/Components, and Resources.

Entities / Components

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 various components. The `Entity` ID is like the row number. It's an integer index that lets you find specific component values.

Component types that are empty `struct`s (contain no data) are called marker components. They are useful as "tags" to identify specific entities, or enable certain behaviors. For example, you could use them to identify the player entity, to mark enemies that are currently chasing the player, to select entities to be despawned at the end of the level, etc.

Here is an illustration to help you visualize the logical structure. The checkmarks show what component types are present on each entity. Empty cells mean that the component is not present. In this example, we have a player, a camera, and several enemies.

`Entity` (ID)	`Transform`	`Player`	`Enemy`	`Camera`	`Health`	...
...						

Entity (ID)	Transform	Player	Enemy	Camera	Health	...
107	✓ <translation> <rotation> <scale>	✓			✓ 50.0	
108	✓ <translation> <rotation> <scale>		✓		✓ 25.0	
109	✓ <translation> <rotation> <scale>			✓ <camera data>		
110	✓ <translation> <rotation> <scale>		✓		✓ 10.0	
111	✓ <translation> <rotation> <scale>		✓		✓ 25.0	
...						

Representing things this way gives you flexibility. 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).

The typical and obvious pattern is to use entities to represent "objects in the game/scene", such as the camera, the player, enemies, lights, props, UI elements, and other things. However, you are not limited to that. The ECS is a general-purpose data structure. You can create entities and components to store any data. For example, you could create an entity to store a bunch of settings or configuration parameters, or other abstract things.

Data stored using Entities and Components is accessed using queries. For example, if you want to implement a game mechanic, write a system, specify what component types you want to access, and do your thing. You can either iterate through all entities that match your specification, or access specific ones (if you know their `Entity` ID).

Bevy can automatically keep track of what data your systems have access to and run them in

parallel on multiple CPU cores. This way, you get multithreading with no extra effort from you!

If you want to modify the data structure itself (as opposed to just accessing existing data), such as to create or remove entities and components, that requires special consideration. Bevy cannot change the memory layout while other systems might be running. These operations can be buffered/deferred using Commands, to be applied later when it is safe to do so. You can also get direct World access using exclusive systems, if you want to perform such operations immediately (but without multithreading).

Comparison with Object-Oriented Programming

Object-Oriented programming teaches you to think of everything as "objects", where each object is an instance of a "class". The class specifies the data and functionality for all objects of that type, in one place. Every object of that class has the same data (with different values) and the same associated functionality.

This is the opposite of the ECS mentality. In ECS, any entity can have any data (any combination of components). The purpose of entities is to identify that data. Your systems are loose pieces of functionality that can operate on any data. They can easily find what they are looking for, and implement the desired behavior.

If you are an object-oriented programmer, 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.

Then, each piece of functionality (each system) can just query for the data it needs. Common functionality (like a health/damage system) can be applied to any entity with the matching components, regardless of whether that's the player or something else in the game.

However, if some data always makes sense to be accessed together, then you should put it in a single `struct`. For example, Bevy's `Transform` or `Color`. With these types, the fields are not likely to be useful independently.

Additional Internal Details

The set / combination of components that a given entity has, is called the entity's Archetype. Bevy keeps track of that internally, to organize the data in RAM. Entities of the same Archetype have their data stored together, which allows the CPU to access and cache it efficiently.

If you add/remove component types on existing entities, you are changing the Archetype, which may require Bevy to copy the data to a different location.

Learn more about Bevy's component storage.

Resources

If there is only one global instance (singleton) of something, and it is standalone (not associated with other data), create a Resource.

For example, you could create a resource to store your game's graphics settings, or the data for the currently active game mode or session.

This is a simple way of storing data, when you know you don't need the flexibility of Entities/Components.

Bevy Version:	0.10	(current)

Intro: Your Code

This page is an overview, to give you an idea of the big picture of how Bevy works. Click on the various links to be taken to dedicated pages where you can learn more about each concept.

As mentioned in the ECS Intro, Bevy manages all of your functionality/behaviors for you, running them when appropriate and giving them access to whatever parts of your data they need.

Individual pieces of functionality are called systems. Each system is a Rust function (`fn`) you write, which accepts special parameter types to indicate what data it needs to access. Think of the function signature as a "specification" for what to fetch from the ECS `World` .

Here is what a system might look like. Note how, just by looking at the function parameters, we know *exactly* what data can be accessed.

```
fn enemy_detect_player(
    // access data from resources
    mut ai_settings: ResMut<EnemyAiSettings>,
    gamemode: Res<GameModeData>,
    // access data from entities/components
    query_player: Query<&Transform, With<Player>>,
    query_enemies: Query<&mut Transform, (With<Enemy>, Without<Player>)>,
    // in case we want to spawn/despawn entities, etc.
    mut commands: Commands,
) {
    // ... implement your behavior here ...
}
```

(learn more about: systems, queries, commands, resources, entities, components)

Parallel Systems

Based on the parameter types of the systems you write, Bevy knows what data each system can access and whether it conflicts with any other systems. Systems that do not conflict (don't access any of the same data) will be automatically run in parallel on different CPU threads. This way, you get multithreading, utilizing modern multi-core CPU hardware

effectively, with no extra effort from you!

For best parallelism performance, it is recommended that you keep your functionality and your data granular. Create many small systems, each one with a narrowly-scoped purpose and accessing only the data it needs. This gives Bevy more opportunities for parallelism. Putting too much functionality in one system, or too much data in a single component or resource `struct`, limits parallelism.

Bevy's parallelism is non-deterministic by default. Your systems might run in a different and unpredictable order relative to one another, unless you add ordering dependencies to constrain it.

Exclusive Systems

Exclusive systems provide you with a way to get full direct access to the ECS `World`. They cannot run in parallel with other systems, because they can access anything and do anything. Sometimes, you might need this additonal power.

Schedules

Bevy stores systems inside of schedules (`Schedule`). The schedule contains the systems and all relevant metadata to organize them, telling Bevy when and how to run them. Bevy Apps typically contain many schedules. Each one is a collection of systems to be invoked in different scenarios (every frame update, fixed timestep update, at app startup, on state transitions, etc.).

The metadata stored in schedules allows you to control how systems run:

- Add run conditions to control if systems should run during an invocation of the schedule. You can disable systems if you only need them to run sometimes.
- Add ordering constraints, if one system depends on another system completing before it.

Within schedules, systems can be grouped into sets. Sets allow multiple systems to share common configuration/metadata. Systems inherit configuration from all sets they belong to. Sets can also inherit configuration from other sets.

Here is an illustration to help you visualize the logical structure of a schedule. Let's look at how a hypothetical "Update" (run every frame) schedule of a game might be organized.

List of systems:

System name	Sets it belongs to	Run conditions	Ordering constrair
`footstep_sound`	`AudioSet` `GameplaySet`		`after(player_moveme` `after(enemy_movemer`
`player_movement`	`GameplaySet`	`player_alive` `not(cutscene)`	`after(InputSet)`
`camera_movement`	`GameplaySet`		`after(InputSet)`
`enemy_movement`	`EnemyAiSet`		
`enemy_spawn`	`EnemyAiSet`		
`enemy_despawn`	`EnemyAiSet`		`before(enemy_spawn)`
`mouse_input`	`InputSet`	`mouse_enabled`	
`controller_input`	`InputSet`	`gamepad_enabled`	
`background_music`	`AudioSet`		
`ui_button_animate`			
`menu_logo_animate`	`MainMenuSet`		
`menu_button_sound`	`MainMenuSet` `AudioSet`		
...			

List of sets:

Set name	Parent Sets	Run conditions	Ordering constraints
`MainMenuSet`		`in_state(MainMenu)`	
`GameplaySet`		`in_state(InGame)`	
`InputSet`	`GameplaySet`		
`EnemyAiSet`	`GameplaySet`	`not(cutscene)`	`after(player_movement)`
`AudioSet`		`not(audio_muted)`	

Note that it doesn't matter how systems are listed in the schedule. Their order of execution is determined by the metadata. Bevy will respect those constraints, but otherwise run systems in parallel as much as it can, depending on what CPU threads are available.

Also note how our hypothetical game is implemented using many individually-small systems. For example, instead of playing audio inside of the `player_movement` system, we made a separate `play_footstep_sounds` system. These two pieces of functionality probably need to

access different data, so putting them in separate systems allows Bevy more opportunities for parallelism. By being separate systems, they can also have different configuration. The `play_footstep_sounds` system can be added to an `AudioSet` set, from which it inherits a `not(audio_muted)` run condition.

Similarly, we put mouse and controller input in separate systems. The `InputSet` set allows systems like `player_movement` to share an ordering dependency on all of them at once.

You can see how Bevy's scheduling APIs give you a lot of flexibility to organize all the functionality in your game. What will you do with all this power? ;)

Here is how schedule that was illustrated above could be created in code:

```
app.configure_set(MainMenuSet
    .run_if(in_state(MainMenu))
);
app.configure_set(GameplaySet
    .run_if(in_state(InGame))
);
app.configure_set(InputSet
    .in_set(GameplaySet)
);
app.configure_set(EnemyAiSet
    .in_set(GameplaySet)
    .run_if(not(cutscene))
    .after(player_movement)
);
app.configure_set(AudioSet
    .run_if(audio_muted)
);
app.add_systems(
    (
        enemy_movement,
        enemy_spawn,
        enemy_despawn.before(enemy_spawn),
    ).in_set(EnemyAiSet)
);
app.add_systems(
    (
        mouse_input.run_if(mouse_enabled),
        controller_input.run_if(gamepad_enabled),
    ).in_set(InputSet)
);
app.add_systems(
    (
        footstep_sound.in_set(GameplaySet),
        menu_button_sound.in_set(MainMenuSet),
        background_music,
    ).in_set(AudioSet)
);
app.add_systems(
    (
        player_movement
            .run_if(player_alive)
            .run_if(not(cutscene)),
        camera_movement,
    ).in_set(GameplaySet).after(InputSet)
);
app.add_system(ui_button_animate);
app.add_system(menu_logo_animate.in_set(MainMenuSet));
```

(learn more about: schedules, system sets, states, run conditions, system ordering)

Bevy Version:	0.9	(outdated!)

The App

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...

```
fn main() {
    App::new()
        // Bevy itself:
        .add_plugins(DefaultPlugins)

        // resources:
        .insert_resource(StartingLevel(3))
        // if it implements `Default` or `FromWorld`
        .init_resource::<MyFancyResource>()

        // events:
        .add_event::<LevelUpEvent>()

        // systems to run once at startup:
        .add_startup_system(spawn_things)

        // systems to run each frame:
        .add_system(player_level_up)
        .add_system(debug_levelups)
        .add_system(debug_stats_change)
        // ...

        // launch the app!
        .run();
}
```

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.

Builtin Bevy Functionality

The Bevy game engine's own functionality is represented as a plugin group. Every typical Bevy app must first add it, using either:

- `DefaultPlugins` if you are making a full game/app
- `MinimalPlugins` for something like a headless server.

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 convenient system you can add, to close the focused window on pressing the `Esc` key. When all windows are closed, Bevy will quit automatically.

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_system(bevy::window::close_on_esc)
        .run();
}
```

Bevy Version:	0.9	(outdated!)

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