Tampere University

Eero Häihälä

INTEGRATION TESTING IN BEVY, A RUST-BASED ECS GAME ENGINE

MSc thesis
Faculty of Information Technology and Communication Sciences (ITC)
Examiner: Kari Systä
Examiner: Hannu-Matti Järvinen
April 2024

ABSTRACT

Eero Häihälä: Integration testing in Bevy, A Rust-based ECS game engine
Master's thesis
Tampere University
Degree Programme
April 2024

Automated testing in the games industry is trailing traditional software development. This leads to more effort spent on manual testing and more bugs in released games. Interviews with the Finnish games industry and a survey of the Bevy community support the hypothesis that this is an issue of lacklustre tooling. A testing toolkit called BITT was developed to test if that would solve the problem and provide adequate tooling. The survey and insights gained while building BITT indicate that it did not and that the specific tools that will fix the problem may not even be a testing tool. It could turn out to reduce the need to test with other validation methods. For example, the Rust programming language with its strict compiler offers some guarantees about the code without any test implementation work. The ECS game engine architecture seems to have a positive impact on the perception of testing and may also serve as a valuable tool.

Keywords: Bevy, ECS, Game engine, Integration testing, Rust

TIIVISTELMÄ

Eero Häihälä: Integraatiotestaaminen Bevy nimisessä Rust pohjaisessa ECS pelimoottorissa
Diplomityö
Tampereen yliopisto
Tutkinto-ohjelma
Huhtikuu 2024

Automaattinen ohjelmiston testaus on pelialalla yleistä, mutta ohjelmistokehitykseen verrattuna puutteellista. Tästä aiheutuu useita ongelmia. Manuaaliseen testaamiseen kuluu resursseja, joita voisi hyödyntää muualla ja lopputuotteissa on enemmän vikoja. Suomen pelialan toimijoiden haastatteluissa ja Bevy-yhteisön kyselyssä ilmeni, että ongelman juurisyynä voi olla puutteelliset työkalut. Kirjallisuus osuuden ohella kehitettiin BITT-niminen työkalu, joka auttaa testaamaan Bevy-pelimoottorilla tehtyjä pelejä. Kyselyn ja oma arvioni mukaan BITT ei korjaa tilannetta. Puuttuva työkalu ei välttämättä ole edes testaustyökalu perinteisessä mielessä. Esimerkiksi Rust ohjelmointikielen vaatimukset luovat vakuuksia ohjelmiston toimimaan ilman erillisen testikoodin kirjoittamista. ECS-pelimoottoriarkkitehtuuri vaikuttaa positiivisesti kehittäjien näkemyksiin testaamisesta ja voi tulevaisuudessa olla yksi työkalu, joka auttaa testausongelman ratkaisemiseen.

Avainsanat: Bevy, ECS, Pelimoottori, Integraatiotestaus, Rust

PREFACE

This thesis was made possible with the help of the Finnish games industry members who volunteered to be interviewed and the Bevy community. Professors Kari Systä and Hannu-Matti Järvinen were indispensable in guiding the process. And finally, I would like to thank many of my friends who helped with the proofreading.

Copilot and Grammarly were used in the final stages to check the clarity of the text.

Tampere, 8 April 2024

Eero Häihälä

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

Bevy	Open source ECS game engine written in Rust
BITT	Bevy Integration Testing Toolkit, a library built for this thesis.
CI	Continuous integration
CI pipeline	Continuous integration pipeline, a set of automated tests that are performed for each change before integrating into the codebase.
CI runner	Virtualised environment that runs the steps of the CI pipeline
ECS	Entity Component System. A software architecture often used in games
GaaS	Games as a Service, games with a server element and a stream of new content
GUT	Godot Unit Testing, a third-party library for Godot
Hypercasual	Type of usually mobile game that is easy to play, marketed for mass appeal, and cheap to produce.
OOP	Object-oriented programming. A programming paradigm in which entities in the program are modelled with objects containing both data and behaviour.
QA	Quality Assurance, the process of or people responsible for ensuring that the product works the way it is supposed to
UI	User interface
UTF	Unity Test Framework, a tool to test games made in Unity

.

1. INTRODUCTION

Automated testing is a core part of the software development process. However, it is often neglected in game development due to various reasons like moving targets, proximity to the user interface and perceived cost inefficiency. New tools like ECS architecture, the Rust programming language, and test tooling like BITT could improve the cost effectiveness of testing.

To better understand the problem and the Finnish game industry, members of five Finnish game studios were interviewed to see what methods are used and what problems are faced with testing games. The labour-intensive nature of writing tests for a game that may drastically change quickly was cited most often as the reason for insufficient automated testing. The hypothesis is that this is an issue of tooling. With better tools, the effort of writing these tests would be lessened, which should lead to better testing and therefore better software. To validate this, I have written a testing toolkit of various useful tools for a Rust-based generalist ECS game engine called Bevy. This toolkit is called BITT and was evaluated in part by a survey of the Bevy community.

The goal of this thesis is to test the hypothesis that insufficient testing in games is an issue of inadequate tooling. This was done using data from the survey and my assessment of applying BITT to my own game.

2. BACKGROUND

2.1 Software testing and the games industry

Software is malleable beyond what any other engineering discipline works with. This has led to several events within the discipline, like the rise of agile methods. Because changing the code is easy, it is vital to ensure that changes do not have unwanted side effects. This is why using a CI pipeline to run automated tests for each change is considered standard practice, as it can run automated validations for each revision. However, when it comes to game development, the practices vary a lot. All the interviewed developers had some level of test automation, but the code coverage and perceived effectiveness varied greatly.

As a practice, quality assurance has been traditionally undervalued in the games industry (Ruuska, 2015), which is likely a part of the problem. Many tools used for automated testing work best when the software is well defined, and the designs change rarely. Almost all interviewees cited changing designs as the main reason they do not test as much as they probably should. One even noted that the quality of life of QA workers is something that should be considered, and that often in an environment where designs change frequently, it is simpler to have someone repeatedly manually test features than it is to write an automated test. At the very least there is a large grey area between the solidified design decisions that should be tested automatically and the freshly implemented trial feature that may be immediately removed and thus should not be tested. This grey area tends to get tested manually by QA workers. Books like “Producing games from business budgets to creativity and design" by Cohen and Bustamante do not even mention automated testing, instead favouring manual testing with dev and test kits by QA personnel (Cohen & Bustamante, 2009).

2.2 General game engine architectural patterns

To understand ECS, it is helpful to understand how traditional game engines operate. Unity is a good starting point, as it shares a lot with both Godot and Unreal. I have made a helpful diagram to illustrate the data model in Unity games. Terms in quotes are ones used by Unity itself, while terms without quotes are general descriptors:

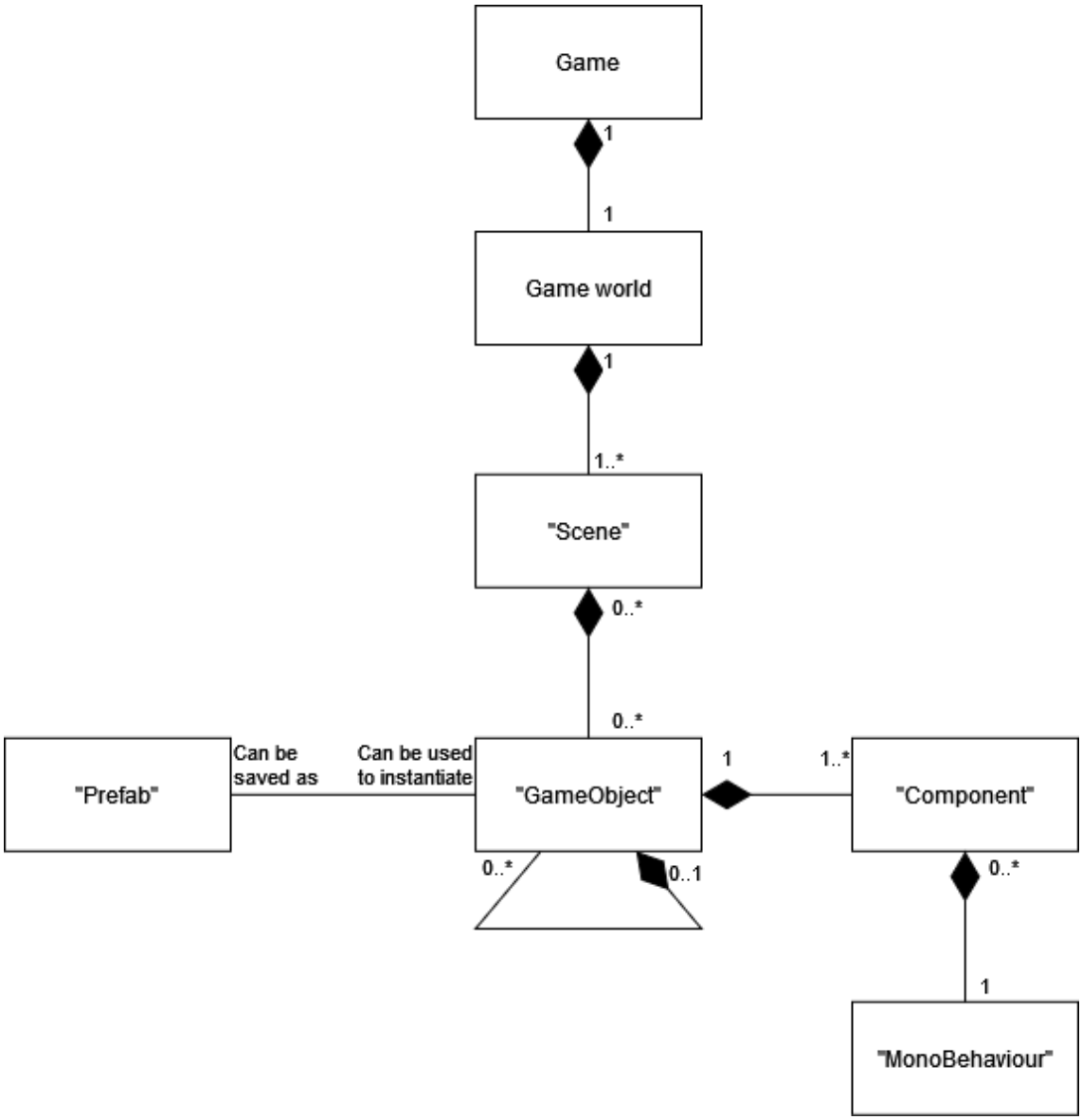

Figure 1 - Diagram of Unity concepts

Several non-essential elements were left out in all the diagrams for readability. The game has a game world as well as some global settings. The game world is then composed of scenes. In Unity, scenes can be saved to disk and loaded later. A scene consists of GameObjects, which represent things in the game world. The player, the ground, the user interface, and light and sound sources are all GameObjects. If an element has a position in the game world, it is represented by a GameObject. GameObjects can be nested to create more complex hierarchies. GameObjects can also be saved on disk, but a saved GameObject is called a Prefab. Prefabs can then be instantiated by game code to dynamically create GameObjects. GameObjects are made up of components. Components provide functionality to the GameObjects their functionality. Components are C# classes that inherit from “MonoBehaviour”, a base class that offers overridable methods for interacting with the game, such as “Start”, which runs on

game start and “Update”, which runs every frame. In addition to the code, the component may hold values that can be set in the Unity editor, a visual editor for tweaking component values among other things. For example, a player movement component may have a configurable speed value that can be easily iterated without code changes. Unity offers other options for writing components besides C#, but C# is the most used and best supported language.

The concepts of GameObjects, scenes and the world are near-universal across traditional engines. As for the dynamic side of game development, an example game was made. In this example game, a player creates projectiles that move further away at a constant velocity. The game can be made with a Unity-like traditional engine, in which case the engine code required for context could look something like this:

Listing 1 - Traditional engine - Example engine code

```
# For syntax highlighting purposes,
# this pseudocode was written to look like python
import engine   # Not an actual module, fixes syntax highlight.

class Component:
    def start(self, game_object: GameObject, world: World):
        # Called at the start of the game
        ...

    def update(self, game_object: GameObject, world: World):
        # Called every frame
        ...

class GameObject:
    # Components contain data and behaviour
    components: list[Component]

    # Children contain GameObjects nested inside this one.
    # Usually an empty list.
    children: list[GameObject]

    # There would likely also be an ID that's forced to be unique.
    # Such IDs tend to not be used in the game code,
    # as holding a reference to a GameObject works.
    name: str

    def get_component(self, component_type: type) -> Component | None:
        # Finds a component of a type on this GameObject
        for component in self.components:
            if isinstance(component, component_type):
                return component
        return None # Can fail to find.
                    # **This** causes many issues with testing

class World:
    # Objects in the world
    game_objects: list[GameObject]

    def get_by_name(self, name: str) -> GameObject | None:
        # This is how Components can find other GameObjects.
```

```
        for game_object in self.game_objects:
            if game_object.name == name:
                # If there are multiple with the same name,
                # it finds one of them.
                # Typically, the one added first.
                return game_object
        return None # Can fail to find,
                    # caller needs to handle that

    def start(self):
        for object in self.game_objects:
            for component in object.components:
                component.start(object, self)

    def loop(self):
        # Primary loop of the game
        while True:
            # Normally, there would be a way to exit gracefully
            # Note the order of execution here:
            # object by object, component by component
            for object in self.game_objects:
                for component in object.components:
                    component.update(object, self)

class Vector:
    x: float
    y: float

# Components required by the engine get implemented in the engine.
class Position(Component):
    position: Vector
```

This would all be provided by the engine. For most projects, this means it comes as is and cannot be modified. Take note of the iteration order for both the primary gameplay loop as well as the start function. The outer loop iterates through the GameObjects and the inner loop iterates through the components. This means it is not trivial to know in a component if another component has already been processed, especially if that other component is on a different GameObject.

The game developer would then implement their part with something that could look like this:

Listing 2 - Traditional engine - Example game code

```
class Velocity(Component):
    velocity: Vector

    def update(self, game_object: GameObject, _world: World):
        pos = game_object.get_component(Position)
        pos.position.x += self.velocity.x
        pos.position.y += self.velocity.y

class Player(Component):
    def update(self, game_object: GameObject, world: World):
        clicked, mouse_position = engine.get_click_event()
```

```
        if clicked:
            player_pos = game_object.get_component(Position).position

            projectile = GameObject(
                components=[
                    Position(position=player_pos),
                    engine.Sprite(image="projectile.png"),
                    Velocity(velocity=mouse_position - player_pos)
                ],
                children=[],
                name="Projectile",  # Will collide
                                    # Some engines add suffixes
                                    # In this case it won't matter
            )

            world.game_objects.append(projectile)

player = GameObject(
    components=[
        Position(position=Vector(0,0)),
        engine.Sprite(image="player.png"),
        Player(),
    ],
    children=[],
    name="Player"
)

world = World(
    game_objects = [player]
)

world.start()
```

A real game would be written differently. Especially the engine side. This ought to illustrate the programming model. A similar programming philosophy works for most of the traditional game engines with minor tweaks on which concepts get combined and how. This philosophy leans heavily on the concepts of Object-Oriented Programming (OOP), with component code holding both state and behaviour. This is one of the key differences between traditional engines and ECS. Sometimes the fact that some behaviour is intrinsic to a type of component may be undesirable, which can be problematic when dealing with premade components that come with the engine and the engine expects to be used. Wrappers and workarounds can be added to deal with this, but it can get complicated.

Unity does offer a Unity Test Framework (UTF) through the in-engine package manager (Unity Technologies, 2023). Three of the five interviewees mentioned using UTF, with some adding a custom layer on top to aid in testing UIs. UTF does offer Edit Mode tests for testing both game code and in-engine tooling within the Unity editor as well as Play Mode tests for testing games in a realistic runtime scenario. These tests can then

be run from the command line by invoking the Unity binary with specific flags. It can also integrate into Rider, the C# IDE by JetBrains. (Unity Technologies, 2023)

Another popular game engine is Unreal Engine by Epic Games, the company behind Unreal Tournament and Fortnite among others. Unreal is hard to describe, as it operates on a different level of abstraction a lot of the time. Instead of thinking about moving GameObjects, often Unreal developers are working with prebuilt player controllers. The developer simply plugs in the values like models, animations, and properties like walk speed and the controller handles things like animation blending automatically. The diagram looks quite similar:

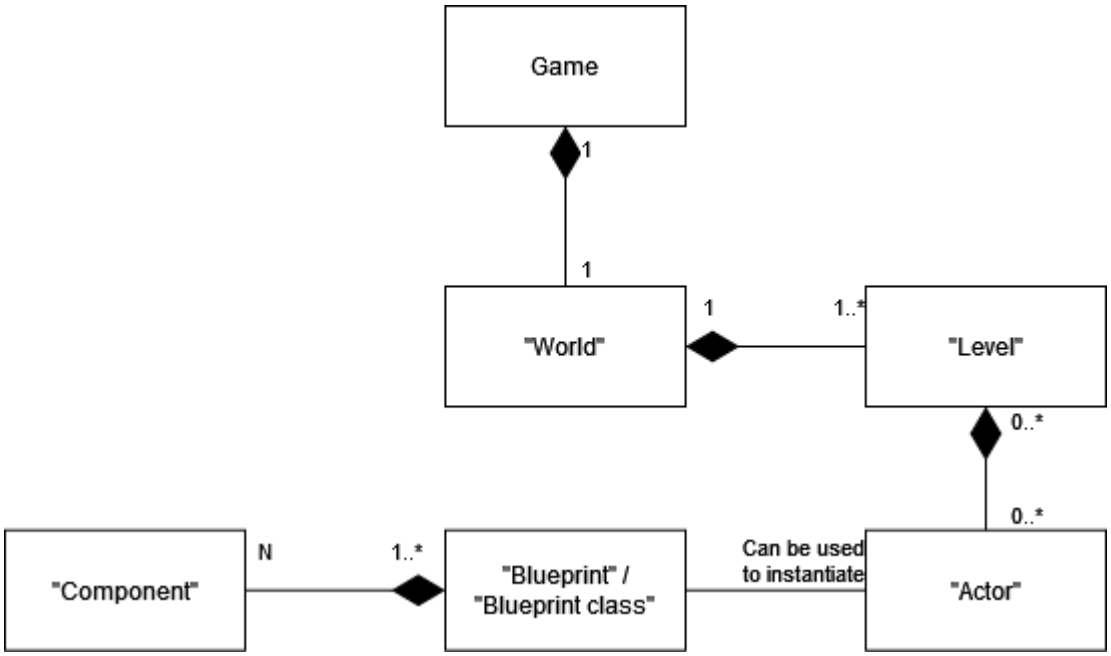

Figure 2 - Diagram of Unreal Engine concepts

In Unreal, like in Unity, the game has a world, although in Unreal the world is a term used in the engine. That world then contains levels, which is a term used similarly to scenes in Unity. These then contain Actors, which share a lot of similarities with Unity GameObjects, but this is where the paths begin to diverge. In Unity, GameObjects themselves are rather minimal, and it is the components that make them do what they do, but in Unreal, the Actor itself may be a specialised type of Actor like a Pawn, which is controllable, or even a Character, which is a Pawn that has some common game actions like walking and jumping. All of those are Blueprints or Blueprint classes. As far as my cursory understanding of Unreal goes, those are the same thing. These Blueprints come with a visual node-based scripting editor and the ability to write C++ code should you so choose. The blueprints come with several components, which are used for roughly similar purposes as the ones in Unity. The editor does guide the user to-

wards writing their code in the blueprint visual editor instead of making new components.

Unreal also offers a first-party testing solution with the Automation System. The system is capable of various levels of testing like unit and feature testing as well as less common types like stress testing. (Epic Games, 2024) Gearbox developed an in-house system called Polaris that offers a no-code testing harness in the Unreal editor, but unfortunately, it is not available for public use (Gearbox software, 2023).

In general, Unreal is a more specialised engine, but you can still see pretty much the same structure of a world containing collections of entities, with each entity consisting of components. Godot takes a slightly different approach:

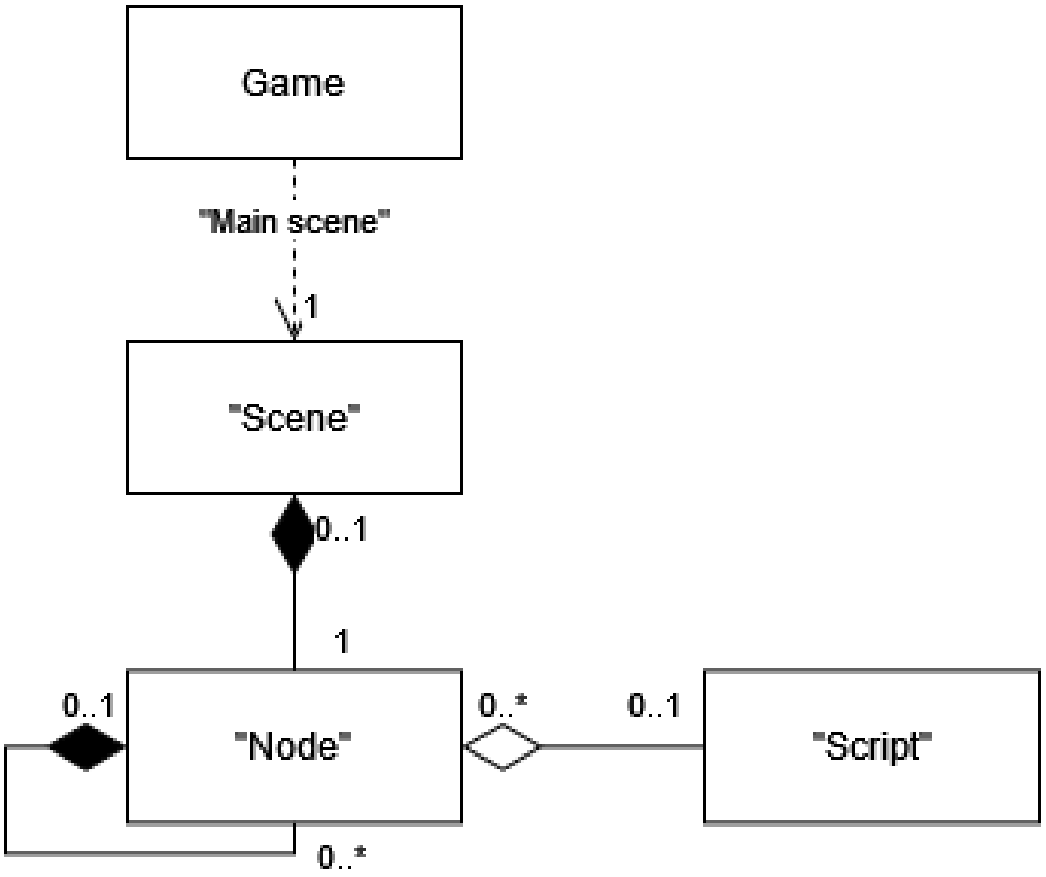

Figure 3 - Diagram of Godot concepts

Instead of Scenes, Actors, and Components, Godot uses the concept of a “Node” for all three. A scene in Godot is simply a node that has been saved to disk. One scene acts as a main scene, which is the only thing that starts up as loaded into the game world. Nodes can be nested into each other and come with different types of premade behaviours. There are nodes for rendering models and sprites as there are nodes for playing sounds. Each node can then have a script attached to it. Game developers

write these scripts to augment the base nodes. For example, a script on an audio player node can access the properties of the audio player like they were local variables in the script.

I find the Godot approach the simplest to understand and the most flexible to use. In Unity, for example, spawning in a scene and spawning a GameObject use different APIs, but in Godot, both of those use the same one. Code in Godot is written using a Python-like scripting language called gdscript with several bindings for other languages available, but similarly to C# in Unity, gdscript is the best supported option.

Godot has the Godot Unit Test (GUT) third-party framework for unit testing. It can be installed through the in-engine asset library (Wesley, 2024). It has a similar feature set to UTF, allowing for unit tests and integration tests.

Custom engines are a diverse bunch and thus it is hard to draw conclusions, but they can have the advantage of editable engine code and an understandable execution order, as the engine internals can be seen and modified. Since custom engines are all unique, it is difficult to make conclusive statements about them. Therefore they will be ignored for the rest of this thesis.

2.3 ECS

ECS stands for Entity Component System, which is an architectural pattern used mostly in games, although it could be used in other contexts, such as simulators. With the ECS pattern, the software is divided into Entities, Components, and Systems. In an idealised ECS system, entities only contain components, which only contain data. Systems statelessly act on these entities and components. See diagram:

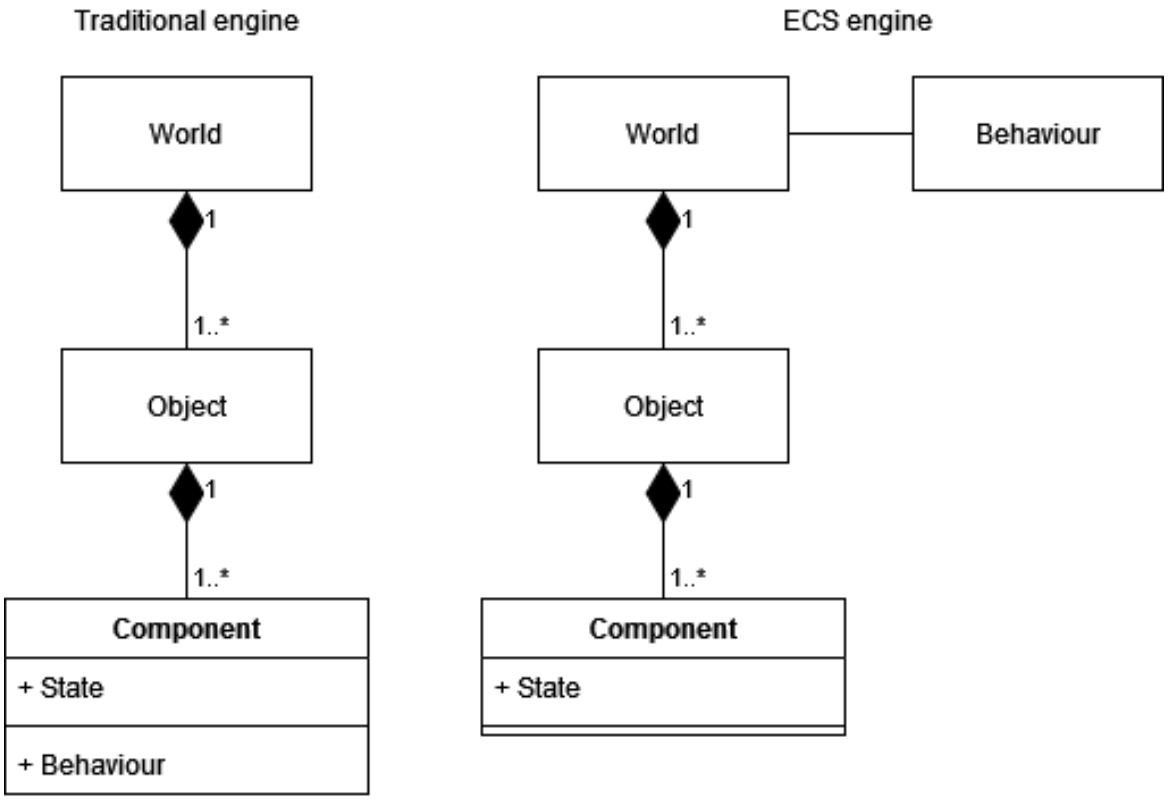

Figure 4 - Side-by-side comparison of traditional and ECS engines

One of the main benefits of this data-driven approach is transparency. As ideal systems do not maintain state, the entire state of the software can be read from the entities and components. Another crucial benefit is control over execution order, as systems are easily orderable to be executed as the developer wants. Both benefits are side effects of the decoupling of behaviour and state. In addition to those, there are often performance gains as parallelisation becomes easier as well as reasonability benefits, as inspecting the system at a certain point in time becomes simpler.

In a hierarchical inheritance-based engine like Unity or Godot, the objects in the game have a list of components, but these components contain both behaviour and state. Typically, these components are implemented as classes that inherit from other components or the base component class provided by the engine. Often these components can be reused with multiple objects in the game world, but there are cases where this can be difficult. The main source of difficulty is how components refer to other components. If one component wants to reuse something done by another component, there are some options, but all of them have undesirable side effects. One option is to refer to other components on the same object through the object, but that causes problems when the developers want to test components in isolation. Another option is to inherit the existing component, but this may bring additional undesired functionality, create

lots of hard-to-decipher layers of abstraction or break places where the first method is used. A third option is to create a God component that contains everything, so it won't need to refer to other components. The first approach tends to be the favourable one, as it leads to several smaller components, but the components are now coupled with each other, as they require state they do not hold. This may be hard to work with if the developer wants to reuse the components in other places. For example, if you are creating a game where the player can crouch, you can create a crouching component that accesses the character movement controller to adjust it when the player is crouching. The crouching component can do other things like modify animations as well. Later you also want non-player characters to crouch, but since the crouching system is coupled to the player movement component, it is not so easy to just reuse. Usually, this leads to more layers of abstraction. What complicates the matter even further is that sometimes some components come from the engine or external libraries and their code cannot be modified. The god component approach goes against traditional programming heuristics like most of the SOLID principles and does not really make reusing any easier, but it likely does cut out some boilerplate code that was accessing other components.

In an ECS architecture, if it quacks like a duck and walks like a duck, it is a duck. If you want to implement crouching, you could have one system, potentially one that handles all the input processing, toggle a crouching flag, and then have separate systems for all the changes that crouching should have. You could have a crouching speed system that queries for entities with both the crouching flag component and the player movement component, and that could work with or without a second system that handles the animation changes. You end up with more modular and typically smaller functions that must make fewer assumptions about the entities they are targeting. These systems are separated from the state, so if any entity gains the required components for one of them to run, then the system runs.

In a traditional engine, every instance of a component runs the same code with potential differences in its stored state. This means that if a component requires some preparatory work, like knowing where the player character is, it is often simpler to repeat this prep work in each component than it is to abstract it out to a centralised location. This means the same preparations are sometimes performed multiple times. In an ECS engine, the developer can run the preparations once at the start of a system and then perform all the relevant updates without duplicating this work. In certain cases, the work itself may even be parallelised trivially. As described by Shalf and Leland in Computing beyond Moore's Law, the performance of individual CPU cores has largely remained static for the past few years (Shalf & Leland, 2015). Parallel computing is one

potential method of increasing performance, but that requires a shift in how programs are written. Deeply coupled structures tend to be harder to parallelise due to their web of dependencies. Mechanisms like SIMD, which stands for single instruction, multiple data could be easier applied to cases when there are easily denotable chunks of parallelised operations (Fortin, Fleury, Lemaire, & Monagan, 2021), with Bevy often taking advantage of SIMD. This ease of parallelism is another side effect of the decoupling of state and behaviour.

Another benefit of ECS is the simplicity by which components get accessed. For example, if we have an object with components A and B. When the behaviour of component A needs to access data in component B, the flow looks something like this in a traditional engine:

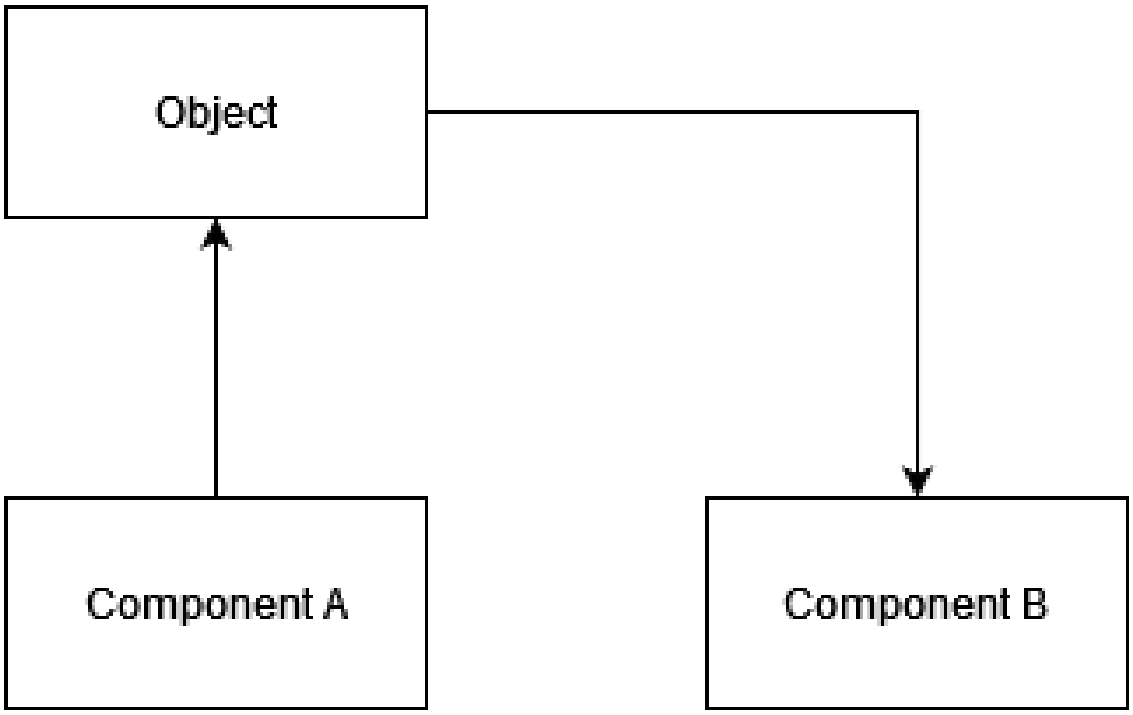

Figure 5 - Sibling component access in a traditional engine

The diagram shows an Object like a GameObject in Unity containing two components, A and B. When Component A depends on data in Component B, it must go through the Object which contains both components. Sometimes the problem cascades, with Component B depending on component C, which may also have dependencies. Sometimes the dependencies go across objects, where a component is dependent on a different object with some component. A common example is some centralised singleton used for data aggregation. When component A is being tested, there is a problem with three distinct approaches. Approach one is to mock the calls to the Object, which will require some work on maintaining the mocks and might cause problems when there are dis-

crepancies between the mocks and the production code. Approach two is to write Component A in a defensive matter, where it can adjust to not having access to the Object. This will add code that by definition never runs outside of the tests and just adds noise to the production code. The third approach is to not bother separating Component A by only testing the Object with all its components attached. Because the cascading dependencies can cross Object boundaries, this usually means testing the entire scene. This is the approach many including UTF play mode tests take since it is likely to be the least costly in terms of work hours when setting it up. It will come with maintenance costs, as tests will fail for reasons beyond what they are testing due to how it forces the developer to include large segments of the game in each test. No matter what approach is chosen, it comes with downsides that could potentially cause problems later. The fundamental reason is the coupling of data and behaviour. If the developer could include Component B's data without the behaviour, it would eliminate the cascading dependencies problems.

In an ECS engine, the model of access is different. As components are purely passive data containers, the access is done from the systems and the mechanism remains the same when accessing single components as it is when accessing any number of sibling components. Diagram:

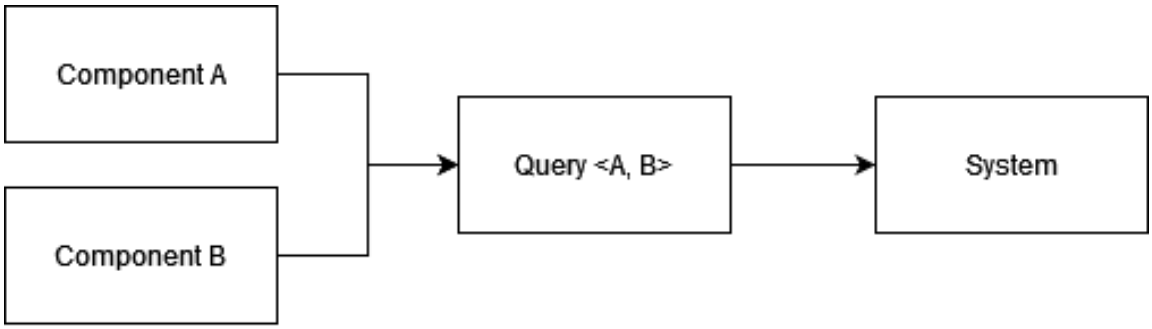

Figure 6 - Sibling component access in an ECS engine

As all behaviour happens in the systems, data access is handled through queries to the game world. Query is a term Bevy uses, but the concept remains the same in other ECS engines. A developer can use queries to request an iterator of all the entities in the game world with a certain combination of components. If no objects contain the required components, the iterator yields nothing. A common pattern is to have a system that starts by querying and then looping over the query results. If the query fails to find targets, the system does nothing, as it is iterating over an empty collection, which means that developers can restrict which systems run by limiting what queries find targets. It is not an error when a query fails to find, so having extra systems in tests is not a problem. Usually, the developer would however just limit what systems to insert into

the app in tests. Several design patterns emerge from the query mechanism. For example, developers can use so-called marker components, which contain no data but mark specific entities for some queries to filter beyond the components with data. This is a useful pattern when a query would target unintended entities without the additional marker component requirement filtering them out. The component system is flexible enough, that it can be used to model complex relations between entities. Bevy handles “nested” entities by having a Parent component on the child which holds the ID of the parent entity and a Children component on the parent which contains an array with child entity IDs. When testing, the developer can simply add the behaviour being tested and enough components so that the query will hit. Tests can be atomic and minimalistic without mocking or defensive production code.

As has already been discussed, the execution order is something many traditional engines abstract away from the developer. Usually, the exact ordering is learnable, but it is made obtuse by the abstraction. In ECS engines the developers can set the system execution order. This will greatly simplify reasoning about what updates have already been processed at some point in the code.

To implement the same example game in an ECS engine, the engine would provide something like this:

Listing 3 - ECS engine - Example engine code

```
import engine   # Same as the traditional engine.

class Component:
    # Just a parent class for data containers
    ...

# This would use the newtype pattern in Rust
class EntityId(Component):
    value: int

# These two take care of parent-child relations
class ParentId(Component):
    value: EntityId

class Children(Component):
    ids: list[EntityId]

# If you want to have named components,
# you could add something like this.
# Unused in this example.
class Name(Component):
    value: str

class Vector:
    x: float
    y: float
```

```python
# Components required by the engine get implemented in the engine.
# Behaviour can be trivially expanded through systems.
class Position(Component):
    position: Vector


class World:
    entities: list[list[Component]]
    systems: list[callable]

    # Used for giving entities unique IDs
    running_id: int

    def query(self, types: list[type]) -> list[list[Component]]:
        output = []

        for entity in self.entities:
            item = []
            for component_type in types:
                for component in entity:
                    if isinstance(component, component_type):
                        item.append(component)

            if len(item) == len(types):
                # Found all components
                output.append(item)

        return output

    def _find_by_id(
            self,
            entity_id: EntityId,
        ) -> list[Component] | None:
        # Only really used internally, not in the game code
        for entity in self.entities:
            for component in entity:
                if component == entity_id:
                    return entity
        return None # Can fail to find.

    def spawn(self, components: list[Component]):
        components.append(EntityId(self.running_id))
        self.running_id += 1

        self.entities.append(components)

        return components

    def spawn_as_child(
            self,
            parent_id: EntityId,
            components: list[Component],
        ):
        parent = self._find_by_id(parent_id)

        if parent is None:
            # This can happen if something gets despawned for example
            raise Exception(
                f"Failed to find parent {parent_id.value}"
            )
        for component in parent:
```

```
            if isinstance(component, Children):
                component.ids.append(EntityId(self.running_id))

        components.append(ParentId(value=parent_id))
        self.spawn(components)


    def run(self):
        # Primary loop of the game
        while True:
            # Normally, there would be a way to exit gracefully
            # Note the order of execution here:
            # System by system, in the order they are defined in
            for system in self.systems:
                system(self)
```

Many parts of this code are simplified for the sake of giving a concise example. The game-developer-facing interface would be like one offered by a real ECS engine, but a real engine would add several optimisations that would clutter this example. A common one is maintaining archetypes, which allow for pre-calculating the iterators returned by the query function. These iterators get updated every time a component or an entity is added or removed, but it is usually faster to do this than it is to scan all the entities for each query, which is what the example code is doing. The archetype can be constructed after the first query of a certain type or before the game starts. There is a slight performance and flexibility trade-off between those two methods. Bevy essentially constructs archetypes at compile time. Archetypes do make certain actions like multiple simultaneous queries with some of them being mutable more complex for the end user, but ECS engines typically offer workarounds that circumvent some of these problems. While they can have a steep learning curve, archetypes yield significant performance gains.

Under some circumstances, two systems can safely run in parallel. When this is applicable depends on how they access the world. Some ECS engines parallelise these systems based on what components they access, with some even allowing for parallelisation within the systems under certain conditions. Bevy distinguishes between mutable and immutable access, which it uses to parallelise even some systems that do query for the same components, but since both get read-only access, there is no possibility of a race condition.

For the rest of the ECS game example, the game developer would then add this:

Listing 4 - ECS engine - Example game code

```
class Velocity(Component):
    velocity: Vector
```

```
class Player(Component):
    # Marker component that's used to identify the player
    # Contains no data, but is useful for querying
    pass


def player_input(world: World):
    clicked, mouse_position = engine.get_click_event()

    if clicked: # Nothing is queried if there is no input
        players = world.query([Position, Player])
        assert len(players) == 1    # There is only one player
        player_pos = players[0][0]

        world.spawn([
            player_pos,
            Velocity(velocity=mouse_position - player_pos.position),
        ])

def handle_velocity(world: World):
    for mover in world.query([Position, Velocity]):
        # This will only run if the query finds something.
        # As each loop only acts on a single entity,
        # this could be iterated in parallel.
        position, velocity = mover
        position.x += velocity.velocity.x
        position.y += velocity.velocity.y


player = [
    Position(position=Vector(0,0)),
    engine.Sprite(image="player.png"),
    Player(),
]

world = World(
    entities=[player],
    systems=[
        # Systems could get imported from other modules.
        # They run in the order defined here.
        player_input,
        handle_velocity,
        engine.render_sprites
    ]
)

world.run()
```

It is a personal preference, but I find the ECS method simpler, as determining what has already been executed is trivial and there are no potential problems with existing behaviour when using a component that was provided by the engine. If a game developer is dissatisfied with how the engine handles rendering sprites, they may simply replace that system with their implementation, and it will not impact how the sprites get used by the rest of the game code.

As it comes to testing, ECS is by nature transparent. The initial state of the system can be trivially set, and assertions can be made easily as the entire state of the game is visible. Test code can run alongside application code with the application code unaware of the test code. The low degree of coupling will allow for easier isolation of subsets of systems of the game. A scenario can be constructed for a test in a way where the production code elements are unaware of being run in a test. Where coupling is necessary it can be easily detected by looking at which systems access the same components.

2.4 Bevy

Bevy advertises itself as “A refreshingly simple data-driven game engine built in Rust” (Bevy project maintainers, 2024), It is an ECS engine with a focus on user ergonomics, which means it offers several features to aid development that are not necessarily part of the basic ECS architecture. Diagram of the data model:

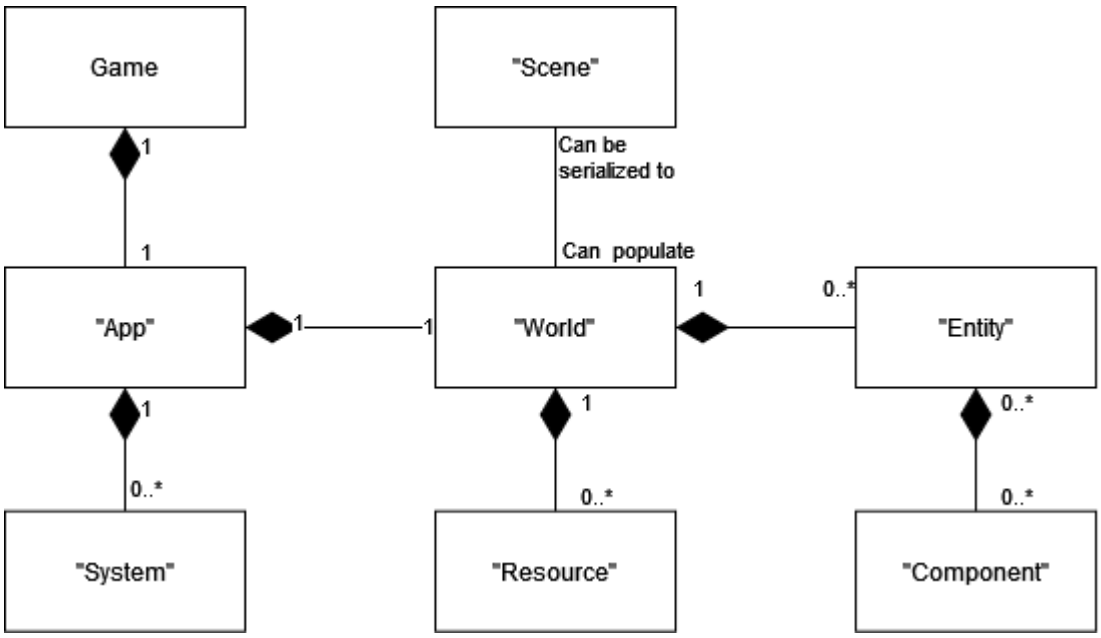

Figure 7 - Diagram of Bevy concepts

A Bevy game starts by creating an App, which contains the systems that are implemented as Rust functions as well as the game world. The world can be serialised to and deserialised from scenes and contains entities which in turn contain components. The world also contains resources, which can be thought of as global singleton components. You could model the same data with a single entity that holds all the global components, but Bevy provides the resource abstraction so that when a developer is reading code, it is apparent to them when something is a global singleton. Often enti-

ties are created via systems, which can be configured to run in a variety of schedules, which define when a system runs. Some schedules only run when the game is starting up, some in the main game loop, and some in a fixed update loop.

Constructing the application is a large part of the work. Bevy uses the builder pattern for the app construction, but sometimes it is helpful to split off chunks of the construction process. Bevy offers the abstraction of Plugins, which can be used to package elements into clear units. Plugins as a concept only exist when the app is being created and related plugins can be stored as PluginGroups. One of those that is provided out of the box is DefaultPlugins. It adds the necessary elements for running the schedules, opening windows, playing audio, skeletal animations, rendering, reading user input and so on. This means that if a developer wants to simply use the ECS framework of Bevy without most of the things associated with game engines, they can do so simply by not including DefaultPlugins.

Games made with Bevy can be built for Windows, Linux, Mac, WASM, iOS, and Android. It is still being actively worked on, so some features like a dedicated visual editor are not available yet, but it is more than likely that will be done by the time you read this. Many third-party options exist to fill the gap in the meantime. The ecosystem around the engine has a wide array of useful tools, although due to lack of first-party support, they are occasionally not as high in quality as the core engine. Some standouts include:

- Rapier, a physics engine suitable for 2d and 3d.
- Bevy egui inspector, a GUI that lets the developer observe and edit component values while the game is running. Similar to how one can edit component values in the Unity editor while the game is running.
- Bevy mod picking, a lightweight method to add picking functionality like drag and drop.
- Leafwing input manager, an easy-to-use abstraction layer between controller inputs and game actions

Bevy is still quite new and needs more work to compete with the traditional engines in the market, but the core of the project is solid. One of the best things about it is how well It integrates with Rust’s type system.

2.5 Rust

Syntactically, Rust looks like any of the C-style languages with curly braces to signal where blocks start and end, the main function as the entry point to the program and familiar constructs like traditional conditionals and loops. There are however several aspects of Rust that make it unique in the programming language space. The most relevant ones here are the trait system, the type system, and the borrow checker.

Rust has no inheritance, forcing developers to rely on composition and the trait system. A trait is like a typeclass in Haskell or an interface in most other languages. They denote a set of features that a type must implement to implement the trait itself. Often you can use a derive macro to automatically generate trivial trait implementations. Macros in Rust are quite different from the typical string substitution that goes on in languages like C, but that is not relevant to the thesis. Bevy uses traits to annotate assets, resources, and components, as that makes it easy to know quickly what a type is meant to represent when you encounter one in the code.

Rust has two ways of composing data structures, enums and structs. Structs work similarly to languages like C with minor differences in how their fields get stored in memory. Enums in Rust can be a collection of variants like languages like Java and Python, but similarly to many functional languages like Haskell data declarations, the variants may also contain values. This lets the developer model the state of their application to make certain states unrepresentable. Take for example the Option<T> type, where T is a generic type. It is an enum with two variants, either Some(T) or None. Much like the Maybe type in Haskell, it can either contain a value or not. This is how Rust handles cases where most imperative languages would return a null. There is no null in Rust. This means that in typical Rust code, the concept of a null pointer is unrepresentable. Whenever a developer wants to access the value contained in a collection and it may or may not exist, they get an Option<T> or another similar type. If they want to access the data within Option<T>, they must explicitly handle the container. This same mechanism is used for operations that may fail. Instead of raising an exception, a Rust function returns a Result<T, E> with two generic types T and E. Result<T, E> has two variants Ok(T) and Err(E). The lack of exceptions that traverse up the call stack simplifies control flow and prevents a huge number of bugs, as it forces the developer to be aware of situations where an operation might fail. The feeling of knowing that the compiler will remind you if you forget about some edge case where a function will not return a value is hard to describe as anything but reassuring. This I feel is one of the main reasons Rust is well suited for game development, as game developers often work with

complicated code filled with corner cases, so it would be nice to have a compiler that ensures all of them are handled.

The borrow checker is where most problems start for new Rust developers. Instead of using pointers, Rust has a different mental model that pertains to both memory management and access. According to Microsoft researchers, 70% of the security related bugs the company fixes each year relate to memory safety (Perkel, 2020). Use after free, race conditions, and unintentional mutation among others are not only causing security problems, but other downsides like crashes and overly defensive code. Rust solves these problems with the concepts of ownership and borrowing. By default, when a variable is passed to a function, ownership of the value is passed alongside it. After ownership is forfeited, the caller may no longer use that variable. The compiler simply refuses to compile the program. Instead, to retain ownership, a developer can pass a borrowed value, which lets the called function read the variable, but not mutate it, as by default variables are immutable. The caller remains the owner of the value and can keep using it afterwards. The signature of the function states whether it accepts mutable or immutable borrows or only owned values. The part that trips most new Rust users up is how you can only have either one mutable borrow or any number of immutable borrows, but never both mutable and immutable borrows out at the same time. This prevents some data races.

As demonstrated by Will Crichton in their article “The Usability of Ownership”, there is a set of programs that are correct and memory safe, but the compiler cannot prove them to be (Crichton, 2020). A common example is any graph where nodes reference each other. For these cases, the language provides escape hatches like internally mutable data types and the unsafe block, in which you can perform some operations you cannot otherwise. Mechanisms like this let you deal with the cases where the borrow checker gets hard to work with, making Rust opinionated, philosophy-driven, and yet practical for real-world use.

Games are performance-critical and complex. A performant language that protects against common mistakes is thus a good fit. The type system and borrow checker work together to eliminate multiple categories of bugs. While the ecosystem is not nearly as mature as with something like C++, the growth in recent years is promising. Even the white house sent out a press release nudging the industry towards memory-safe programming languages, which mentions Rust by name (White House Press Releases, 2024). The value of learning from the previous generation of tools is also something to acknowledge, although measuring the impact of that on the quality of Rust tools is at the very least difficult.

The strict compiler can be thought of as an automated testing tool. Some types of bugs from dynamically typed languages are impossible to represent, as Rust is statically typed. This makes Rust code relatively safe with very little testing. No tool can prevent a mismatch between implementation and the intended design, but Rust does a pretty good job of preventing discrepancies between the intended implementation and the written implementation.

2.6 User interface testing

As stated in A deep learning-based automated framework for functional User Interface testing, “Carrying functional tests at the UI level is a laborious, manual and tedious process. It often consumes hefty resources like time and cost as it requires prerequisite technical concepts.” (Khaliq, 2022) There are tools like Selenium and Cypress that allow for regression testing, but they are laborious to write and only prevent errors that have already been detected prior.

There are two main approaches to making tests that involve the UI, such as integration or end-to-end (e2e) tests. Test cases can be written as code, which is then executed by a test runner that performs the described operations and then makes assertions about the state of the program. The other method is called capture and replay by the team behind Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing (Di Martino, 2021). With capture and replay testing, user input is recorded and then played back to the program with assertions at the end. There is also the third option of automatically generated pseudo-realistic user input, but Di Martiano et al found that students with some familiarity with the code being tested outperform those mechanisms (Di Martino, 2021). The first two approaches can be seen as opposing views about what makes UI testing difficult. Manually written tests optimise for maintenance and tend to be more robust since they use more generic mechanisms for finding UI elements. Capture and replay tests optimise for test creation and are quick to generate but will often break when the UI changes. For example, if a button is moved, capture and replay tests will likely need to be re-recorded, while programmatic tests could find the button at the new location. This is not to say the manual tests do not break, they do. They are just slightly more resilient to change. In both cases, some work will have to go towards maintaining the tests as the designs change.

When it comes to games, popular traditional engines do offer solutions for writing procedural e2e tests, but often they need more domain-specific tweaks depending on the game. Games are diverse and it is difficult to create a tool that accounts for all of them. This means that often the game developer must tailor the testing tool to suit their specific game. Capture and replay tests require very little support code. They operate on such a low level of abstraction they can get away with just interacting with the engine features. They however cannot be applied to many types of games. Because all the inputs are generated ahead of time, doing any significant randomness makes them unviable for example.

Several attempts at automating test generation using deep learning-based AI solutions have been made in recent years with some of them yielding impressive results. For example, the team behind A deep learning-based automated framework for functional User Interface testing claims with tests generated using AI achieved single-digit percentage of flaking under most of the applications they tested:

Table 15
Percentage of Passing and Failing tests when modifications are injected into the AUTs.

Category	Type of modification	GroceryStore-with-server		Souq_ShopOnline		shoearstore		E-Commerce-Android		Ecommerce-Android-App	
		Passed	Flaky	Passed	Flaky	Passed	Flaky	Passed	Flaky	Passed	Flaky
Layout	Addition	98.7	1.3	98.3	1.7	97.9	2.1	96.9	3.1	94.8	5.2
	Deletion	94.3	5.7	96.8	3.2	99.2	0.8	93.7	6.3	96.5	3.5
	Location Change	100	0	100	0	99.8	0.2	100	0	99.9	0.1
Visual	Colour	99.9	0.1	99.7	0.3	99.9	0.1	98.9	1.1	99.9	0.1
	Size	97.7	2.3	96.9	3.1	98.5	1.5	99.1	0.9	99.8	0.2
	Dark Mode	92.9	7.1	88.7	11.3	91.2	8.8	94.3	5.7	86.9	13.1
Text	Size	100	0	100	0	99.8	0.2	99.9	0.1	100	0
	Colour	99.8	0.2	99.9	0.1	99.9	0.1	100	0	99.9	0.1
	Font	100	0	100	0	99.8	0.2	99.9	0.1	100	0
	String	100	0	100	0	99.9	0.1	100	0	100	0

Figure 8 - Accuracy Percentage of Passing and Failing tests when modifications are inserted into the AUTs (Khaliq, 2022)

This is impressive but raises several questions. There is no measurement of the usefulness of the tests, which is hard to estimate numerically. There are likely flaky tests that did not get caught by the mutations, as some flaking may require precise changes. Under most circumstances, even a few percentages of flaky tests make the generator unusable. This will likely improve, but whether it gets to a good level or sees any use in games remains to be seen.

3. THE PROBLEM

In short, the problem is that integration testing in game development, specifically Bevy development is sparse. Perhaps the discrepancy in automated testing between game development and traditional software development is caused by a lack of good tools on the game development side.

Additional automated testing would be valuable, as it could catch bugs and free up developer time, increasing the value of games and the efficiency of development resources. This is valuable to both producers and consumers of games. Perhaps the lessons learned here can be generalised to work with other engines, or maybe other developers get inspired by BITT to build equivalent tools for other engines. The challenge is that the existing tooling for testing, especially with Bevy requires a lot of manual work to set up and has cases where it is clumsy to apply.

4. RESEARCH METHODS

Three methods of research were applied for this thesis. Review of the existing literature in the games testing space, screening interviews of Finnish game developers, and a quantitative survey of the Bevy community. The primary goal of interviewing game developers was to supplement the sparse research about automated testing in games with up-to-date industry knowledge. The objective of the survey was twofold. It provides data to validate whether the hypothesis about automated testing in games lacking due to poor tooling holds. It will also validate if BITT, the testing toolkit developed alongside this thesis, alleviates the problem in a meaningful way. Agile methodologies were employed in the development of BITT with occasional feedback from fellow Bevy users.

5. INTERVIEWS

To fill the gaps in the literature, several game developers from five Finnish Companies were interviewed. Potential candidates were sourced from the Neogames 2022 Finnish game industry report (Neogames Finland Association, 2024). 54 of them were emailed asking for an interview with five of them agreeing on one. One was performed live on the TUNI campus, while the other four were conducted remotely via Google Meet. The interviewed people had titles like CTO, developer, and QA specialist, all of whom appeared to have a solid grasp on what is happening in the industry.

These interviews illuminated some trends in the industry. There were not enough participants to draw conclusions about the industry, but there were commonalities in the views held by the interviewees. These interviews were semi-structured, meaning there was a script but for the most part, the conversation flowed into a variety of interesting tangentially related topics. Visualising the data is difficult due to the open nature of the conversation and the flexible answers. For example, about half of the companies did frequent unit testing. The other half left unit testing up to individual developers who occasionally do it when they feel like it. There were topics on the script that mostly applied to certain types of games, so if an earlier answer clarified that the studio was working on exclusively single-player games, the follow-up topics like server infrastructure were skipped. The list of questions went over several iterations, but the final one looked like this:

- What is your personal relevance to the topic
    - Current/past roles and companies
    - Personal interest
    - Education
- Do you have a QA department
    - Is it in-house or external
    - Approximate size as a percentage of total staff
- Do you do automated QA

    - Unit testing
    - Integration / end-to-end testing
    - Infrastructure testing
- Techniques used
    - Linter
    - CI pipeline
    - Automated actor in a sandbox
    - Rust
    - Docker
    - Terraform
- Future predictions
    - Recent/upcoming changes in tech
    - Market split between Games as a Service (GaaS) and hypercasual
    - AI impact
        - OCR or similar well understood techniques for UI testing
        - Using fully autonomous AI agents
- Is all this representative of the industry as a whole

Some interesting findings that came up in the interviews:

- Unit testing happens to some extent at nearly every company, but most responders felt it was not that useful nor sufficiently systematic.
- Every company had or at some point had a CI pipeline that built the game, with most pipelines also running some tests.
- Static analysis like linting and formatting was mostly left up to the whims of individual developers.
- Of the companies that had QA departments, a typical size was roughly 10% of staff. Smaller companies sometimes had a person occasionally doing in-house QA, often a smaller percentage than the companies with dedicated departments. Sometimes this person was hired to work part-time.

- Everyone occasionally leverages external QA with nobody able to give me solid numbers on how much. Platform QA was mentioned several times.
- Only one of the companies used Rust. Some did find it potentially interesting. Some considered it an interesting toy or a fad.
- Everyone agreed that going forward, the amount of automated testing will increase in the games industry.
- Most were sceptical but curious about the benefits of AI in testing. The ability for AI to play the full game was a notable upside, while potential difficulty in getting usable reports from AI was a downside. AI hallucinations were a concern.
- Unity play mode tests were a common method of integration testing. Some had similar systems with custom engines.
- Every company belonged to one of two groups. Either they found testing the UI difficult, or they had a custom tool for doing that. Most of these tools used existing UI frameworks, enabling easy programmable state transitions. Some concerns for UI testing were the flakiness of running the whole software and the frailty of changes in design.
- Many believed strongly in the 80/20 rule, where 80% of the results come with the first 20% of effort and vice versa. This was used as a justification for what the interviewees felt was inadequate testing.
- In areas closer to traditional software development like backend web servers, testing and QA practices were more advanced and resembled traditional software development.
- Certain types of features made game testing more difficult. An example given was physics engines, which tend to have small deviations in timing leading to large differences in behaviour.

In addition to unit tests, there were several more unique practices for ensuring code quality. Some of these indicated more about the culture of the developers than the testing needs of the games. Some highlights include:

- An autonomous network of bots, some with scripts and some just randomly doing inputs, running in a sandbox around the clock.
    - Scripted actors made sure certain paths still worked.
    - Random input was meant to find crashes.

    - All actors measured performance.
- Recording video of people playing and replaying that for a motion input game
- Centralised QA department in addition to QA experts in teams with the main responsibility of optimising the processes and improving the tools of the QA experts in game teams.
- CI server in the office that played a sound of glass breaking when a build failed.
- Robot framework was used by one company.

Reasons given for why testing felt inadequate:

- Design changes frequently and testing slows down iteration.
- Insufficient resources.
- It is difficult.
- It is time consuming.

To clarify, insufficient resources are separate from testing being difficult or time consuming because in some interviews it became clear that even if the resources were provided, the workers themselves felt that the point of diminishing returns had been hit.

There was a clear improvement over the interviews on my end in terms of active listening and asking follow-up questions. This unfortunately means the first few were likely not as well done as they could have been. I welcome anyone to do a follow-up study. Potential topics can be found in the assorted list of interesting tangents from the interviews below:

- Quality of life of QA engineers should be considered but tends to not be.
- Good tooling will provide the same quality benefits as testing but also speed up the iteration loop instead of slowing it down. Instead of waiting for a CI server to test, the tests should happen when the changes happen directly in the editor.
    - This was referring to things like simple validations that run automatically when developing the game.
- The rise of platforms like Roblox will change the industry. Remains to be seen how, but many interviewees brought Roblox up without prompting.

    - As of writing, Roblox offers little to no tools for any QA besides manual testing.

- Recent developments with big providers like Unity are causing some instability in the industry. One interviewee speculated that going forward, more people will be building custom engines to not rely on the likes of Unity.
    - Dependence on Unity was not as much about the core engine but surrounding tools like advertisement, analytics, and SaaS servers offered by Unity.
- Because in some environments, the iteration cycle contains a mandatory approval process, for example by console manufacturers or Apple/Google for their respective mobile stores, developers engineer their apps in a way that circumvents this by fetching data after launching instead of during installation.
- In some spaces, specifically mobile, a split between long-term supported Games as a Service (GaaS) and quick-to-produce hypercasual games leads to some differences in testing. Of the interviewees who indicated the direction of their companies, all were heading towards GaaS.
    - GaaS requires the software to continue working and thus values QA.
    - Hypercasual games require streamlined production and thus value tools.
- A lot of legacy follows the experts. How things were done at previous companies was a recurring justification for certain answers. For example, unit testing was usually pushed by backend developers from traditional software development.

6. BITT - THE ARTEFACT

As a part of this thesis, I have created BITT, Bevy Integration Testing Toolkit, a Rust library that aids various integration testing approaches. They work as separate tools in a toolbox and the end user can use as few or as many of them as they choose.

6.1 Other testing solutions in Bevy

There are essentially three approaches to testing games in Bevy, BITT notwithstanding. Approach 1 is to use the built-in Rust test system and test the Bevy code like it was any other Rust code. This approach is somewhat limited due to how coupled systems are to the core ECS engine of Bevy. Approach 2 is what Bevy recommends, which still uses the Rust test system. In this approach, the developer creates a simplified game with the systems that are being tested. They then run this game for a few update cycles and finally query the world to run assertions against the state of the application. (Bevy project maintainers, 2024) This follows the typical Assemble Act Assert (AAA) test structure and works well for cases where the interactions of a handful of systems need to be tested. BITT could probably add a helper to make setting up this type of test easier, but it is not too bad as it stands. Approach 3 is to write a custom test system which handles interfacing with Bevy (Nauseam, 2022) and (jroddev, 2022). BITT aims to be a universal replacement for Approach 3 as well as implement some of the features from UTF, GUT, Unreal Automation system and Polaris. Approach 3 does offer the benefit of tailoring the test gear to the game, which BITT cannot do.

6.2 Requirements and justifications for BITT

BITT was made to solve the problem of inadequate test tooling for Bevy. There are types of tests, such as end-to-end or integration tests that are hard to implement currently in Bevy, which is where BITT aims to help.

BITT itself ought to be easy to use. Considering the findings in the surveys and my personal experiences, projects like this live or die based on how well they fulfil their non-functional requirements. In this case, those would be ease of use, robustness, and

performance. Testing for many users is a chore and not something they are passionate about, so the process must be as easy as possible to foster a userbase. A core part of a pleasant user experience for a testing tool is that it works reliably and robustly. Some problems with robustness are unavoidable, but at the very least test results should be reproducible. Finally, for non-functional requirements, performance is key. Games are performance-critical, and the tooling must not interfere with this. The value of BITT will be greatly enhanced by the ability to run it on a CI runner, as this would allow for automated validations in the CI pipeline. Ideally, BITT should be able to run tests on low-power machines like those offered by GitHub Actions for free.

The functional requirements are all focused on enabling developers to make the tests they think about reality. Every test a developer can think of should be able to be implemented with Bevy with help from BITT.

6.3 Input playback

Input playback implements a simple capture and replay system and serves as the main tool in BITT. Given a generic game loop like so:

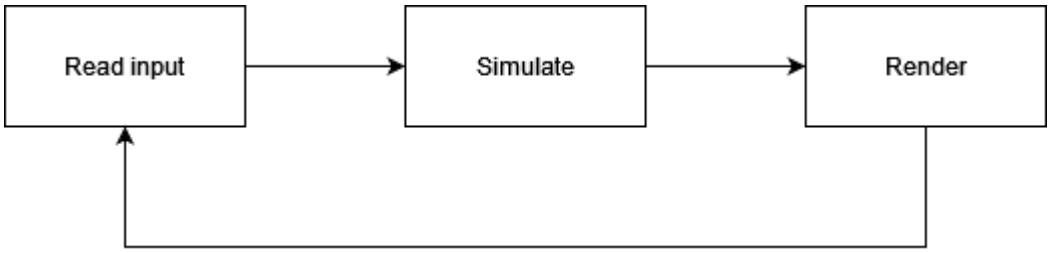

Figure 9 - Basic game loop

The most natural way to test most of the game would be to run a similar loop with most of the game code unaware it is being tested. In input playback testing mode, BITT modifies this loop by adding an extra phase like so:

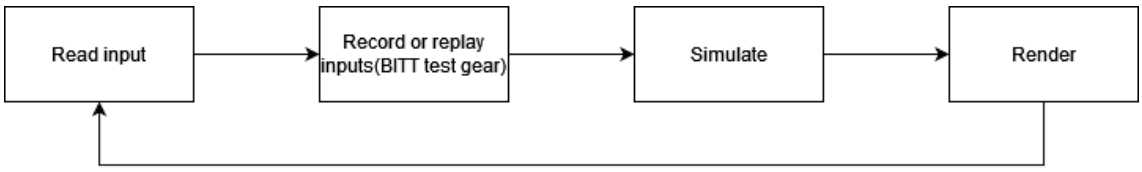

Figure 10 - Basic game loop with BITT test gear

This is done through the test gear plugin the BITT user adds to their Bevy app alongside a system of their making that utilises an Asserter resource. This resource gets

added to the app by the plugin and is used to mark the test as passed or failed under whatever conditions the user defines it should.

The test gear plugin will run in one of two modes, recording or playback. When initialising, it will look for a test script in a predetermined path. If one is found, playback mode is selected, if not recording mode is used.

In recording mode, the plugin will collect inputs while the game is running, and when the Asserter is marked as passed, concludes that the desired state is reached. It then writes the stored inputs to a script to be used by the playback system on subsequent runs. There is an option to automatically fail the test if the script cannot be found to prevent accidentally waiting for input in a CI pipeline.

If a script is detected at startup, playback mode is used. The inputs stored in the script will be played back. After the recording ends, the system waits for a small grace period of a few seconds to see if the test gets marked as passed or failed. If it does not get a resolution, it considers the test a failure. If a test fails, the program will exit with a non-zero code, causing CI pipelines to fail. Artefacts like screenshots and performance metrics are saved on disk, which lets the user know why the test failed. When running locally, the game window opens like usual so the user can see what went wrong, but there are some cases where due to differences in the CI runners, a test will only fail in the CI pipeline.

6.4 Headless default plugins and timeout

In some cases, it is helpful to run the game without creating a window. Bevy comes with MinimalPlugins, a PluginGroup that can technically run the game. However, this will offer different behaviour than the DefaultPlugins. For this, I created HeadlessDefaultPlugins, a version of the standard DefaultPlugins that simply disables the plugin responsible for creating the window and mocks its functionalities so that the other plugins run normally.

There is also a simple Plugin that adds the Asserter similarly to input playback. When added by itself like this, instead of replaying inputs, it waits for a predetermined amount of time. If the Asserter has not been passed by the end, fails the test. This can be useful for cases when an automated agent is playing the game, or to ensure some event happens within a timeframe when the player does nothing.

6.5 Applying it to my own game

One of the initial motivations for creating BITT was that I could apply it to my projects. Doing that was a mixed bag with both successes and failures. The setup process was not too bad. It takes a fair number of steps to get everything set up, but this is something that can be improved. Initially, I tried to apply several mechanisms that would let me write several tests. After a few attempts, I found the process quite cumbersome and gave up, satisfied with a single sanity check test that does a broad sweep of the game. During the attempts, I discovered the same problems that the interviewees were bringing up with flaking and maintenance problems. Changes in game design, such as number tweaks and object placements will cause tests to fail. It is difficult to test just the implementations and not the design when the test is running the entire application.

The game in question is a player-versus-player one-versus-one fighting game, meaning it is fought over several rounds of combat. Diagram of the game flow under normal circumstances:

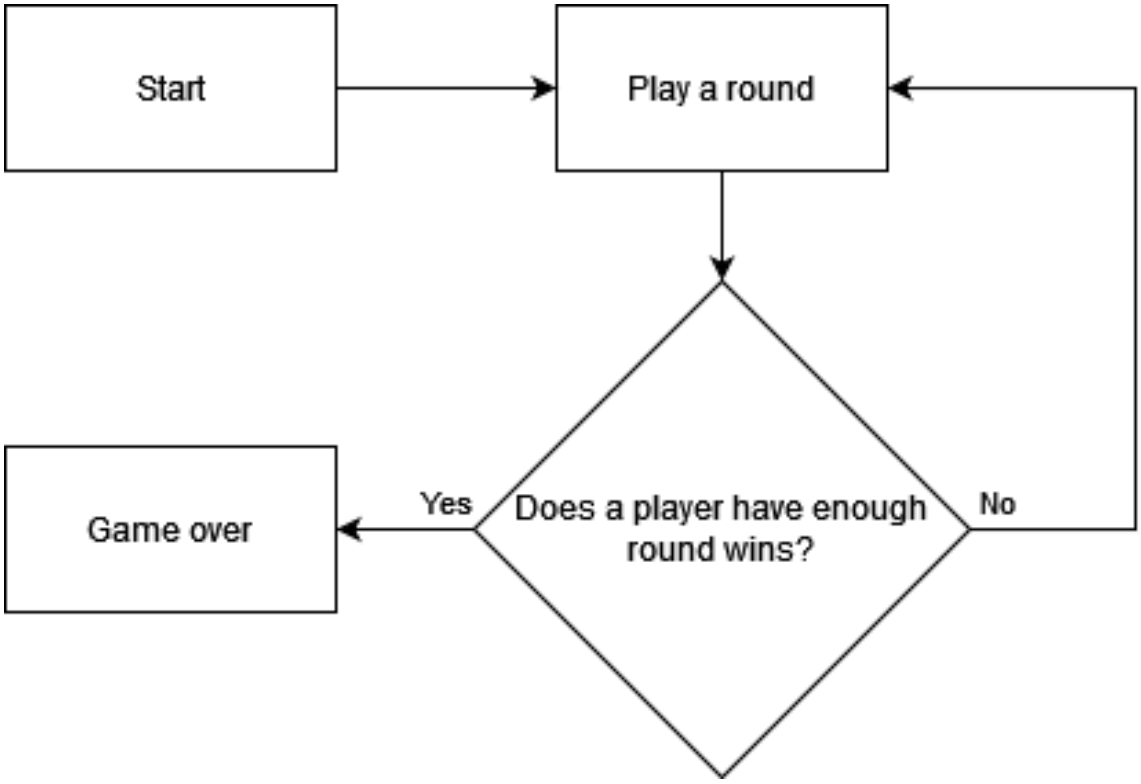

Figure 11 - High-level gameplay flow in the game under test

The test starts the game, plays through a round, and checks that the next round starts. This makes sure that:

- Game starts
- Player input reading works

- Many systems like damage and player character movement work
- The high-level game loop between rounds works

There are lots of untested cases, but setting this up was quick and easy and is likely to catch some of the most obvious bugs, like if no input gets registered or a bug prevents the game from starting at all. One of the interviewees told an anecdote where a sizeable portion of the bug reports ended when the CI pipeline was made to start each level of the game to ensure the levels could start. Ultimately, this seems like a useful compromise between some coverage and effort spent on testing.

I encountered several problems while applying BITT to the project. Bevy offers several tools for tasks like input parsing to cover a variety of use cases. The game project I applied BITT to uses a different way of parsing inputs than the one assumed by BITT, which meant BITT did not replay correctly initially. Issues like this are likely plentiful, as I do not fully comprehend the internals of Bevy, so mocking the input is not foolproof. If the playback framerate is too low, either Bevy or BITT can drop inputs. This is automatically a problem for 3D games that want to run checks in CI, as even the Windows CI runners offered by GitHub struggled hard as you can see on the table of average frame times in milliseconds:

	2D test 1	2D test 2	2D test 3	3D game
My PC	7.4884143ms	7.3784404ms	8.163867ms	7.0022016ms
GitHub CI	24.592295ms	22.906557ms	29.193766ms	839.86664ms

Figure 12 - Average frame times table

Frame time is the amount of time it takes to process each frame and is the inverse of framerate, which is the frequency of frames presented to the user. To maintain the industry standard of 60 frames per second, the game would have to have a frame time of less than 1s/60, or about 16.6667 milliseconds. All the tests were executed on a Windows machine. Tests 1 and 2 are running the same game with differing inputs. We can see that the test runner performs worse on every test, but the difference becomes significantly worse on the test that requires 3D rendering, where the average frame takes almost a second to process. This is likely due to the CI runner not having a dedicated GPU. Paid GitHub plans would allow for more powerful machines, but that is not accessible to everyone and should not be a requirement for using BITT. BITT does offer a headless mode, where no window is opened. Some forms of input work fine, like key presses and controller input, but others like mouse movements unfortunately do not. It

seems to perform very well in CI with performance very similar to that of my local machine, but the input restrictions make it unviable in certain types of games. The CI runner also didn't play audio, which isn't a massive issue, but it changed how certain systems worked in the game. The audio tracks never played so they were never marked as concluded, which leads to side effects in cases where a system assumes that an audio clip will eventually end.

Another problem I discovered was that even local multiplayer was quite difficult to test, as it required simultaneous use of multiple controllers. This seems like a problem that could be solved, for example by allowing multiple recordings to be combined. Online multiplayer could also be solved using similar means, but as the performance is a problem already with simple games, adding more services will likely just make it worse.

In conclusion, I find BITT does enable some types of tests but is cumbersome to use and does not apply to many types of games. Failure to fulfil the non-functional requirements is the final nail in the coffin. It isn't particularly easy to use and due to the performance problems, it is only reliable when running locally. Running the whole game in a low-power CI environment seems like an impractical solution for now for most projects.

7. DEVELOPMENT OF THE ARTEFACT

7.1 Initial attempts and roadblocks

I've had several attempts to test with Bevy before BITT. All of them failed due to different reasons. The first approach was to fake the environment. The second approach was to create a wrapper for the Bevy app. You could describe the approach I took with BITT as integrating within the game and so far, it seems to be working worlds better than its predecessors.

The fake environment approach had to do with what I falsely perceived to be the problem. Bevy apps typically include a set of standard plugins to their app. These will add the required systems, resources, and entities to run typical game functions like playing audio, rendering graphics, and reading player input. I perceived the plugin responsible for opening a window to be a problem, as it prevented using Rust's default test runner. The windowing plugin does not work on non-main threads and the built-in test runner runs all the tests on secondary threads. Since many of the default plugins depend on each other, removing the windowing plugin to run tests on a headless CI runner caused a cascade of problems with other plugins that assumed some effects from the windowing plugin to be present. Naturally, my response was to try to plug the holes and create a fake environment that looked convincing enough for no plugin to fail, but this turned out to be a fool's errand. This was also the first contact with how creating windows is a lot more complicated than anticipated. Bevy uses a library called "winit" to do that, which has several limitations, the most important one being that the windows ought to be created on the main thread. On certain platforms, you could get away with using secondary threads, but tricks like that are not a great basis for a universal testing tool. Another problem was that the stubbing was incomplete and led to several places in the game code where the developer had to add a guard to check if a resource existed or not, which was only relevant for tests.

The second approach was to not modify the app as much but to make a wrapper for it. Test gear would get inserted by a wrapper and the game loop would run outside of the app. This was a short-lived experience due to how early on I ran into more winit problems. For example, by default, Bevy does not return the function responsible for starting the game. This is due to how the event loop by default is handled by winit. There

are alternatives, you can use a Bevy inbuilt loop, but this can cause differing behaviour. You can trigger updates in a loop of your own, which is what I tried. This however proved to cause differing results for even simple tests, which made me abandon the approach altogether.

7.2 Initial work on the current artefact

With the current artefact, I wanted something ergonomic to use, as that is one of the primary tenets of the Bevy engine. I wanted something rigid enough but ran the actual game with as few modifications as possible to maximise actual bugs caught and minimise issues with the testing gear. After several attempts, I settled on the approach described in the artefact chapter, and on the 17th of January 2024 published the first version to crates.io for the world to. A day later I then fixed all the glaring problems like lacking documentation and after that, I have been publishing occasional updates. As of writing, BITT has about 1500 downloads on crates.io, the package distribution service for Rust.

In addition to the toolkit, I have developed some demo games that I use to showcase how to use the tools and test them.

7.3 Cooperation with the Bevy community

I interact with the Bevy community on various forums semi-regularly and announced the release as well as asked for feedback there as this was my first contribution to the Bevy ecosystem. Overall, the reception seems to be mild but positive. I am planning on monitoring how the situation develops and pushing the project towards a more mature state.

Some members of the community helped get the survey out there, which I greatly appreciate.

7.4 Plans for future development

I am planning on developing BITT further, as I feel that the current sharp edges do not do enough to fix the underlying problem of testing being difficult. Some features I have planned are:

- Video recordings of tests, useful for debugging CI
- More abstract testing tooling that would let users write the test steps manually
- A command line tool that helps with the setup process
- A dedicated sharding solution that helps run several tests in parallel in CI

In addition to those, there are several pressing concerns like the poor usability beyond the bare minimum use case and spreading awareness of the project, so more people contribute with their experiences and expertise. One problem with the usability that bothers me, but I have not been able to figure out a workaround is the second test problem. As previously stated, with the default options, Bevy doesn't return control flow once the game exits, it just stops the execution completely. This makes running several tests cumbersome, as many mechanisms like loops where every iteration runs a test case are not doable. The current best solution I have found is to use either command line arguments or environment variables to pass information to the application about what test is being run and rerun the game for each test from a shell script. This is cumbersome and makes things like reporting more difficult than they need to be. One solution would be to make a command line tool that handles the looping and test case selection. This would be a lot of work and would not solve the problem as much as it would act as an official workaround.

8. SURVEY OF THE BEVY COMMUNITY

After a few BITT releases and some time, a survey was sent to the Bevy community. The survey had two parts. The first one was meant to be for anyone familiar with Bevy and game development. The second part was meant for BITT users with more topical questions. The survey was posted on Reddit, which is a common communication channel for Bevy users and the Bevy discord server. The Reddit post got to the front page of the Bevy subreddit but received no attention on the rust_gamedev subreddit. For the discord channel, the posting was repeated a few times to ensure users from various time zones had a chance to catch it. A responder also sent the survey to Mastodon. In total, nineteen people answered the survey. This is less than I would like, but enough to see some trends emerging.

The first part had seven questions in total, with most of them gathering background information about the responder. The first question asked about the background of the responder:

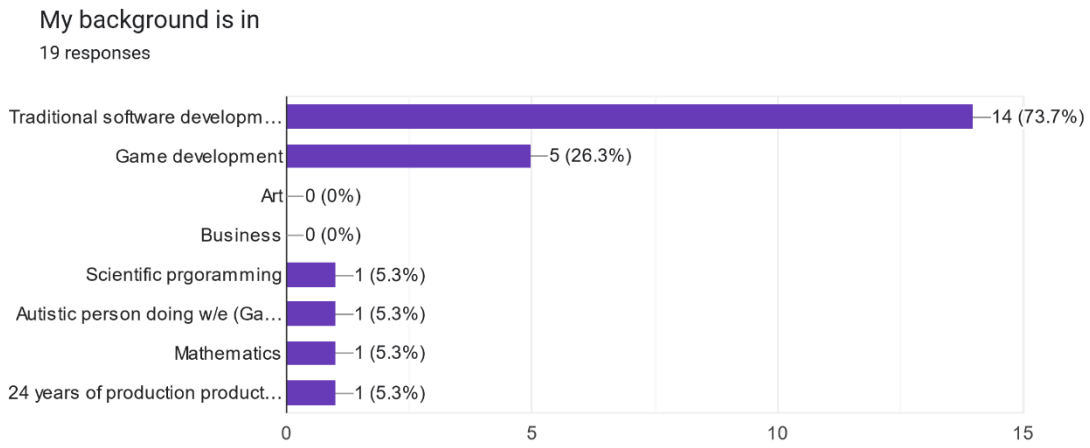

Figure 13 - Survey background question

The question allowed picking any number of options from “Traditional software development”, “Game development”, “Art”, “Business”, and “other”, which the responder could specify themselves. A majority of ten respondents answered with traditional software development, two answered just game development and three answered with both. There were also various other individual answers, with one of them also marking

down traditional software development. I find this interesting considering how the question was sent to forums only read by Bevy users, but most of the responders don't say they have a background in game development. Maybe this is a side effect of lacking games education or a large interest in Bevy specifically from the software development people. The latter is likely at least in part true and stems from how Bevy advertises itself.

The second question was aiming to get an idea of the responder's experience level:

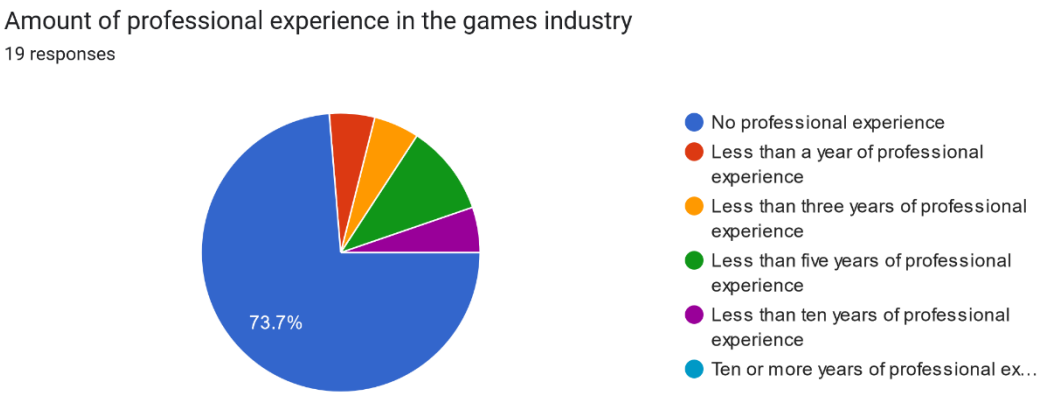

Figure 14 - Survey games industry experience question

The responses here echo the previous question, with a significant majority having no professional experience in the games industry. This does raise the question of whether this data is usable, and a further study should take place before any drastic decisions are made based on these findings.

The third question was about bevy experience with most people using it for occasional dabbling:

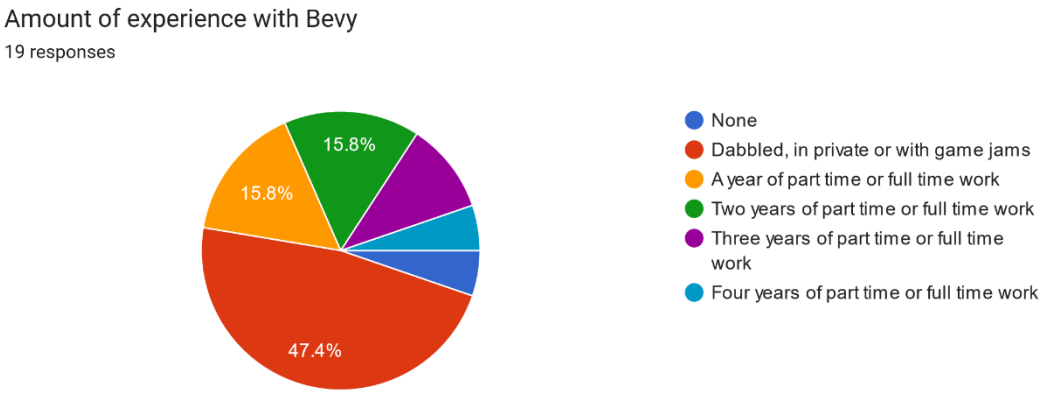

Figure 15 - Survey Bevy experience question

Bevy is still quite fresh so there are not that many game studios that have committed to using it, which could explain this. Understandably committing to a new engine for a professional project is a risk. What I find interesting about this is how the percentage of respondents for dabblers is noticeably smaller than the percentage of respondents with no games industry experience. This implies that there is a noticeable number of responders who are starting their game development careers with Bevy or otherwise don't consider themselves to have a background in the games industry despite years of experience with Bevy.

The next question was about automated testing tools in the games industry, the results of which seem to match what was said in the interviews. The answers for 1 and 5 were labelled "shoddy" and "excellent" respectively with the same labels used for other questions like this one and the results look like this:

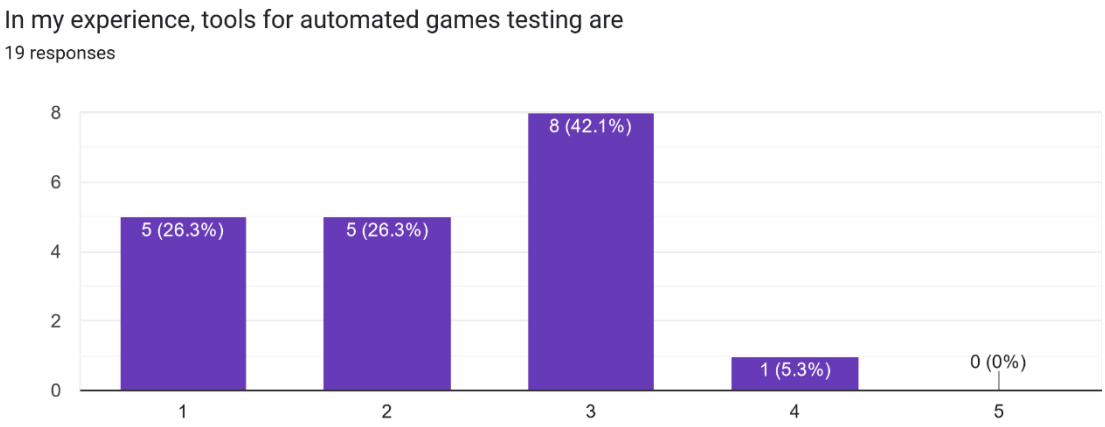

Figure 16 - Survey games industry testing tool satisfaction question

As you can see, just a single responder deemed the tooling better than neutral. This clearly shows that there is work to be done with testing tools in games. I acknowledge that this may be skewed, as the previous questions show that the responders mostly don't come from a game development background.

The next question was an open question about what the responders found lacking in generic game testing tools. It got 13 responses with the main themes being lacking tooling and complexity tied at four mentions with lacking learning materials mentioned three times. The sample size is quite small, but to the extent that it can, I would say this validates the claim that lacking testing is an issue of tooling.

The next pair of questions was like the previous pair, but this time focusing on Bevy instead of the games industry with the bar chart looking like this:

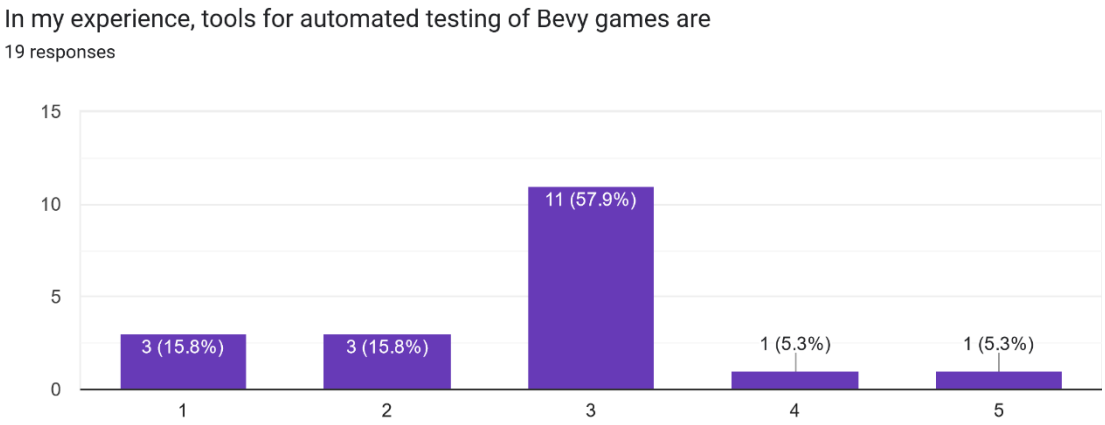

Figure 17 - Survey Bevy testing tool satisfaction question

You can see that most responses indicated a strong neutral stance with one responder even considering it an excellent experience. This supports the claim that testing in an ECS engine is simpler or at least more mediocre rather than outright bad. The open question had 10 responses with immaturity of Bevy and lacking learning materials gaining three mentions each with complexity at two. Complexity is slightly less prevalent in the Bevy answers, but that may just be due to how it is somewhat overshadowed by the immaturity of the project. Bevy is still very new, and it has several areas where tools are lacking, with testing being one of those areas.

The second half of the survey was about BITT. However unfortunately none of the respondents have used BITT, so the responses offer little to no insight. The first question was intended to be about the reactions users had about BITT, but most of the responses indicated the responders had no idea what BITT was.

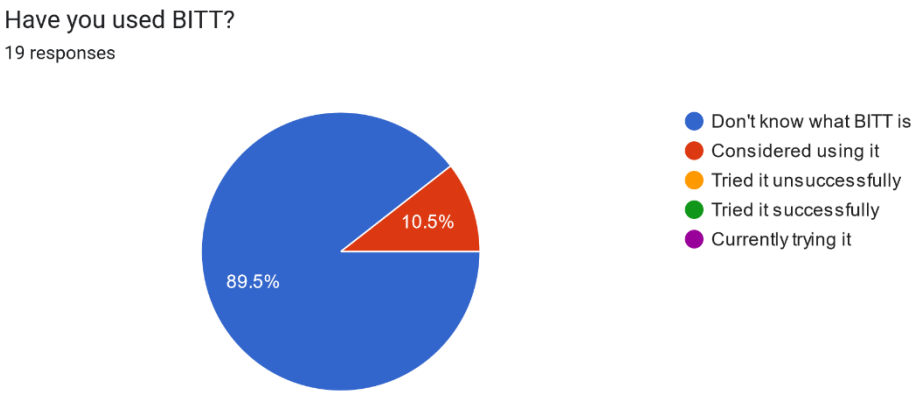

Figure 18 - Survey BITT experience question

In hindsight, I realise that the question is missing the option for people who have looked at BITT but have no intention of using it for one reason or another. One of the responders did express excitement to try BITT out in the open comments. In a way, I do think that the lack of responses is a failure in marketing, but it would be foolish to not acknowledge that the fault may lie in the product. It is possible that none of the responders had tried BITT because they thought it looked like it was not worth the effort or the capture and replay system simply would not work for their games.

The lack of interest in BITT could also imply that Bevy users don't feel like they need it. Figure 17 shows that there is room for improvements in the tooling space, but BITT was likely insufficient to fulfil those needs.

9. CONCLUSION

To summarise, BITT was built to see if a new testing tool would encourage more Bevy developers to test their code. My assessment after having used it is that it just about crosses the threshold of usability, and I will keep using it for my projects but won't expect widespread adoption. The lack of BITT responses in the survey could either mean that BITT was insufficiently advertised or that it did not fulfil the task it was made for.

What the interviews and the surveys did make clear was that the tools for game development and specifically game testing have room for improvement. The survey also points out that many Bevy users feel that some combination of Rust, Bevy, and the ECS architecture makes their games easier to test.

Due to the niche nature of Bevy, the results presented may not apply to the whole industry. The desire for better tools seems universal across the interviews and the survey, but the specifics of these tools remain unclear. It would be beneficial to perform a more comprehensive study in the games industry to examine how the tooling could be improved.

10. BIBLIOGRAPHY

Bevy project maintainers. (2024). *how_to_test_systems.rs*. Noudettu osoitteesta Github: https://github.com/bevyengine/bevy/blob/main/tests/how_to_test_systems.rs

Bevy project maintainers. (21. 1 2024). *Landing page*. Noudettu osoitteesta bevyengine.org: https://bevyengine.org/

Cohen, D. S.;& Bustamante, S. A. (2009). *Producing Games from Business Budgets to Creativity and Design.* Burlington, MA : Focal Press.

Crichton, W. (2020). The Usability of Ownership. *arXiv.org*.

Di Martino, S. F. (2021). Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. *Software Testing, Verification & Reliability*.

Epic Games. (2024). *Automation System Overview*. Noudettu osoitteesta Unreal Engine 5 documentation: https://docs.unrealengine.com/5.3/en-US/automation-system-in-unreal-engine/

Fortin, P.;Fleury, A.;Lemaire, F.;& Monagan, M. (2021). High-performance SIMD modular arithmetic for polynomial evaluation. *Concurrency and computation* .

Gearbox software. (2023). *A Gearbox Plugin for Codeless Automated Tests | Unreal Fest 2023*. Noudettu osoitteesta Youtube: https://www.youtube.com/watch?v=wu5cOM7L5vs

jroddev. (5. July 2022). *Writing Tests in Bevy*. Noudettu osoitteesta jrodev blog: https://blog.jroddev.com/bevy-tests/

Khaliq, Z. S. (2022). A deep learning-based automated framework for functional User Interface testing. *Information and software technology* .

Nauseam, C. (2022). *automated-testing-in-bevy*. Noudettu osoitteesta Chad Nauseam Home: https://chadnauseam.com/coding/gamedev/automated-testing-in-bevy

Neogames Finland Association. (2024). *The Finnish Game Industry Report 2022*. Noudettu osoitteesta neogames.fi: https://neogames.fi/finnish-game-industry-report-2022/

Perkel, J. M. (2020). Why scientists are turning to Rust. *Nature*. Noudettu osoitteesta https://www-nature-com.libproxy.tuni.fi/articles/d41586-020-03382-2

Ruuska, E. (May 2015). Quality assurance testing in video games. Tampere: TAMK.

Shalf, J. M.;& Leland, R. (2015). *Computing beyond Moore's Law.* IEEE.

Unity Technologies. (2023). *Unity Test Framework*. Noudettu osoitteesta Unity documentation: https://docs.unity3d.com/Packages/com.unity.test-framework@2.0/manual/index.html

Wesley, B. (2024). *Gut*. Noudettu osoitteesta Github: https://github.com/bitwes/Gut

White House Press Releases. (2024). Future Software Should Be Memory Safe. *Fact Sheets and Briefings / FIND*.