ASSIGNMENT OF MASTER'S THESIS

Title: 3D game with an open world for Android
Student: Bc. Adam Novák
Supervisor: Ing. Martin Půlpitel
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2020/21

Instructions

Design and implement a 3D open-world role-playing game prototype for Android. Use the Unity platform and Data-Oriented Technology Stack (DOTS) for its creation. Design and implement a real-time combat system, artificial intelligence, and a procedural map generator.
Artificial intelligence must be able to handle combat and daily routine. Procedural map generation must create maps from scratch – it cannot use premade sections.
Adapt user interface and controls for mobile devices. Design the prototype using standard software engineering methods. The implemented prototype must be appropriately documented and tested.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 2, 2019

Master's thesis

3D game with open world for Android

Bc. Adam Novák

Department of Software Engineering
Supervisor: Ing. Martin Půlpitel

January 6, 2021

Acknowledgements

I want to thank my friends and family who supported me during the creation of this thesis, especially because the COVID-19 pandemic made everything a little more challenging. I would also like to thank my supervisor, who enabled me to work on this exciting project and was open to many changes I requested.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 6, 2021 . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2021 Adam Novák. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Novák, Adam. *3D game with open world for Android.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Tato práce se zabývá návrhem a implementací prototypu hry využívající Unity Data-Oriented Technology Stack (DOTS). Cílem práce je prozkoumat možnosti a potenciál DOTS, který je nyní v ranném stádiu vývoje. Unity prezentuje budoucnost jako výkonnou alternativu k GameObjectům. Klíčové oblasti práce zahrnují pohyb postavy na 3D terénu, soubojový systém, umělou inteligenci, procedurální generátor map a uživatelské rozhraní. Výsledná aplikace běží na Androidu 5 a novějším.

Klíčová slova Data-Oriented Technology Stack, DOTS, Unity, Android, UI Toolkit, hra, procedurální generování map, boj, umělá inteligence

Abstract

This thesis focuses on design and implementation of a game prototype in Unity Data-Oriented Technology Stack (DOTS). The aim is to explore capabilities and potential of DOTS, which is in early development and is striving to become a more performant alternative to GameObjects. Key areas include character movement on 3D terrain, combat system, artificial intelligence, procedural map generation and user interface. The final application runs on Android 5 and newer.

Keywords Data-Oriented Technology Stack, DOTS, Unity, Android, UI Toolkit, game, procedural map generation, combat, artificial intelligence

Contents

Introduction **1**

1 Unity Data-Oriented Technology Stack **3**
1.1 The C# Job System . . . 3
1.2 Burst compiler . . . 4
1.3 Entity Component System . . . 4
1.3.1 Entities . . . 4
1.3.2 Components . . . 4
1.3.2.1 GameObject conversion . . . 5
1.3.3 Systems . . . 6
1.3.3.1 Groups . . . 7
1.3.4 Archetypes . . . 7
1.4 Packages . . . 7
1.4.1 Unity Mathematics . . . 7
1.4.2 Animation . . . 7
1.4.3 UI Toolkit . . . 8
1.4.4 Physics . . . 8
1.4.5 Rendering . . . 8
1.5 Project Tiny . . . 9
1.6 Code . . . 9
1.6.1 Accessing components . . . 10

2 Assignment **13**

3 Analysis **15**
3.1 Requirements . . . 15
3.1.1 Assignment requirements . . . 15
3.1.2 Requirements . . . 16
3.2 Procedural map generation . . . 17
3.2.1 Prim's (Jarník's) algorithm . . . 17

3.2.2 Cellular automaton . . . 18
3.2.3 ClassiCube . . . 19
3.2.4 The fantasy map generator (Voronoi) . . . 20
3.3 Data persistence . . . 21
3.3.1 ScriptableObjects and PlayerPrefs . . . 22
3.3.2 Files . . . 22
3.3.2.1 Text files . . . 22
3.3.2.2 Binary files . . . 23
3.3.3 Databases . . . 23
3.3.3.1 SQLite . . . 24
3.3.3.2 LiteDB . . . 24

4 Design **27**
4.1 Asset Management . . . 27
4.2 Architecture . . . 28
4.3 Movement . . . 28
4.4 Procedural Map Generation . . . 29
4.4.1 Procedural terrain generation . . . 29
4.5 Combat . . . 30
4.5.1 Input . . . 31
4.5.2 Abilities . . . 31
4.5.3 Modifiers . . . 32
4.5.3.1 Example . . . 33
4.5.3.2 Equipment . . . 34
4.6 Artificial Intelligence . . . 34
4.6.1 Groups . . . 35
4.6.1.1 Sensors . . . 35
4.6.1.2 Behaviour selection . . . 36
4.6.1.3 Behaviour execution . . . 36
4.6.2 Daily Routines . . . 36
4.7 Character entity . . . 37
4.8 System communication . . . 37
4.9 User interface . . . 38

5 Realisation **41**
5.1 Code editors . . . 41
5.1.1 Movement systems . . . 41
5.2 Asset management . . . 45
5.2.1 Asset loading . . . 45
5.2.2 Database . . . 47
5.2.2.1 Issues . . . 48
5.2.3 Asset management tools . . . 48
5.3 Scene management . . . 49
5.4 Physics . . . 50

5.5 Combat 51
5.5.1 Stats 52
5.5.2 Modifiers 52
5.5.3 Abilities 52
5.5.3.1 Input 52
5.5.3.2 Activation 53
5.5.3.3 Instantiation 53
5.5.3.4 Behaviour 54
5.6 Items 56
5.6.1 Map generation 61
5.6.1.1 Room generation 61
5.6.1.2 Path generation 61
5.6.1.3 Spawning 62
5.6.1.4 Presenter 62
5.6.1.5 Editor 64
5.6.1.6 Performance 65
5.7 Artificial intelligence 66
5.7.1 Sensors 66
5.7.2 Behaviours 67
5.7.2.1 Activatable behaviours 67
5.7.2.2 Daily Routines 68
5.7.2.3 Requestable behaviours 69
5.8 Input 71
5.9 User interface 73
5.9.1 World-space user interface 73
5.10 Animations 75

6 Testing **79**
6.1 Compilation 79
6.2 Testing scenario 80
6.3 Results 80
6.3.1 Performance 81
6.3.2 Testing conclusion 81

Conclusion **83**

Bibliography **85**

A Glossary **89**

B Acronyms **91**

C Contents of enclosed CD **93**

List of Figures

1.1 GameObject conversion.[3] 6
1.2 An example of job syntax.[1] 10
1.3 An example of job syntax. 11

3.1 Example of a maze generated with Prim's algorithm. 18
3.2 Example of a cave system generated with cellular automata. 19
3.3 Example of Perlin noise. 20
3.4 Example of noise generated with random number generator. 20
3.5 Example of a map generated by the fantasy map generator. 21

4.1 Design of modular procedural map generator 30
4.2 Combat parts 31
4.3 Modifier architecture 33
4.4 Example of the modifiers on a health property. 34
4.5 The order of AI (artificial intelligence) group execution 35
4.6 Entity structure of an AI character. The player's character has a similar structure. The primary difference is that it does not have behaviours. 37
4.7 Dependency of important parts 38
4.8 Wireframe of the user interface 39

5.1 The order of execution of the movement systems 42
5.2 Implementation of the Slope and Step calculation system. 44
5.3 Implementation of the Grounding system. 45
5.4 First part of asset loading system. 46
5.5 Second part of asset loading system. 47
5.6 Implementation of a 2D vector field using reflection in UI Toolkit. 49
5.7 Implementation of a ParentScene component 50
5.8 Flow of the combat system 51
5.9 Implementation of the initialization part of a projectile behaviour. 55
5.10 Implementation of the active part of a projectile behaviour. 55

5.11 Implementation of the application part of a projectile behaviour. . 56
5.12 Implementation of loading key item from database. 58
5.13 Implementation of a pick-up event creation system with event entity archetype definition. 59
5.14 Unlock system implementation. 60
5.15 Implementation of the random spawner. 62
5.16 Part of Mesh presenter implementation. 64
5.17 Procedural map editor 65
5.18 Example of two problems caused by sphere cast in two-dimensions. 67
5.19 Combat behaviour selection. 68
5.20 Daily Routines Behaviour system. 69
5.21 Implementation of requestable pathfinding behaviour. 70
5.22 Implementation of Input System update. 71
5.23 Implementation of reading input data for character movement. . . 72
5.24 Implementation of frame per second UI system. 74
5.25 Example of a UXML (Unity Extensible Markup Language) file. . . 74
5.26 Example of a USS (Unity Style Sheets) file. 75
5.27 AnimatorProxy component implementation with custom Authoring MonoBehaviour. 76
5.28 Implementation of a humanoid animation system. 77

List of Tables

1.1 Comparison of various component access methods 10

Introduction

In recent years, Unity grew from a game engine into a platform used by other industries, including film and car industries. Unfortunately, existing scripting methods struggle in more performance-intensive applications. To address this, Unity introduced a Data-Oriented Technology Stack, DOTS for short. Instead of objects (GameObjects and MonoBehaviours), DOTS uses Entity Component System. In DOTS, Unity focuses on using value types instead of reference types, which improves performance by increasing the cache utilization on the CPU. Multithreading is also significantly improved. Compile-time rules prevent race-conditions. Native support and code generation make creating multithreaded code much more manageable.

The focus of this thesis is on creating a prototype game that demonstrates the capabilities of DOTS. First, I will look at what DOTS is and its various packages. After learning the basics about the technology, I will better define the prototype, what it is about and what it will feature. Once the definition is complete, I will look at how to create specific parts of the prototype and define requirements for the prototype. After the preparations, I will create a high-level design of various systems that fulfil the requirements. With design complete, I will ultimately implement the needed systems and solve some previously unforeseen issues. Once I complete the implementation, I will test the prototype and look at how it could be expanded in the future.

Chapter 1

Unity Data-Oriented Technology Stack

Data-Oriented Technology Stack (DOTS) shapes almost every aspect of how the prototype will work. Because it plays such a significant role, it is essential to understand how it works. DOTS is a rewrite of the core of Unity to make it more performant and multithreading friendly. Unfortunately, as of 2020, the only packages considered stable are Burst compiler and C# Job System. All other packages are either in preview, experimental or not yet publicly available. Fortunately, unless DOTS runtime (Project Tiny) is used, the standard Unity features can be used too.

I have decided to use only stable and preview packages. It is challenging to work with experimental packages because they are bare-bones. Luckily, the three most essential packages meet this requirement The C# Job System, Burst compiler, and Entity Component System (ECS).

1.1 The C# Job System

The C# Job System provides developers with a way to run optimized, multithreaded code. Before the job system, Unity did not offer any tools, besides those included in the .NET framework, for multithreading. Many developers ended up either opting out of multithreading entirely or developing their multithreading solutions. The C# Job System tries to change this by giving developers APIs to create jobs which are executed by Unity's internal Job System allowing for better management of resources and detection of race conditions. There is also a new challenge for developers. The job system supports only blittable types[1] and native collections. Consequently, before the creation of a job, every collection (Array, List, Dictionary, Queue) needs to

[1] blittable types—data types with identical memory representation in managed and unmanaged code

be converted into native collection (NativeArray, NativeList, NativeHashMap, NativeQueue).

1.2 Burst compiler

With new performance first mindset, Unity also introduced a new compiler—Burst. Burst compiles code from IL/.NET bytecode to highly optimized native code using LLVM.[1] Burst provides excellent performance benefits, but there is a cost. Burst is limited in what it can optimize, and during my experiments with it, there were issues with foreach and using statements.

While Burst provides excellent performance benefits, it also poses further restrictions on the developers. It cannot optimize every piece of code. Unity actively develops Burst, adding new features and reducing the restrictions imposed in developers. In 2020 there were two major Burst updates.

1.3 Entity Component System

The Entity Component System (ECS) is the core of the Unity DOTS (Data-Oriented Technology Stack). As the name indicates, it consists of three parts: Entities, Components, and Systems. In Unity ECS is contained inside an Entities package.[1] In the following sections, I will look at each part in more detail.

1.3.1 Entities

An entity represents an individual "item"; it can be anything from a message to a player's character. An Entity is essentially just a unique identifier (ID). It holds no data or logic. Instead, data is in components, and logic is in systems.

In Unity, an Entity is a structure (struct) with two integers — Index and Version. The index uniquely identifies an active entity; however, when an entity is destroyed, the same index can be used by a different entity, this is called recycling. To distinguish between the old, recycled entity, and the new one, the version is used. Every time an Entity is recycled, its version is incremented, so the system knows when you try to access a recycled (destroyed) entity.

1.3.2 Components

A component represents data. It contains all the data that the entity has, for example, health, items in inventory, known skills etc. One entity can have multiple components attached to it. Components contain no logic; logic is in systems.

[1] a collection of compiler and toolchain technologies

In Unity, standard components (IComponentData) attached to an entity must be unique. For example, there cannot be multiple Health components on a single entity. Also, components are blittable structures. Blittable structures have two main advantages in performance compared to classes.

1. Structures are value types, meaning I do not have to dereference them (access more memory) to get the data.

2. Blittable structures do not have to be converted from managed to unmanaged memory, improving performance.

In .NET, blittable types are only numbers. Some examples of non-blittable types include Char, Boolean, String, Object, and Array.[2] This does not mean that text or arrays cannot be in components, but there are some restrictions. Unity provides fixed-length array types, including strings, that I can use inside components. The problem with fixed-length array types is that they always take the same amount of memory no matter how much data I store in them, and there is no way to extend them. To solve this problem, Unity also supplies a component called a dynamic buffer (IBufferElementData). Compared to standard components, these components are contained in a resizable buffer which is attached to an entity.

There are also two components, which allow reference types: shared components (ISharedComponentData) and class components (IComponentData). Shared components are components which share with multiple entities. All entities with the same shared data value are placed inside the same chunk. Class components are the same as struct components, but they are a reference type (class) instead of a value type (struct).

1.3.2.1 GameObject conversion

In Unity 2020.2, it is impossible to create or modify entities outside of scripts. The only way to create them before the launch of the game is with GameObject conversion.

GameObject conversion consists of two parts: Authoring and Conversion. Authoring contains GameObjects with scripts that inherit from IConvertGameObjectToEntity interface, which tells Unity to call a conversion method on them. The interface also defines a Convert method. During conversion, Unity creates a new entity for each GameObject and invokes the Convert method with the object's entity, Entity Manager and conversion system object. In the convert method, components, with data from the GameObject, are added to the entity. After the conversion, the entity is ready for use.

GameObject conversion is a relatively slow process. Luckily, Unity can do the conversion during the compilation of the application, so the runtime only loads the post-conversion data. Ahead of time conversion is not always

possible, and one example is when GameObject is instantiated in code because the reference to that GameObject is needed.

The developer can also choose to keep the GameObject after the conversion. Keeping the GameObject allows DOTS systems to access both ECS components and GameObject components.[3]

Figure 1.1 contains a diagram showing the conversion process.

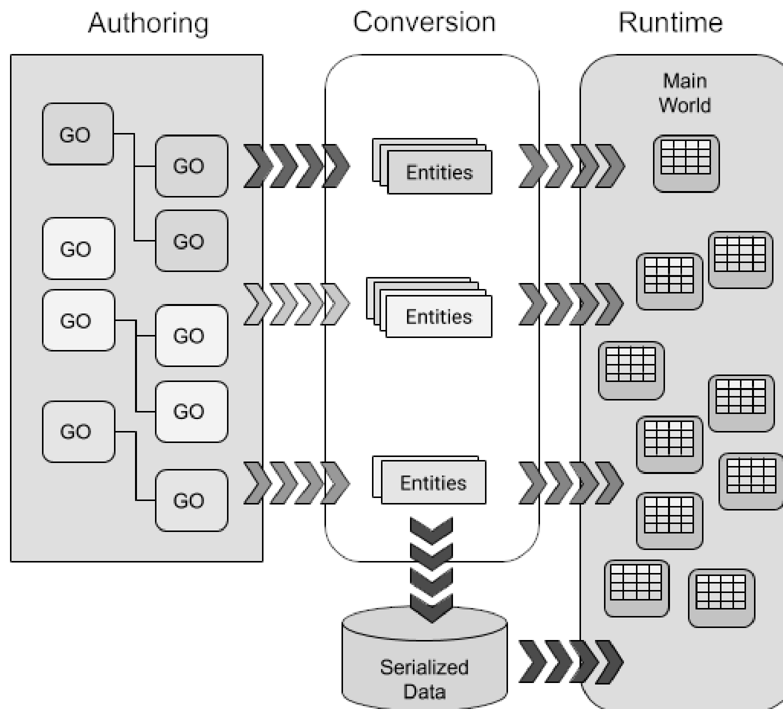


Figure 1.1: GameObject conversion.[3]

1.3.3 Systems

A system provides the logic that transforms component data from its current state to its next state. System iterates over all the components that satisfy its criteria (set of components). A single system might only do a smaller part of the complete behaviour to reuse as much code as possible.

Unity provides an abstract class (SystemBase) that offers many useful APIs, but for now, I will focus on simple iteration. At first, Unity had different implementations for systems that used only the main thread and those that used job systems; however, that changed with SystemBase. SystemBase provides a more straightforward API that Unity automatically converts in the background to its runtime format (using code generation). In Unity, programmers specify which components the entity must have and which it must

not have. Unity automatically iterates over all the entities and provides the matching entities in each iteration.

1.3.3.1 Groups

Groups are unique systems that help with system organization. They do not contain any logic themselves but can have children systems. As with any other system, groups can run before or after any other system, even another group. Groups can also be used to update their children manually.

1.3.4 Archetypes

An archetype is a unique collection of types of data components and is used to categorize entities. Archetypes improve performance as every entity within an archetype is stored within a chunk that is passed to appropriate systems. If an archetype is not specified, it is up to Entities to dynamically resolve it at runtime. Adding and removing components can be costly for the same reason.

For event entities[1] it is beneficial to create archetypes. Adding a new entity to an existing archetype is a cheap operation. In tests, I conducted, creating new entities with archetypes was two times faster than adding components to existing entities.

1.4 Packages

Many parts of DOTS are encapsulated inside a package. In this section, I will look at select few which are relevant to this prototype.

1.4.1 Unity Mathematics

Unity Mathematics is a C# math library with support for Burst compiler. It provides vector types and math functions. The syntax is similar to shaders, which has been critiqued by many because it violates C# naming conventions.[2] Unity argued that in C# built-in types (such as int and float) also user lower-case type names and that making functions lowercase makes them more compatible with shader code.[4]

1.4.2 Animation

An animation package is at a very early stage of development. As of 2020, Unity considers it experimental — very early preview. It offers animation blending, IK (inverse kinematics), root motion, layers, and masking. Those

[1] Event entities are entities that exist only for a frame or two.

[2] For example, functions in math class use all lower_case names instead of PascalCase names.

who would look for GUI (graphical user interface) would be out of luck as it can only be accessed with scripts. For this prototype, I decided to use the current animation system, Mecanim.

1.4.3 UI Toolkit

UI Toolkit is the third generation UI (user interface) solution in Unity. The Web inspired UI Toolkit. Just as websites, it has markup language (UXML) and style sheet language (USS). In Unity 2020.2 the UI Toolkit is considered stable for editor tools and in preview for game UI. UI Toolkit is attempting to improve complex hierarchies' performance, which causes issues in previous Unity UI solutions.

Unity did not leave UI designers in the cold. There is a preview package (UI Builder) for UI designers adding a visual editor to UI Toolkit.

1.4.4 Physics

In DOTS, Unity offers not one, but two physics solutions: Unity Physics and Havok Physics. Both of which are free for all Unity developers. These new solutions are a complete departure from the current Nvidia PhysX solution. Each solution has its advantages and disadvantages, but both offer the same API, making the switch between them as simple as downloading a package and flicking a switch.

The Unity Physics is a stateless open-source physics solution written in HPC# (High Performance C#). Because it is stateless, it is suitable for multiplayer games. The solution was developed by Unity with help from Havok. Unity presents this as a solution suitable for most games—from very small to very large.

The Havok Physics is a stateful closed-source physics solution written in native C++. It supports caching and is very fast and stable.

Both engines can run at the same time. It is also possible for a developer to implement their solution instead of using those provided by Unity.[5]

1.4.5 Rendering

In recent years, Unity made significant efforts on reworking the render pipeline, introducing Scriptable Render Pipeline (SRP). The SRP has two provided variants: URP (Universal Render Pipeline) and HDRP (High Definition Render Pipeline). Anyone developer can, however, create their own variant. The HDRP is a render pipeline with a focus on the best graphics possible. On the other hand, the URP focuses on running everywhere at the cost of fewer graphics features. These render pipelines took many years to develop and are still in active development with more features coming with every release.

To not throw these efforts away, Unity decided to introduce Hybrid Renderer package for DOTS. Hybrid Renderer is not a render pipeline, but rather

a system that bridges DOTS with URP and HDRP. Unfortunately, not every feature is supported and for URP many essential features, such as point lights and lightmaps, are unsupported.[6]

Unity is also developing a pure DOTS renderer that is part of Project Tiny. This renderer is, however, much less capable and only supports a basic feature set.

1.5 Project Tiny

It may seem that Unity DOTS still needs existing GameObject oriented architecture to work, but this is not entirely true. Project Tiny is the ultimate DOTS experience on Unity.

Project Tiny focuses on small, light, and fast experiences such as mobile games, playable ads, and instant social experiences. Project Tiny targets new DOTS runtime which does not rely on existing UnityEngine components. It supports desktop, web, Android and iOS platforms. The 3D and 2D rendering is very fundamental and not suitable for most games. Input, audio, animation, physics are all provided as part of Project Tiny; however, each of these components has a minimal feature set. There is currently no UI support, but UI Toolkit should be supported by the end of 2020.

Developing anything larger in Project Tiny is very difficult. There is no way to substitute missing features with components from standard Unity. Because of this, I did use Project Tiny in this prototype.[7]

1.6 Code

Unity Entities has various ways to implement components and system. In this section, I will look at some of these implementations to make examples easier to understand.

The most frequently used and performant method is Entities ForEach function in a SystemBase system with struct components. Both Burst and jobs support it. Struct components are standard C# structs that inherit from IComponentData. Entities ForEach function is a lump of syntactic sugar. Behind the scenes, a code generator converts this function into a job syntax. An example of how Entities ForEach looks like is in Figure 1.2, and an example of how job syntax might look like is in Figure 1.3.

The only system type that allows the use of Entities syntax is SystemBase. There are other base classes for systems, but Unity is working on unifying them under SystemBase. When an example of such a class occurs, I will explain the difference.

It is also possible to Burst compile struct systems. However, Unity is still actively working on this, and the documentation does not even mention it, yet.

```
protected override void OnUpdate() {
  Entities
  .WithAll<LocalToWorld>()
  .WithAny<Rotation, Translation, Scale>()
  .WithNone<LocalToParent>()
  .ForEach((
    ref Destination outputData,
    in Source inputData
  ) => {
    /* do some work */
  })
  .Schedule();
}
```

Figure 1.2: An example of job syntax.[1]

1.6.1 Accessing components

There are multiple ways to access a component and which ways are available depends on the type of a component. Accessing components directly with EntityManager should be avoided. EntityManager can only be called on the main thread and thus makes it impossible to create multithreaded code. The following table shows where the different components are supported.

Table 1.1: Comparison of various component access methods

	struct[1]	class[2]	shared[3]	buffer[4]
EntityManager Get/Set/Remove	✓	✓	✓	✓
ForEach.Run without Burst	✓	✓	✓	✓
ForEach.Schedule with Burst	✓	✗	✗	✓
GetDataFromEntity	✓	✗	✗	✓
SystemBase Get/Set[5]	✓	✗	✗	✓
CommandBuffer Remove	✓	✓	✓	✓
CommandBuffer Set	✓	✗	✗	✓

[1] Struct implementing IComponentData (blittable)
[2] Class implementing IComponentData
[3] Struct implementing ISharedComponentData
[4] Struct implementing IBufferElementData
[5] Uses code generation to replace the call with the best available method to access data.

```
public class RotationSpeedSystem : SystemBase
{
  [BurstCompile]
  struct RotationSpeedJob : IJobParallelFor
  {
    [DeallocateOnJobCompletion] public NativeArray<ArchetypeChunk> Chunks;
    public ArchetypeChunkComponentType<RotationQuaternion> RotationType;
    [ReadOnly] public ArchetypeChunkComponentType<RotationSpeed>
        RotationSpeedType;
    public float DeltaTime;

    public void Execute(int chunkIndex)
    {
      var chunk = Chunks[chunkIndex];
      var chunkRotation = chunk.GetNativeArray(RotationType);
      var chunkSpeed = chunk.GetNativeArray(RotationSpeedType);
      var instanceCount = chunk.Count;

      for (int i = 0; i < instanceCount; i++)
      {
        var rotation = chunkRotation[i];
        var speed = chunkSpeed[i];
        rotation.Value = math.mul(math.normalize(rotation.Value), quaternion
            .AxisAngle(math.up(), speed.RadiansPerSecond * DeltaTime));
        chunkRotation[i] = rotation;
      }
    }
  }

  EntityQuery m_Query;

  protected override void OnCreate()
  {
    var queryDesc = new EntityQueryDesc
    {
      All = new ComponentType[]{ typeof(RotationQuaternion), ComponentType.
          ReadOnly<RotationSpeed>() }
    };

    m_Query = GetEntityQuery(queryDesc);
  }

  protected override void OnUpdate()
  {
    var rotationType = GetArchetypeChunkComponentType<RotationQuaternion>();
    var rotationSpeedType = GetArchetypeChunkComponentType<RotationSpeed>(
        true);
    var chunks = m_Query.CreateArchetypeChunkArray(Allocator.TempJob);

    var rotationsSpeedJob = new RotationSpeedJob
    {
      Chunks = chunks,
      RotationType = rotationType,
      RotationSpeedType = rotationSpeedType,
      DeltaTime = Time.deltaTime
    };
    this.Dependency rotationsSpeedJob.Schedule(chunks.Length,32, this.
        Dependency);
  }
}
```

Figure 1.3: An example of job syntax.

Chapter 2

Assignment

The provided instructions are vague and only define a handful of very general requirements. It is essential to define better what the prototype is about. With the provided instructions in mind, I came up with an idea for a game.

The game world is divided into a procedurally generated areas and a main, handcrafted world. The player ventures from the main world into various procedurally generated areas to find keys. These keys unlock new areas in the main world. The AI in the main world is peaceful and uses routines to make the world feel more alive. The procedurally generated areas, on the other hand, have hostile AI that attacks players on sight. The player has no means of fighting back and must rely only on wits and speed to find a key and leave the area.

Chapter 3

Analysis

In this chapter, I will analyse several aspects of the prototype. First, I will create a set of requirements to define what the implementation must do. After requirements, I will examine implementations of various approaches to procedural generation. The analysis finishes with a look at various ways to store data in Unity.

3.1 Requirements

The software requirements describe the features and functionalities of the target system.

3.1.1 Assignment requirements

Assignment requirements contain only requirements which are in the assignment of this thesis.

- 3D graphics,
- open world[1],
- role-playing genre[2],
- built for Android platform,
- built on Unity platform,
- uses Unity DOTS (Data-Oriented Technology Stack),
- has real-time combat,

[1] A world that player can explore and approach objectives freely.
[2] Player assumes a role of a character withing a fictional setting.

- includes characters controlled by AI,
- AI supports combat,
- AI supports daily routines,
- includes procedural map generator which builds maps from scratch,
- has UI (user interface) adapted for mobile devices.

3.1.2 Requirements

List of all requirements combining both functional and non-functional requirements into a single list.

- Prototype requirements:
  - runs on Android 5.0 and newer (API 21),
  - user-interface supports various screen densities and aspect-ratios,
  - uses Unity DOTS,
  - 3D graphics,
  - role-playing genre.
- Player character requirements:
  - responds quickly to player inputs,
  - can climb smaller obstacles (such as stairs),
  - can move at variable speeds.
- World requirements:
  - open world,
  - procedurally generated environments:
    - does not use premade sections,
    - can generate rooms,
    - every area is accessible,
    - location selection for
      - items
      - player
      - enemies
      - portals (to the next level or/and an exit)
    - spawned items are accessible,
    - result is mesh or texture,

- Artificial intelligence requirements:
  - able to fight,
  - has daily routines,
  - can move,
  - can find paths between two positions in the non-procedurally generated world,
  - integrates with movement used for the player character,
  - support for senses,
  - behaviours:
    - combat behaviour:
      - target selection,
      - ability usage,
      - integration with the combat system,
    - movement behaviour,
    - behaviour selection,

3.2 Procedural map generation

Procedural map generation is a method of creating maps with algorithms instead of manually. Games often use procedural map generation to create a near-infinite number of map layouts. Diablo 3 uses tiles made by artists, with hand-picked areas for asset placement, stitched together and filled with objects by the algorithm. Premade tiles, however, violate the requirement from instruction to create maps from scratch. Probably the best-known example of procedural map generation from scratch is Minecraft. Map generation in Minecraft is unfortunately not only complicated but also proprietary. Instead, I will look at ClassiCube, open-source reimplementation of Minecraft Classic.[1] First, I will look at Prim's (sometimes called Jarnik's) algorithm.

3.2.1 Prim's (Jarník's) algorithm

Prim's (also known as Jarník's) algorithm is a greedy algorithm that finds a minimum spanning tree on an unweighted graph. By adding randomization, I can modify this algorithm to generate a maze. Unfortunately, the algorithm creates only corridors, which makes for a rather dull level design.[8] Example of a maze generated with this algorithm is in Figure 3.1.

[1] Minecraft Classic is the official name for version 0.0.23a_01 of Minecraft. Anyone can play it inside their internet browser for free at classic.minecraft.net.

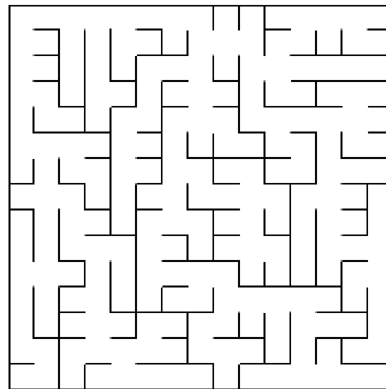


Figure 3.1: Example of a maze generated with Prim's algorithm.

3.2.2 Cellular automaton

Cellular automaton is a cell model with following properties:

- cells live on a grid,
- each cell has a state,
- each cell has a neighbourhood.[9]

To create a cave system, I can get help from the Game of Life rules, invented by Cambridge mathematician John Conway.[10] The rules are as follows:

1. Each cell with one or no neighbours dies, as if by solitude.
2. Each cell with four or more neighbours dies, as if by overpopulation.
3. Each cell with two or three neighbours survives.
4. Each cell with three neighbours becomes populated.

In the original Game of Life, the cellular automaton created interesting ever-changing shapes. However, I can use it to create impressive cave systems. Example of a cave system, generated by this method, is in Figure 3.2.

The downsides of this method are lack of control and similar style of each map.

Figure 3.2: Example of a cave system generated with cellular automata.

3.2.3 ClassiCube

ClassiCube is a complete opensource reimplementation of Minecraft Classic. It is written in programming language C and supports OpenGL and Direct3D 9. The game has a very well documented map generation algorithm.

The generator generates the world map in 9 steps. Choose map size, create height-map, create strata, carve out caves, create ore veins, flood fill-water, flood fill-lave, create a surface layer, create plants. Only height-map, water and surface generation are relevant to this prototype.

The game uses various noise functions, but all of them rely on Perlin noise. Perlin noise, named after its inventor Ken Perlin, is a type of gradient noise. Ken Perlin originally designed it to create procedural textures for the movie Tron in the early 1980s. Nowadays, it is often used in video games for procedural map generation, among other things. Unlike random number generator, Perlin noise forms a naturally ordered ("smooth") sequence of pseudo-random numbers.[9] For visual reference see Figures 3.3 and 3.4.

The game creates height-map using three octave noise functions.[1] Height-map defines height at which the algorithm places surface blocks, practically defining the landscape. With simple math equations and threshold conditions, the algorithm generates a value for each pixel (block) on the height-map.

The above-ground water is flood-filled into the map. It spreads first on the x-axis and then the z-axis from the edges of the map.[2] The algorithm fills the water at the height of water level minus one.

At this point, the algorithm created a height-map and filled the edges with water if possible. However, the terrain still has no materials (textures).

[1] An octave noise function is a summation of multiple noise functions (in this case Perlin noise) where each function has a different frequency and amplitude.

[2] Most 3D games have a coordinate system that uses y-axis for height and xz axes for the width and depth.

Figure 3.3: Example of Perlin noise.

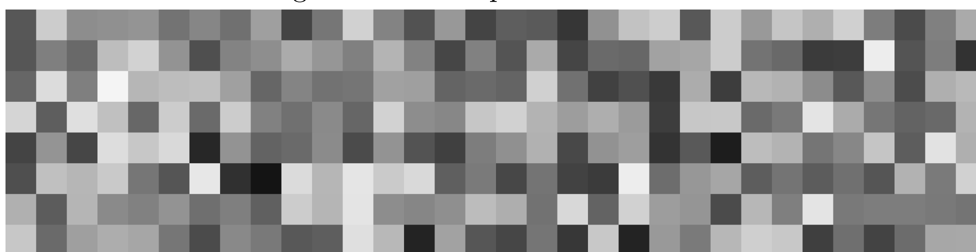


Figure 3.4: Example of noise generated with random number generator.

The algorithm decides the material in the final crucial part, the surface layer creation phase.

The surface layer creation supports only three types of materials: sand, gravel, and grass. The algorithm uses several conditions that check probabilities and position relative to water, to decide what material a block will have.

3.2.4 The fantasy map generator (Voronoi)

Fantasy map generator created by artist, designer, teacher, and researcher Martin O'Leary. The algorithm uses Voronoi polygons and various other methods to generate realistic-looking fantasy maps. The algorithm makes these maps in six steps.

The first step generates many random points to create an irregular grid. According to the author, the irregular grid helps hide artefacts that a regular grid would create and gives the map more organic feeling. These points are then relaxed using Lloyd relaxation.

The second step of the algorithm focuses on generating landscapes. Unfortunately, it is impossible to simulate natural landscape creation processes in a reasonable amount of time, so the algorithm uses clever tricks to get around this problem. The algorithm creates "proto-landscapes" built with geometric primitives. These primitives make up the general outline of the terrain. There are also few operations: normalize, relax, round and set sea level. A particular sequence of primitives and operations defines the landscape.

The third step is a more physical process, erosion. More specifically, water

erosion which is by far the most significant type of erosion. In the context of the map, the erosion adds features to our terrain while making it smaller. The algorithm assumes constant rainfall across the whole map and traces how the water would flow down.

In the fourth step, it is time to render the terrain on a map. The algorithm renders coastlines, rivers, and slopes.

The fifth step renders cities and borders. For city generation, the algorithm prefers building cities near a water source and away from each other. The borders are calculated simply by measuring the distance of each grid point from the city (using a custom metric).

The algorithm adds labels to the generated map (such as borders, city markers, rivers) in the final step.[11]

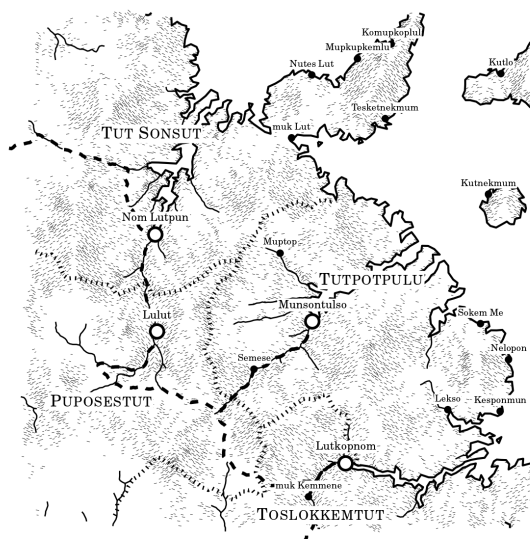


Figure 3.5: Example of a map generated by the fantasy map generator.

The Fantasy map generator is suitable for generating large scale maps. One of the most significant missing features is road generation. For an actual level generation, this algorithm could be a good start. However, the map generation would end up being very tricky. Nonetheless, some of the ideas in this generator can be used to create more realistic looking maps.

3.3 Data persistence

There are many ways to store various types of data in Unity games. To narrow down the possibilities, I am going to create groups. Each group can store static (such as item definitions and procedural configurations) and dynamic

data (player save data). It is important to note that as of the time of writing, there were no solutions that worked out of the box with Unity DOTS.

3.3.1 ScriptableObjects and PlayerPrefs

A ScriptableObject is a data container that developers can use to store large amounts of data independently from class instances. ScriptableObjects are reusable between many instances of an object, reducing the amount of memory needed and the probability of a mistake. There are many different editor utilities that developers can use to speed up the development available for ScriptableObjects. During runtime, ScriptableObjects are read-only. Unfortunately, they do not yet support Unity DOTS and need to be manually converted to entities.[12]

For smaller user data, developers can use PlayerPrefs. PlayerPrefs is an API that allows developers to store user data as key-value records. It supports only numbers (int and float) and text (string). On Android PlayerPrefs use SharedPreferences, which load all preferences into memory on first access, storing larger JSON files is therefore not a good idea.[13]

Unity does not provide a solution for storing large amounts of user data.

3.3.2 Files

Nearly every solution that stores data locally uses files, because files are the only way to reasonably and persistently store data on a computer or a phone. In this case, using files means directly managing data in files instead of leaving it to a library. The primary two categories are text and binary files.

3.3.2.1 Text files

Text files store data in a human-readable format. Instead of using a computer representation of data, we convert the data to text and store it inside a file. However, the data can still be scrambled with, for example, encryption.

The most common formats for storing data in text files are JSON and XML. When I refer to text files, I will only refer to JSON and XML. In this comparison, both formats have the same set of advantages and disadvantages, so I will not differentiate between them.

Advantages

- Human-readable,
- Git-changes tracking,
- less likely corruption,
- order of items does not matter.

Disadvantages

- Less space efficient.

3.3.2.2 Binary files

Storing data in a binary file is a more efficient operation. Binary (native) files are smaller and need not be converted to text, saving on processor time. Applications often use binary format only internally, and as a result, there are no significant standards for binary formats. The binary file formats are prone to breaking. Even different architectures can produce different binary files (the most common reason is Endianness). There is also an increased risk of corruption when the format changes; for example, when a new property is added.

Advantages

- Space efficient,
- performant.

Disadvantages

- Prone to corruption,
- Git can't detect changes,
- error recovery is difficult,
- humans can't efficiently read binary code.

3.3.3 Databases

A robust system that stores data in an organized form is called a database. There are many ways to categorize databases, but I will focus on embedded[1] SQL and NoSQL, more precisely, SQLite and LiteDB.

[1]Embedded databases run as a part of an application rather than in a separate process or a separate machine.

3.3.3.1 SQLite

SQLite is a Relational Database Management System (RMDBS). It is written in C and distributed as a C library. Because C is a compiled language, SQLite needs to be recompiled for each processor architecture. The database itself is embedded inside the application that runs it. SQLite is a popular choice among embedded databases and is included in Android. Because every version of Android has a different version of SQLite, it is wise to include SQLite in the application, improving the stability of the application and reproducibility of some issues.

Advantages

- performance,
- atomicity, consistency, isolation, durability (ACID),
- errors are automatically detected and repaired,
- optional encryption,
- good editing software support.

Disadvantages

- needs to be compiled for each processor architecture,
- lists are not stored easily,
- Git can't detect changes.

3.3.3.2 LiteDB

LiteDB is a small, embedded, open-source, .NET NoSQL database. Because it is written in C#, it can be reused on any platform Unity supports. The library supports many important concepts such as indexing, encryption (which can help protect a saved game against tampering), thread-safety, and recovery after a failure. Uncommonly to NoSQL databases, LiteDB supports ACID and is thread-safe. Unfortunately, it cannot be used inside jobs, because the library code cannot be statically analysed. From my experience, the API is intuitive and easy to use.

One of the more popular projects that use LiteDB is password manager Bitwarden.

Advantages

- Portable,
- ACID,
- thread-safety,
- recovery after failure,
- optional encryption,
- automatic property serialization.

Disadvantages

- GIT can't detect changes,
- only a few external tools.

Chapter 4

Design

Software design is a process of transforming user requirements into a form, that helps the programmer in the implementation phase. The prototype requires a very flexible design due to the rapid development of the DOTS. To make the design more flexible, I decided to make it less detailed and worry about the details during the implementation phase. This decision proved to be very fruitful as some parts of the DOTS API changed almost every week and sometimes even changed between design and implementation.[14]

4.1 Asset Management

Managing assets in Unity is not a straightforward task. The most basic approach is using the Asset Database in editor and Resources in a player. Asset Database provides mostly unrestricted access to any file inside the project folder. The primary limitation of the Asset Database is that it can only be used within the editor.[15] Resources provide the API to load assets in a player but can only access files inside a resources folder[1]. Usage of Resources folder significantly affects how the project is structured and makes it more challenging to manage. Developers can also use Asset Bundles, but they require much work from the developer to implement.[16]

Luckily, there is a third solution, the Addressables library. The Addressables library provides developers with an asynchronous way of loading any asset inside the Assets folder[2]. It supports loading assets from various locations, including the internet.[17]

Unfortunately, none of these solutions supports DOTS. It is not clear how will DOTS load assets in the future. However, the development team is planning to bring some parts of Addressables into DOTS as built-in functionality.

[1] Resources folder is any folder named "Resources" within the project.

[2] In every Unity project, the Assets folder is a folder contained within the projects top-level directory. It contains all the games assets such as scripts, textures, models, sounds, and prefabs.

The first step is sub-scenes. Unfortunately, they only support pure DOTS scenes, so they are not viable for scene management in this prototype.[18]

I have chosen Addressables as the most promising approach to asset management.

4.2 Architecture

Software architecture is not a precisely defined term. In this thesis, I will use Ralph Johnson's definition that architecture is the shared understanding that the expert developers have of the system design.[19]

4.3 Movement

There are many parts involved in making a character move. First, there has to be some input, that systems convert into motion. These systems have to handle collisions, steps, slopes, rotation, and more, depending on the requirements. Lastly, the right animation has to play.

The player or the AI provide the input in this prototype. There are two primary methods of making a character move. Either modifying the position directly or using a physics system and applying force to the character to move it. The choice is very much dependent on the game. For this prototype, I chose physics system, because the character can better interact with other physics objects.

There are several hidden problems when using physics. The primary cause of these problems is the separation of physics and animation. Usually, the representation of the character in the physics system is much simpler than its actual 3D model. In the simplest form, the character is a capsule that moves through the world. The model and animations visible to the player are often only for show. The primary reason for this simplification is performance. It is orders of magnitude faster to handle physics between the capsule and a box than between two complicated and often non-convex meshes.

The first problem is that if a character capsule walks down a slope, it does not stick to the ground but instead floats forward and slowly falls to the ground. The second problem is climbing steps or small bumps. The physics will not allow it, and the character capsule gets stuck on even the smallest of steps. In older linear role-playing games, characters could get stuck on a small pebble, unable to step over it.

A system can solve the first problem by guaranteeing the player stays on the ground if the ground is close enough. Another system can solve the second problem by lifting the character once it is near a step.

4.4 Procedural Map Generation

Procedural map generation is a task with varying levels of difficulty. When approached from the more straightforward side, it entails only a simple algorithm that decides at random if a cell inside a two-dimensional grid is passable or not. The more complicated approach is a procedural generation of worlds, beings, decorations, and virtually everything. The most ambitious procedural map generation project known to me is a game No Man's Sky, where a player has the whole, entirely procedurally generated, universe to explore.

In the next section, I will look at the map generator design, which I created based on the requirements specified in the Analysis chapter.

4.4.1 Procedural terrain generation

With the list of requirements complete, I can now create a high-level design. Very early on, it became apparent that the system should not use DOTS. The design would be unnecessarily complicated. There are no performance benefits in using DOTS because map is procedurally generated only during level loading. I settled on using standard object-oriented programming for this task.

There is one thing I wanted to keep from ECS, and that is modularity. Every part of the generator should be replaceable, except for core logic. I separated the generation into four replaceable parts: room generator, path generator, spawner, and presenter.

The room generator is responsible for generating a set number of continuous rooms with restrictions only from map configuration. The path generators job is to ensure a player can get from any room to any other room. The task for spawner is to spawn items, player, enemies, and portals. Last but not least, presenter presents everything in a proper format. The format depends on the situation; in the editor, it can be an image; on the other hand, in-game it can be a mesh.

See Figure 4.1 for the diagram of the procedural generator's architecture.

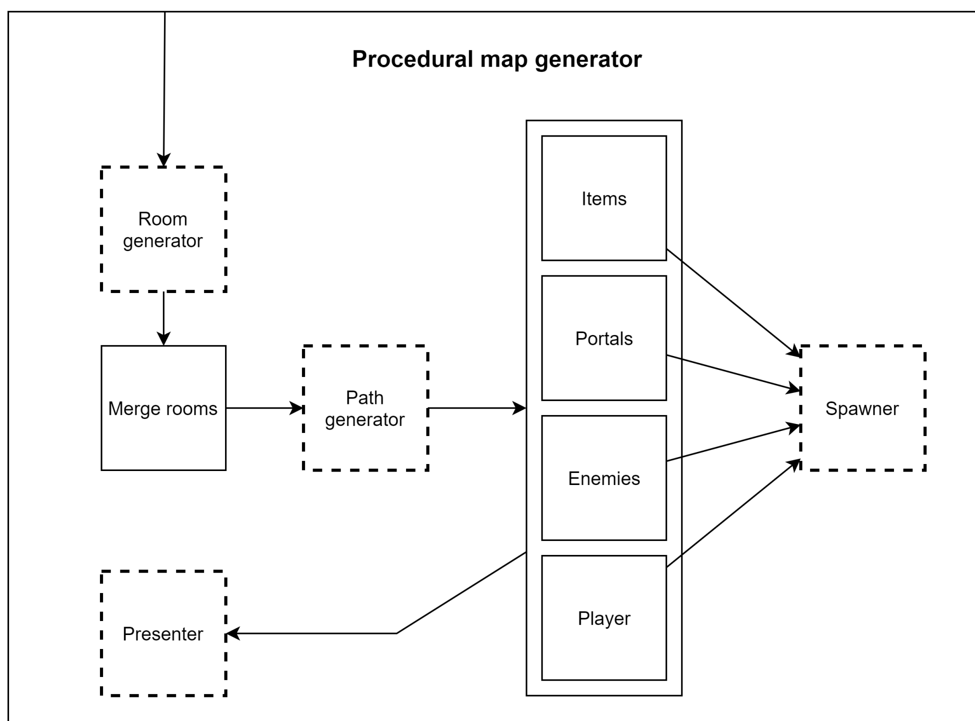


Figure 4.1: Design of modular procedural map generator

The dashed borders represent replaceable parts. The solid borders represent parts of the core.

4.5 Combat

In most games, combat is a fight where a player tries to beat enemies. In role-playing games, there are often rewards that enemies drop when they die. Games often feature health with only two states, life and death. It could be frustrating to have a character that is weaker the less health they have. Games instead do the opposite, making the players character stronger at low health.

In this prototype, I decided to incentivize combat avoidance to players. Because the prototype is limited in scope, the result is that the player has no means of fighting back and has to avoid enemy attacks. Nevertheless, the design needs to allow possible future integration of player combat.

I focused on range abilities for enemies because melee combat systems are very challenging to implement properly. One of the main challenges is that the actual combat systems and animations are often separate, which results in a difference between what the player sees and what happens. The combat system performs only simple calculations to determine hits while the animation system can perform flashy moves.

The combat has two parts, abilities, and modifiers. Each ability describes

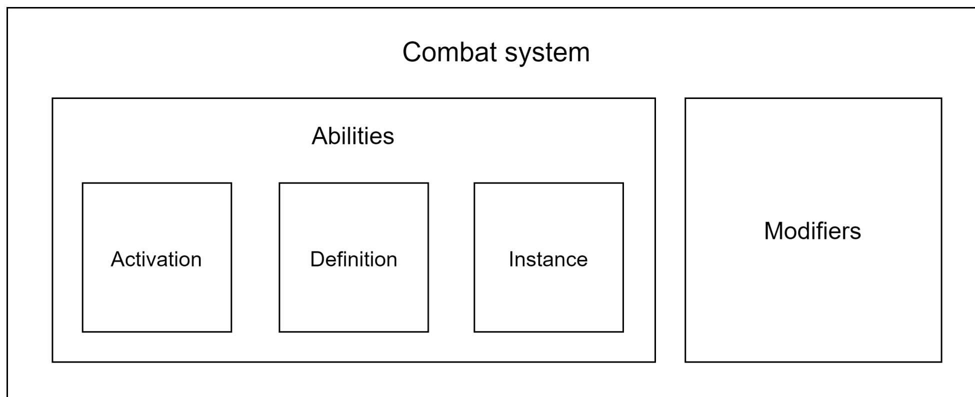


Figure 4.2: Combat parts

and executes one or more actions. Some examples of abilities include fireball, punch, shooting an arrow and so on. Modifiers serve as a universal method of modifying characters properties.

4.5.1 Input

Before anything else, there must be something that triggers the ability, an input. Abilities are requested by attaching an Ability Request component to an ability definition entity. Even though attaching a component requires synchronisation, its flexibility is more important in this case. The component makes it straightforward to implement any trigger as a source, such as player input or AI input.

4.5.2 Abilities

The ability part of the combat includes all attacks performed by a living character. Examples of such attacks are punch, fireball, shooting an arrow and swinging a sword. I divided this part into three segments: definition, activation, and instance. Segments can only communicate between themselves with events and tag components. Each segment can further be divided into phases, which are not restricting in methods of communication.

Definition segment describes actions and provides logic for ability creation.

Activation segment handles activation of abilities. I further divided it into two phases: validation and activation. Using phases improves performance because segments cannot communicate with component properties and have to use synchronization. Unfortunately, the downside is that each system has to check whether it can use the ability.

The activation phase initializes the ability. It loads all required assets and applies properties from the definition to the instance. Applying properties

from definition allows varying strength of each ability between characters and even using various other properties such as character's strength.

Finally, the instance segment contains all the logic abilities require after activation. This phase does not have a specific structure, because the logic can vary significantly from ability to ability.

4.5.3 Modifiers

A combat system that affects only health can be rather dull. To make it more interesting, I decided to add the ability to modify any player property. Examples of player properties include energy, movement speed, and strength. There are many possible problems with changing these values. For example, if the change is temporary, how will multiple changes interact with each other, how to avoid code duplication when these values represent different properties. To solve these problems, I designed modifiers.

The modifier system is a unified system for modifying any floating-point value. Modifiers operate on entities exclusively designated to them. There are five main types of operations (listed in order) addition, multiplication, clamping[1], absorption[2], and override. Operations modify subtotal component attached to the same entity. Unity DOTS guarantees there are no race conditions — operations are executed in order.

After the update, the last executed operation system stores the result in the subtotal component and cleanup system runs. After the execution, the cleanup is performed to ensure that long frame times do not reduce the actual value over time. Specific systems use the result to apply it to the appropriate property inside a primary entity component.

Modifiers are very versatile and can be used for all kinds of things, including abilities, equipment effects and zone effects. Having all these effects on many different entities updated every frame can potentially lead to performance problems. I decided to split permanent and temporary modifiers to ease this issue. Permanent modifiers are only updated when change occurs; this has a performance penalty on change, but it increases overall performance if these changes are not very common. On the other hand, temporary modifiers are updated every frame. However, they have no significant performance penalty when adding or removing a modifier.

[1] Clamping is a process of limiting value to a minimum and maximum bound.

[2] Absorption reduces the magnitude of a value by absorption amount.

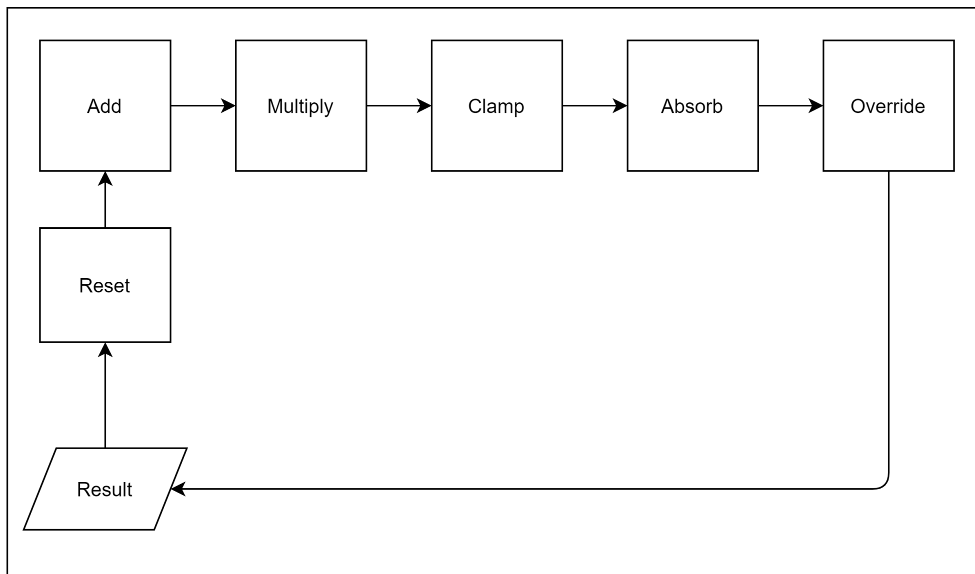


Figure 4.3: Modifier architecture

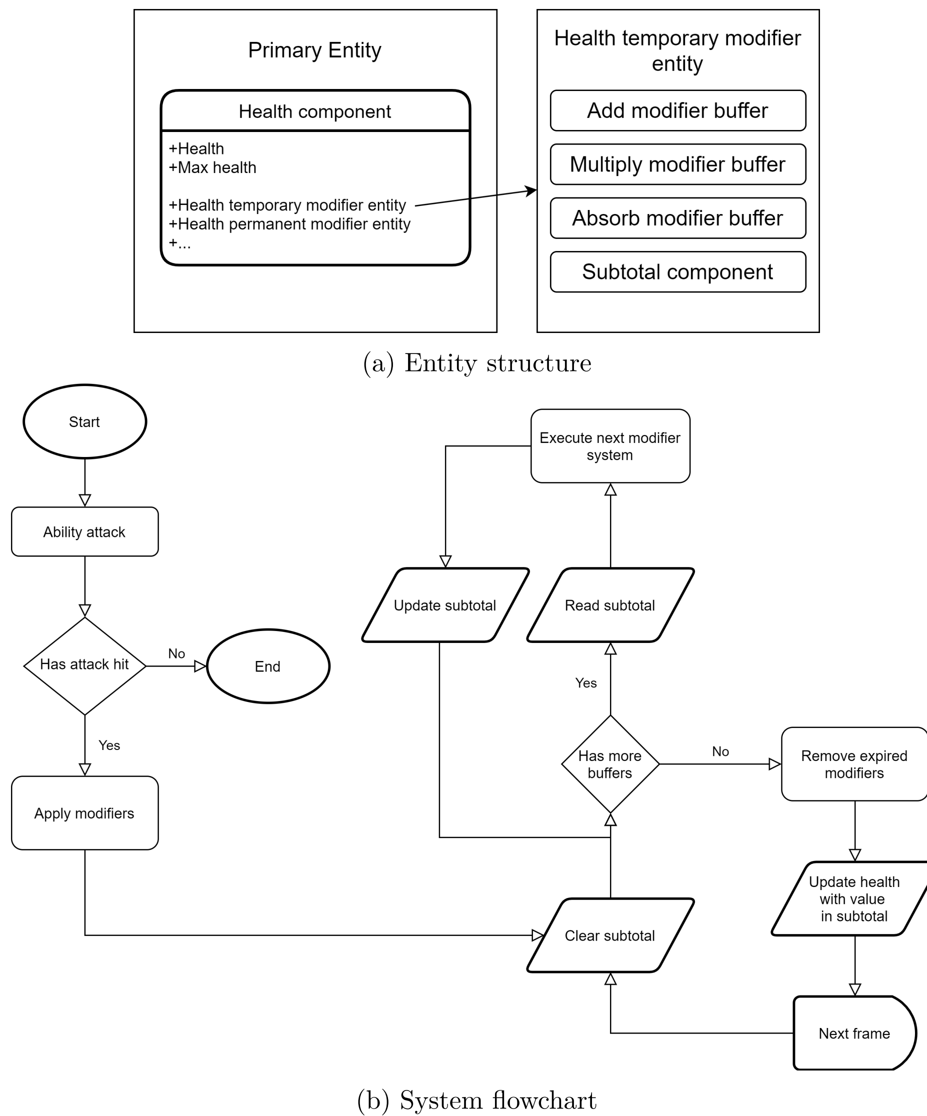
4.5.3.1 Example

In the following example, I will look at how modifiers work on the health property. See Figure 4.4 for visual representation of the following description.

There are two significant entities in this example—a primary entity and a temporary modifier entity. The primary entity has a health component attached to it, which contains current health value, maximum health value, temporary modifier entity and permanent modifier entity. The temporary modifier entity has a buffer for additive, multiplicative, and absorb modifier operations and a subtotal component.

At the start, there is already an activated ability that has modifiers attached to it. All modifiers on the ability are temporary and only affect current health. This ability is executing an attack in the current frame. If the attack hits, it applies modifiers to all entities it hits—in this example, to the Health temporary modifier entity. Because no entity structures are modified, the system can immediately take the new modifiers into account without thread synchronization. Entities package ensures there are no race conditions.

During the update phase, the system executes the operations in order, reading subtotal at the start and saving its modified value at the end. Because this entity only has Add, Multiply, and Absorb buffers, only their respective systems will run. After running the final update system, the cleanup system removes all expired modifiers. A specified system then updates the health value in the Health component attached to the Primary Entity. Finally, the modifiers wait for the next update to update the values again.



(a) Entity structure

(b) System flowchart

Figure 4.4: Example of the modifiers on a health property.

4.5.3.2 Equipment

All characters have several equipment slots. The most critical slot is the hand slot, which contains a weapon. Weapons are the source of abilities. A character can change available abilities by swapping weapons.

4.6 Artificial Intelligence

Excellent AI enhances players experience with fun challenges. On the other hand, inadequate AI only frustrates them. In this prototype, AI serves as

an opponent and makes the world feel more alive. Machine learning AI is often considered the current peak of the AI. However, creating good machine learning AI is extremely difficult. Instead, I chose to create an AI which has behaviours and switches between them based on conditions. This method retains much control over the AI and is very extendable. The only downside is the theoretical performance penalty with many behaviours, but this can be solved, by smarter evaluation of only behaviours that have a chance to be selected.

4.6.1 Groups

The groups help define the basic structure of the AI and its execution order. AI needs to gather information, process it, decide what to do next and finally do it. For each of these parts, there is a group. All the sensors that gather information are executed in the Sensors group. The information from the sensors is then processed to more advanced formats inside the Pre-Processing group. Behaviours report how much they want to run in the Behaviour selection group, which then chooses the most eager behaviour. Lastly, the selected behaviour is executed in the Behaviour execution group. Inactive behaviours are also allowed to execute their logic in the Behaviour execution group, but they must not change the state of the character. For example, they can update internal information based on sensors data, but cannot move the character. The Figure 4.5 shows the order of execution in a visual form.

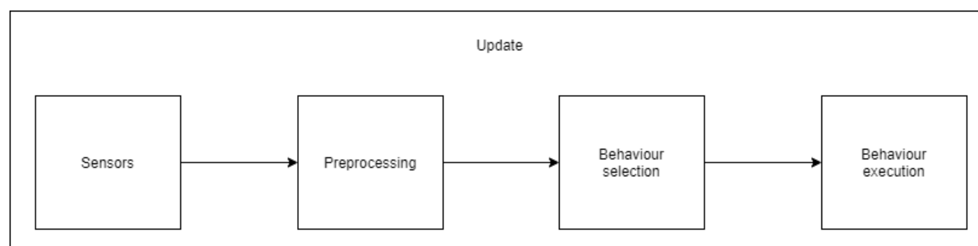


Figure 4.5: The order of AI group execution

4.6.1.1 Sensors

Sensors gather the information that the AI uses for its decisions. The most common example of a sensor is vision or enemy awareness. The difference between vision and enemy awareness is that vision does not penetrate walls, while enemy awareness does. Vision is a handy tool for detecting the player; however, it is not as handy for navigation. Hearing and smell are challenging to implement, but in the future could provide unique game mechanics. Example of a much more common sensor for a game can be danger awareness. Danger

awareness provides AI with specific information about abilities in space, which can then be used by the AI to evade them.

These use cases are essential for enabling design freedom in content. Enabling any sensor requires sensors to be independent of each other; for this purpose, each sensor will have one or more buffer containing information. The sensors systems run on entities with these buffers to update them every update.

For example, the visual sensor consists of two buffers. First contains a list of all the visual sensors the character has (the character might have two eyes that can see to the sides but not in front). The second buffer contains a list of all entities it can see.

4.6.1.2 Behaviour selection

Behaviours are selected based on two properties: priority and value. The priority describes how important behaviour is. For example, when in combat, AI should not decide that it is time to eat. To avoid this, the priority for combat must be higher than hunger. The second property, value, describes how much it wants to be activated. For example, when the character is more thirsty than hungry, the thirst will have higher value and will be chosen.

Each behaviour can use sensor results and properties to calculate these values. The selection system chooses the behaviour with the highest priority as the active behaviour. Suppose there are multiple behaviours with the same priority. In that case, the system compares values and chooses the system with the highest value. If multiple behaviours share the same selection properties, only one of them is chosen as active.

4.6.1.3 Behaviour execution

The final group is also the most loosely defined. It executes behaviour logic. Behaviours are free to decide if they want to run only when active or execute some logic every update.

4.6.2 Daily Routines

Daily Routines give non-player characters something to do when they have no other task. Even if the NPCs only walk around, it makes the world seem livelier than if they stand somewhere, waiting for the player or decorating the street. Because in this prototype, they are not a vital gameplay feature, I chose to make them more straightforward. The NPCs walk between a list of checkpoints and wait at each checkpoint for a specified amount of time. Routines have an essential prerequisite, pathfinding.

Pathfinding falls under requestable behaviours because it responds to event entities. Because the terrain is not yet convertible to DOTS, it is best to use the built-in NavMesh generation with pathfinding. Built-in NavMesh support

will make things much more manageable and solve many challenging problems that pathfinding on meshes imposes.

The routines are activatable behaviour. They need to report their priority, which should be low, to the selection buffer every update. Once they are active, they need to create a pathfinding event for a path to the next checkpoint. When the path is available, they need to follow it until they reach the checkpoint and repeat the process.

Because the characters are a decoration, there is an early optimization that can improve performance. Because the paths are three-dimensional and avoid obstacles, the characters do not need to use physics and can follow the path at a certain speed.

4.7 Character entity

To make a character structure clear and avoid mistakes, I created a diagram that shows how the AI character's entity structure looks. The diagram is in Figure 4.6.

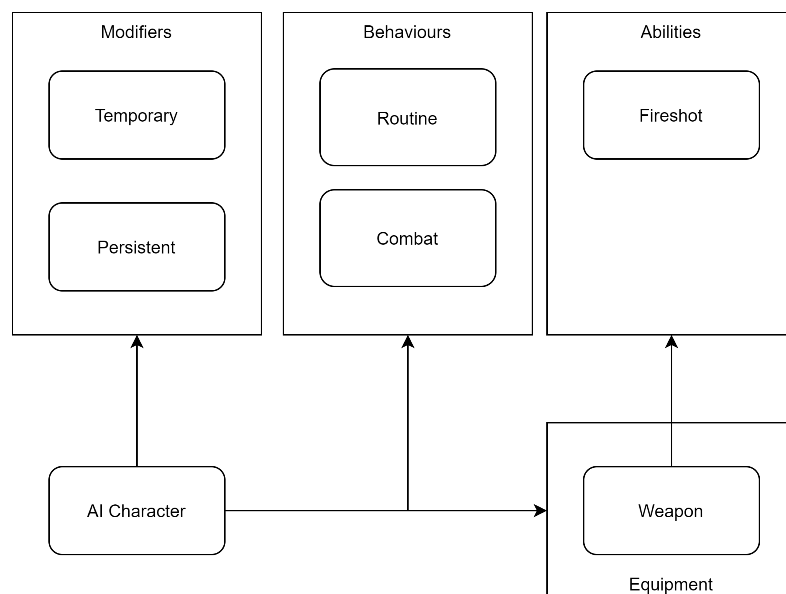


Figure 4.6: Entity structure of an AI character. The player's character has a similar structure. The primary difference is that it does not have behaviours.

4.8 System communication

In an ideal world, systems would not directly depend on each other and could be freely replaced. ECS makes this simpler to implement because components

are the only thing that connects different systems. Unfortunately, when a component changes, it might affect systems that rely on it. To properly understand what important parts rely on, I created a diagram that shows direct dependencies of various parts. The diagram is in Figure 4.7.

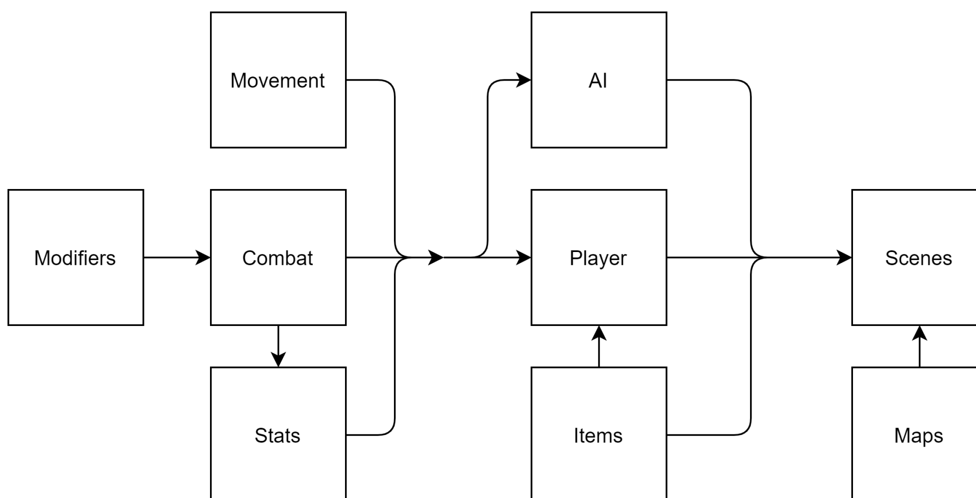


Figure 4.7: Dependency of important parts

4.9 User interface

This prototype's user interface needs to display only a few essential pieces of information—the health, energy, and items. It also needs to allow the player to control the character. The bottom of the screen contains interactive elements to make it easier to control the game. The top of the screen primarily contains informative elements, because fingers will not obstruct them.

Since players can only move in this prototype, the only interactive element is the movement control—a floating joystick. Compared to buttons and standard joystick, the floating joystick is the most comfortable to use. The difference between standard and floating joystick is that standard joystick has a static centre position. In contrast, the floating joystick has a centre position set to the first position the player touches in a selected area.

The most significant informative elements in this prototype are health, energy, and items. Many games have health in the top left corner and other information in the top right corner. It would be counter-intuitive for the user to design it any other way. The health and energy are in the top left corner. The items are a vertical list of icons, displayed from the top right corner downwards.

The Figure 4.8 contains a wireframe of this design.

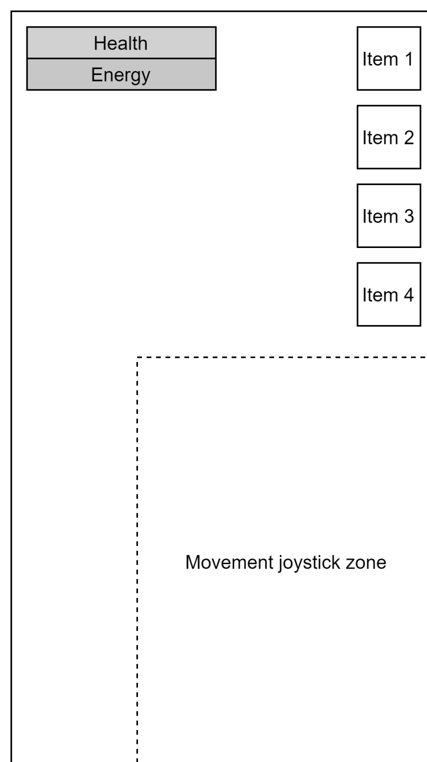


Figure 4.8: Wireframe of the user interface

CHAPTER 5

Realisation

The realisation of the prototype was more complicated than it might seem at first glance. Because DOTS is in such an early stage of development, the documentation is lacklustre, many features were not yet implemented, and some features were broken. The community is also much smaller and not that well versed compared to standard Unity development. In the following chapters, I will look at how I implemented the design.

5.1 Code editors

Unity supports several code editors including Visual Studio Code, Visual Studio 2019, and Rider. From the three Rider was by far the best option for Unity development. Visual Studio Code lacks many features which more robust IDEs like Visual Studio 2019 and Rider have. Visual Studio 2019 is a much more robust editor, but unfortunately, it is not a very performant option. It was very slow and often unresponsive for several seconds. Rider, while not perfect, provided the best working experience. It has a Unity plugin developed by JetBrains which supports Entities package. Early in the development, there were performance issues with Entities ForEach function, because it has hundreds of different overloads. Luckily, JetBrains resolved this issue in a later release. It suffers from occasional disconnects from Unity, but over time these became less common.

5.1.1 Movement systems

Because the movement systems use physics, they are executed in a fixed-step group. If the movement systems were executed outside of this group, they would create weird artefacts, and they would not correctly work as there might be either no updates or multiple updates per physics step. However, systems that show the result to the player need to be updated each frame. In

this case, the Animation System. Figure 5.1 shows the order of execution of each group.

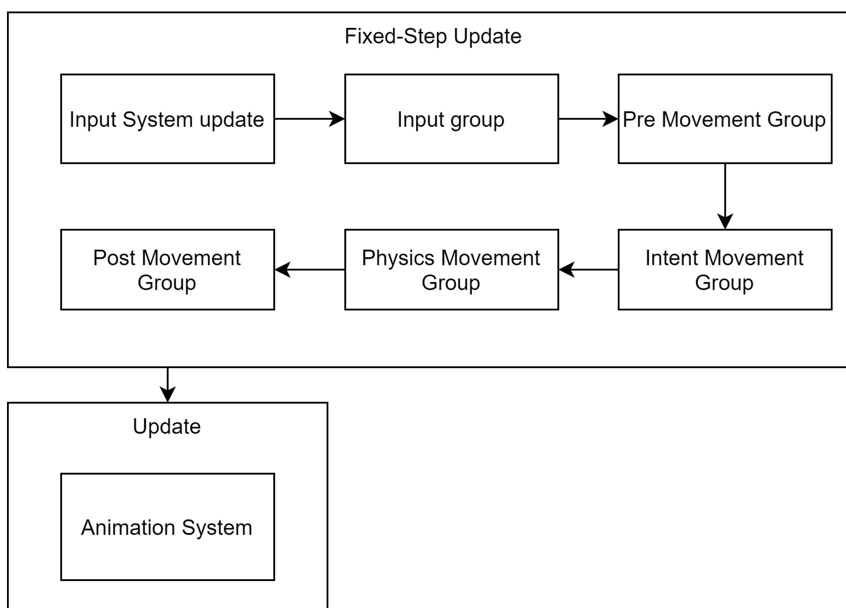


Figure 5.1: The order of execution of the movement systems

A system manually triggers the Input System update at the start of the fixed-step group. Manual trigger ensures that all inputs are correctly registered. Unfortunately, it also requires every system that reads input to be executed in the fixed-step group.[1]

Sometime after the Input System update, the Input system reads the input and sets its values into an input intent component. This component is then accessed by pre-processing components that prepare essential information for the Movement itself, such as the height of the character and initialization of data required for the movement calculations.

The physics movement group systems are responsible for calculating the final velocity vector, which the physics system uses. The group contains two primary systems — the Grounding system (Figure 5.3) and the Slope and Step calculation system (Figure 5.2). The Grounding system keeps the player on the ground while walking down a slope or stairs. The Slope and Step calculation system has two roles. The computationally more demanding role is to calculate the speed at which the character should climb the slope. The second role sets the y coordinate (height) to the y coordinate of the next expected position. The coordinate is obtained at no extra computational cost

[1]The Input System update is costly on mobile devices and during debugging took one millisecond each frame to process. Unfortunately, there were no better solutions at the time of writing.

because the system already knows it from the first role. In the future, the Slope system, and the Step system could be improved by only using velocity.

```
var physicsWorld = _physicsData.BuildPhysicsWorld.PhysicsWorld;
var fixedDeltaTime = Time.fixedDeltaTime;
Entities
.WithoutBurst()
.ForEach(
  (
    ref PhysicsVelocity physicsVelocity,
    ref Translation translation,
    ref DirectionalMoveData directionMoveData,
    in ExtraMoveData extraMoveData,
    in BaseMoveData baseMoveData
  ) => {
    var velocity2D = physicsVelocity.Linear.xz;
    if (velocity2D.IsZero()) {
      return;
    }
    var futurePosition = PhysicsMath.NextPosition(
      translation.Value,
      physicsVelocity.Linear,
      fixedDeltaTime
    );
    var normalizedVelocity = math.normalize(velocity2D);
    futurePosition.x += normalizedVelocity.x * extraMoveData.Radius;
    futurePosition.z += normalizedVelocity.y * extraMoveData.Radius;
    if (EntityPhysics.GroundRayCast(
      in physicsWorld,
      in futurePosition,
      extraMoveData.HalfHeight,
      0,
      out var futureResult
    )) {
      var angleStartPosition = EntityPhysics.GroundRayCast(
        in physicsWorld,
        in translation.Value,
        extraMoveData.HalfHeight,
        extraMoveData.HalfHeight,
        out var feetResult
      )
        ? feetResult.Position
        : translation.Value;
      var angle = angleStartPosition.AngleYTo(futureResult.Position);
      float velocityPercentage;
      if (angle <= baseMoveData.MaxFullSpeedSlope) {
        velocityPercentage = 1f;
      } else if (angle <= baseMoveData.MaxClimbSlope) {
        velocityPercentage = math.sqrt(
          1 -
          (angle - baseMoveData.MaxFullSpeedSlope) /
          (baseMoveData.MaxClimbSlope - baseMoveData.MaxFullSpeedSlope)
        );
      } else {
        velocityPercentage = 0f;
      }
      if (velocityPercentage > 0 && velocityPercentage <= 1) {
        physicsVelocity.Linear.x *= velocityPercentage;
        physicsVelocity.Linear.z *= velocityPercentage;
        translation.Value.y = futureResult.Position.y;
      } else if (velocityPercentage <= 0) {
        physicsVelocity.Linear.x = 0;
        physicsVelocity.Linear.z = 0;
      }
    }
  }
)
.Run();
```

Figure 5.2: Implementation of the Slope and Step calculation system.

```
var physicsWorld = _physicsData.BuildPhysicsWorld.PhysicsWorld;
Entities
.ForEach(
  (
    ref Translation translation,
    ref DirectionalMoveData directionMoveData,
    in PhysicsVelocity physicsVelocity,
    in ExtraMoveData extraMoveData,
    in BaseMoveData baseMoveData
  ) => {
    float maxGroundDistance;
    if (!directionMoveData.IsGrounded && physicsVelocity.Linear.y < 0.0f) {
      maxGroundDistance = InAirGroundCheckDistance;
    } else {
      maxGroundDistance = GroundCheckDistance;
    }
    if (EntityPhysics.GroundSphereCast(
      in physicsWorld,
      in translation.Value,
      AboveFeetCheckDistance,
      maxGroundDistance,
      extraMoveData.Radius / 2f,
      out var result
    )) {
      directionMoveData.IsGrounded = true;
      if (result.Position.y < translation.Value.y) {
        translation.Value.y = result.Position.y;
      }
    } else {
      directionMoveData.IsGrounded = false;
    }
  }
)
.Run();
```

Figure 5.3: Implementation of the Grounding system.

5.2 Asset management

This section will look at how the prototype loads assets and tools needed to manage database data.

5.2.1 Asset loading

Because many systems rely on loading assets such as prefabs and database files, the Addressables are among the first things that I need to implement. Jobs, Burst, and DOTS do not support Addressables. Nevertheless, that does not mean it is impossible to use them while using DOTS.

The most expensive operations in asset loading are loading the asset and creating its instance. Neither of these operations can be executed inside a job or compiled with Burst. Thus, I decided it is better to use class components and non-blittable structs instead of complicated pointer workarounds. This solution might slightly reduce performance, but it will likely be replaced with a DOTS compatible variant in the future.

Addressables are asynchronous, so instead of an asset, they return an asynchronous handle, which contains information about the asset's current loading status and its value. I created a two-part system which handles this task. The

```
Entities
.WithAll<PrefabAsset>().WithNone<LoadHandle>()
.WithoutBurst()
.ForEach(
  (Entity entity, in LoadAsset loadAsset) => {
    var operation =
      Addressables.LoadAssetAsync<GameObject>(
        loadAsset.Path.ToString()
      );
    commandBuffer.AddComponent(
      entity,
      new LoadHandle {Value = operation}
    );
  }
)
.Run();
```

Figure 5.4: First part of asset loading system.

first part (Figure 5.4) sends a request to load the asset and adds a component handle to the request entity. The second part (Figure 5.5) checks each frame if the loading has finished. If so, it removes the request components and adds a result component containing the loaded asset. Keeping the same entity ensures, that no request information, such as instantiation[1] tag, is lost. Unfortunately, because Addressables use generics, each type requires its system. It is possible to load any asset as type object; however, this can lead to unpredictable results such as Sprites loaded as Textures.

To instantiate an object, I created a new system that requires Instantiate tag and the loaded asset component. Because Instantiation is synchronous, there is no need for multiple parts.

[1]Instantiation is the process of creating a new instance.

```
Entities
.WithAll<PrefabAsset>()
.WithoutBurst()
.ForEach((
  Entity entity,
  in LoadAsset loadAsset,
  in LoadHandle handle
) => {
  switch (handle.Value.Status) {
    case AsyncOperationStatus.None:
      return;
    case AsyncOperationStatus.Succeeded:
      commandBuffer.RemoveComponent<LoadAsset>(entity);
      commandBuffer.RemoveComponent<LoadHandle>(entity);
      commandBuffer.AddSharedComponent(
        entity,
        new PrefabAssetData {
          Prefab = handle.Value.Result as GameObject,
          Path = loadAsset.Path
        }
      );
      break;
    case AsyncOperationStatus.Failed:
      // Log error
      commandBuffer.DestroyEntity(entity);
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
})
.Run();
```

Figure 5.5: Second part of asset loading system.

5.2.2 Database

During the initialization of the prototype, the database instance is created and stored inside a singleton. Singleton makes it possible to use the database in exclusive mode, which improves performance. Systems can access this instance with GetSingleton method. The database cannot be accessed directly and has to be copied to an accessible location outside of the editor. The database is copied before the initialization of the prototype. In this case, the persistent data path[1] was the most suitable location for the copy. Systems accessing the database are not allowed to use Burst or run outside of the main thread.

[1] Persistent data path provides the location of a directory, where data persistent between runs should be kept. On Android it is resolved by calling getExternalFilesDir.

5.2.2.1 Issues

During design, the database showed great promise in supporting multithreaded access to the data. Sadly, while the database solution makes it possible, Unity needs to perform static code analysis on job code. Because LiteDB is a library, Unity cannot statically analyse the code and throws an error message. The only remaining alternative is to run the database requests code without Burst on the main thread. Fortunately, the database is not accessed often, so this is not much of an issue.

At the moment of implementation, Entities package did not offer any way to run code outside of ForEach only when ForEach is executed at least once. Loading database from singleton and getting a collection takes 1 ms. If it were outside of ForEach, it would reduce the frame budget by 1 ms for each system that needs to do this. To avoid this penalty, I decided to call database requests inside ForEach instead.

5.2.3 Asset management tools

The database comes with the increased difficulty of creating tools for creation. These tools are needed because it is complicated to create database records manually as there are often unseen references to C# classes. For the creation of these tools, I decided to use UI Toolkit and reflection. While reflection will make the initial implementation more complicated, it will make it universal, enabling its use on various classes without new mapping.

The UI Toolkit creates a tree layout and redraws it at its discretion. Updating a node in the layout requires a reference. Keeping a reference would be difficult with reflection, but luckily, I could use events and keep the reference inside them. The classes can be split into their respective fields, thanks to reflection. However, dividing structs[1] is much too complicated and not worth the effort. Each struct needs its implementation, which cannot share code with a different implementation. The code cannot be shared because of type-safety required by generics.

There are only around twenty to thirty structs in use for this prototype's necessary structures, including signed numbers, unsigned numbers, vectors, strings, enums, lists, and arrays. This number of structs is manageable, and field implementations range from a few lines for the most straightforward structures to tens of lines for the more complicated ones. Figure 5.6 shows an example of a straightforward structure.

[1] Primitive types, such as int and float, are also structs in C#.

```
if (fieldType == typeof(int2)) {
  var typedValue = (int2) value;
  result = new Vector2IntField(name) {
    value = new Vector2Int(typedValue.x, typedValue.y)
  }.Also(
    field => field.RegisterValueChangedCallback(
      evt => {
        fi.SetValue(parentObject,
          new int2(evt.newValue.x, evt.newValue.y)
        );
        onValueChanged?.Invoke(field);
      }
    )
  );
}
```

Figure 5.6: Implementation of a 2D vector field using reflection in UI Toolkit.

5.3 Scene management

One of the unexpected problems I encountered was scene management. Unity provides a SceneManager API; however, it only works with GameObjects. When using the SceneManager, it loads a GameObject scene, then the objects with ConvertToEntity component (MonoBehaviour) are converted to entities. The main problem is that the converted entities are not associated with the scene, so when the scene is unloaded, they are not.

My workaround for this issue was attaching a ParentScene component to every entity (Figure 5.7). This component identifies to which scene components belong. When a scene is unloaded, all entities belonging to that scene are destroyed.

I later expanded the scene management to make it possible to load scenes within DOTS and create procedural maps—more about implementing procedural maps in Section 5.6.1.

```
public struct ParentScene : IComponentData {
  public FixedString64 Id;
}

Entities
.WithNone<ParentScene>()
.ForEach(
  (Entity entity, int entityInQueryIndex) => {
    commandBuffer.AddComponent(
      entityInQueryIndex,
      entity,
      new ParentScene {Id = sceneId}
    );
  }
)
.ScheduleParallel();
```

Figure 5.7: Implementation of a ParentScene component

5.4 Physics

While some things have no DOTS packages, physics has two packages: Unity Physics and Havok. For this prototype, I chose Unity Physics because it is open-source. Even though the documentation is lacking, I can at least look at the source code. As of version 0.5.1, Unity has documented only a handful of functions. The primary source of information is a manual and an example project. Because of the poor documentation and occasional bugs, the implementation consisted mostly of trial and error. One of the most common use cases—collision and trigger events—is particularly challenging to do. The easiest way to implement it is to copy implementation from the example provided by Unity.[20]

Sadly, to find a specific collision, each system has to use a linear search to find whether it is present or not. To reduce code duplication, I decided to add a CollidedWithPlayer tag component for often used collision detection with the player.

I encountered an issue with physics using LocalToWorld component.[1] According to documentation, Unity made systems should update LocalToWorld component with translation, rotation, and other relevant components. However, this was not behaviour I observed and instead, I had to set LocalToWorld manually. Resolving this issue was made even more complicated because there were no runtime physics debugging tools at the time of implementation.

[1] LocalToWorld component is a 4y×4 transformation matrix containing position, rotation, and scale information in world coordinates.

5.5 Combat

Combat consists primarily of three parts: the combat system, the modifier system, and the stats. The combat system contains abilities which apply the modifiers. The modifiers define how the stats are modified. The stats describe the characters' properties. In this section, I will look at how I implemented each part in reverse order of execution. Figure 5.8 shows the flow of the combat system.

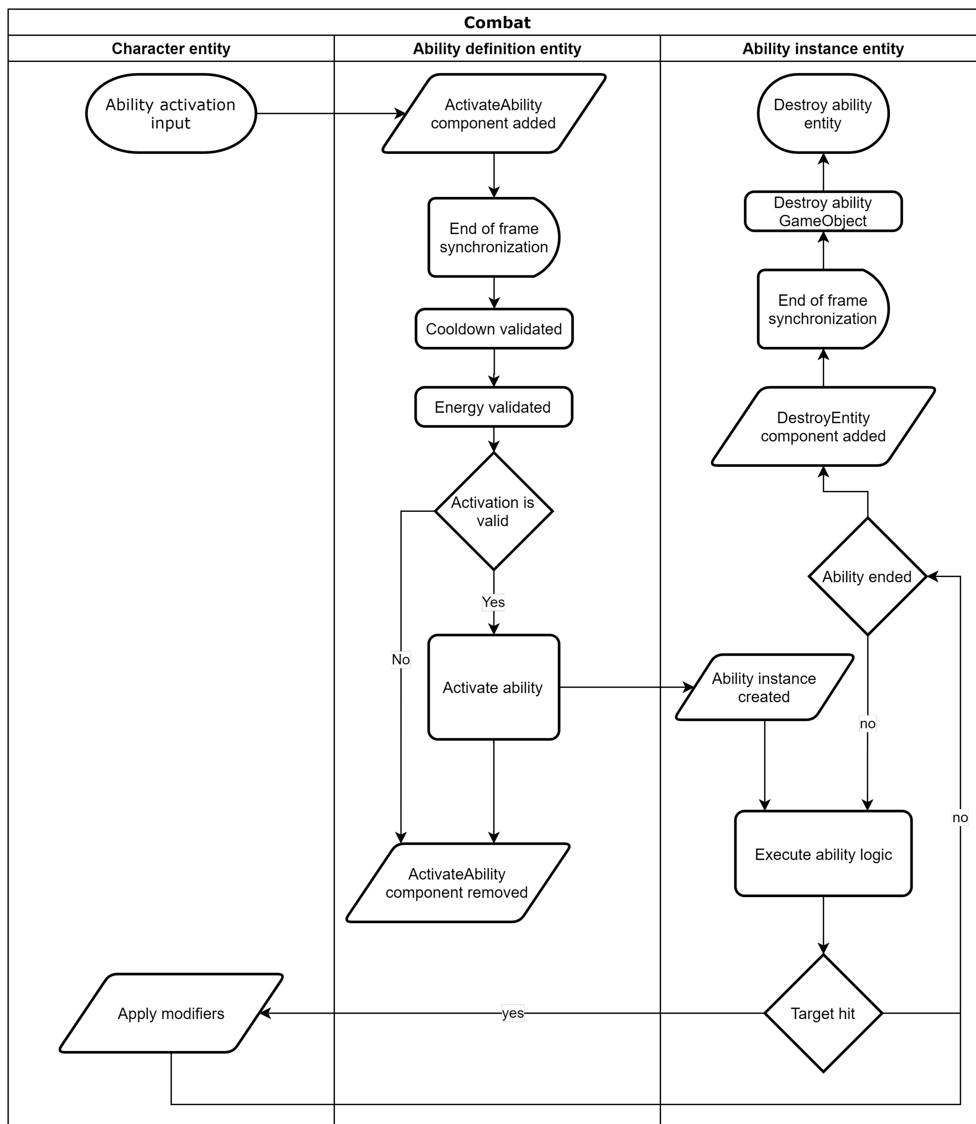


Figure 5.8: Flow of the combat system

5.5.1 Stats

The stats, also known as character properties, describe the properties of every character in-game. It contains attributes (such as strength and agility), health, and energy.

Health, as its name implies, is used to describe characters current health in a measurable way. By design, health should be only modified with modifiers. Health changes are propagated from modifier entities to the primary entity. Various systems can then access health changes within the health component.

Energy follows the same rules as health.

5.5.2 Modifiers

Modifiers serve as the primary logic system for changing properties in a non-destructive way. I implemented them as buffers attached to an entity. Each type of modifier has its buffer and update system. Update systems contain logic that updates the subtotal value with modifier values. There are three types of modifiers from update perspective: always update, update on change and never update. An update tag component makes it possible to differentiate between these three types, making them practically a single type. If the update component is present, update systems calculate a new value based on existing modifiers. Otherwise, no modifier system is executed.

The removal system removes expired modifiers after the update is executed. Executing this system after the update allows update systems to apply the modifier partially. I expect there to be many modifiers that should not be checked for expiration, like template modifiers and permanent modifiers. For this purpose, I added persistent and temporary tags. The removal system checks only temporary modifiers for expiration.

Specialized systems need to apply modifier subtotals (results) because application logic is different for each property. For this purpose, each character modifier entity contains a tag component, specifying to which property it belongs. Implementing it in this way allows application systems to leverage modifier tags and for example, only update, when the update tag is present.

5.5.3 Abilities

Abilities are the final piece of the combat system. Abilities describe what the ability (skill) does, how it looks, and whom it impacts. The implementation has four phases: input, activation (validation, activation), instantiation, behaviour. Each ability is different, but they share these four phases.

5.5.3.1 Input

Everything starts with an input, either from the player or the AI. In the input phase, a system creates an Ability Request component which contains

the entity that requested it, input source (AI or player), position, and forward vector. The input system attaches this component to the ability entity, finalizing the input phase. The activation phase begins the next frame because the component is attached at the end of the frame.

5.5.3.2 Activation

The activation phase consists of two subphases the validation and the activation.

In the validation subphase, various things are validated, such as if the ability is on cooldown and whether there is enough energy to use the ability. During this subphase, reservations ensure that the conditions still hold in the activation subphase. A good demonstration of a possible problem is if two abilities activate simultaneously and the user has energy only for one. If there were no reservations on energy, the user would end up with negative energy. Each validation has its system and optional component, to make it possible to add additional validations without rewriting the existing code in the future.

The activation subphase is executed after the validation subphase. Every ability has its activator system that contains the logic for ability initialization (activation). Unfortunately using a tag component to identify valid requests would impose an unnecessary performance penalty. To avoid this, each system has to check for the result of the activation phase manually. The logic of each ability differs, but certain operations such as copying the modifiers to ability instance or applying cooldown and energy are shared between abilities.

5.5.3.3 Instantiation

The instantiation phase is not an exclusive part of the combat system. However, it introduces a significant delay and performance penalty. It uses asset management systems to load and instantiate abilities. This is a necessary step because the best way—and only advised way—to create physics and graphics objects is through prefabs. The same holds for graphical effects and particle systems. Prefabs can also store shared properties between instances.

In the best-case scenario, the asset management systems can prepare a new instance for ability before the next frame. However, in the worst case, it might even take several frames to do so. I decided not to solve this potential problem in the prototype because it is a theoretical issue. There is one more frame delay after the asset instantiation system creates the instance of the prefab. The activation system copying components with instance-specific data causes this delay. It can be mitigated in the future, by adding necessary components to the prefab and only updating them at runtime with proper values.

5.5.3.4 Behaviour

Behaviour is the most extensive phase because it differs for each ability. It has three parts initialization, active, and application. I do not consider them subphases because they might be executed at the same time. A projectile that hits a target but is not destroyed by it is an excellent example of such a situation.

The initialization part consists of logic that sets up necessary runtime information for the active part. For example, an ability that travels on a curve needs to know its current position on the curve.

The active part contains all logic that decides the movement, shape, and other logic not related to registering an ability hit. Projectiles move using physics, so the movement of the projectile is not an example. Nevertheless, each projectile has a logic that limits its distance and lifetime, primarily to ensure it does not travel for all eternity; such a system is a piece of this part.

The application part contains all logic that detects when it hit a target and what to do afterwards. An example from projectiles is the logic of applying modifiers when a projectile hits the player.

In some cases, one system can contain more than one part.

Examples in Figure 5.9, Figure 5.10 and Figure 5.11 show all three parts on a projectile ability.

```
Entities
.WithAll<Projectile>()
.WithNone<ProjectileRuntime>()
.ForEach(
  (
    Entity entity,
    ref Rotation rotation,
    in Projectile projectile,
    in Translation translation
  ) => {
    commandBuffer.AddComponent(
      entity,
      new ProjectileRuntime {
        LastPosition = translation.Value
      }
    );

    rotation.Value =
     quaternion.LookRotationSafe(projectile.Direction, math.up());
  }
)
.Schedule();

Entities
.WithNone<ProjectileRuntime>()
.ForEach(
  (
    Entity entity,
    ref PhysicsVelocity velocity,
    in Components.General.Projectiles.Projectile projectile
  ) => {
    velocity.Linear =
     math.normalize(projectile.Direction) * projectile.Speed;
  }
)
.Schedule();
```

Figure 5.9: Implementation of the initialization part of a projectile behaviour.

```
Entities
.ForEach(
  (
    Entity entity,
    in Components.General.Projectiles.Projectile projectile,
    in ProjectileRuntime runtime
  ) => {
    if (projectile.MaxDistance <= runtime.DistanceTravelled ||
      projectile.MaxAge <= runtime.TimeElapsed) {
      commandBuffer.AddComponent<DestroyEntity>(entity);
    }
  }
)
.Schedule();
```

Figure 5.10: Implementation of the active part of a projectile behaviour.

```
var elapsedTime = (float) Time.ElapsedTime;
Entities
.WithAll<Components.General.Projectiles.Projectile, ModifierAddValue>()
.ForEach(
  (Entity entity, in CollidedWithPlayer hit) => {
    var health = GetComponent<Health>(hit.PlayerEntity);
    var sourceBuffer = GetBuffer<ModifierAddValue>(entity);
    var targetBuffer =
      GetBuffer<ModifierAddValue>(health.TemporaryModifierEntity);
    var targetArray =
      new NativeArray<ModifierAddValue>(sourceBuffer.Length, Allocator.Temp);
    for (var i = 0; i < targetArray.Length; i++) {
      var item = sourceBuffer[i];
      item.Modifier.ActivatedAt = elapsedTime;
      targetArray[i] = item;
    }
    targetBuffer.AddRange(targetArray);
    targetArray.Dispose();
  }
)
.Schedule();
```

Figure 5.11: Implementation of the application part of a projectile behaviour.

5.6 Items

Items are objects that players can use. In the prototype, there are two types of items: keys and weapons. Only keys have a visual representation as weapons are not obtainable by the player. I have chosen to use sprites to represent keys as they can be used in the game world as well as the user interface. Each item's definition is stored inside the database (as class objects) and loaded on demand. Because it is impossible to use LiteDB inside jobs, the items are loaded synchronously on the main thread. Load systems automatically load items tagged with LoadDatabase component from the database. Each type of item needs its system. There is only one type of item loaded from the database, so it would not be worth implementing a robust loading system from the start. Each item type needs its system to convert data from a database format to components (Figure 5.12).

Each item has two core components: Item and ItemText. The Item component identifies each item with a unique identifier. Title and description are inside the ItemText component. Specific to keys is an ItemIsKey component that identifies them as keys and provides the key's value.

Each character has multiple slots to equip items. The only used equipment slot is the hand, which contains weapons. Weapons provide their owners with abilities making them essential for combat. Even though players cannot use weapons in the prototype, the combat system can accommodate player combat in the future without many changes.

I implemented weapons as standard items with extra components, which provide information about available modules. Modules contain a reference to ability entity. When a change occurs, equipment changed tag component is added to alert the rest of the systems about this change. This tag allows other

systems to update computed information from the equipment; for example, abilities are cached on the primary character entity and need to be updated when change occurs.

The keys have a specialized system to load them from the database when needed. The only required information to load a key from the database is a position, rotation, and item identifier. Every obtainable item spawned in the world requires the item definition and needs a physical representation and physics collider. A system can create the physics collider, but sprite needs to be loaded from a prefab, because of the Hybrid Renderer. The logic for picking up an item is straightforward, as I prepared everything necessary in earlier sections. The pick-up system only needs to require CollidedWithPlayer and ObtainableItem components to know that it should add an item to the inventory. For extra versatility, I decided to make this into an event (Figure 5.13). Event entity allows multiple systems to know about item pick-up. Change event becomes especially handy for the user interface. Unlocking is implemented similarly with extra validation if a player has all the required keys (Figure 5.14).

```
var database = GetSingleton<GameDatabase>().Value;
var collection = database.GetCollection<IDatabaseItem>();

Entities.WithoutBurst().WithAll<LoadDatabase, ItemIsKey>()
.ForEach((
  in Item item,
  in Translation translation,
  in Rotation rotation
) => {
    var itemId = item.Id.ToString();
    var result = collection.FindOne(x => x.Key == itemId);
    if (result is KeyItemDatabase databaseRecord) {
      var entity =
        commandBuffer.CreateEntity(_itemArchetype);
      // Set item components
      var iconPath = databaseRecord.Icon?.Addressable;
      if (iconPath != null) {
        var iconEntity =
          commandBuffer.CreateEntity(_iconArchetype);
        // Set icon components
      }
      var obtainableEntity =
        commandBuffer.CreateEntity(_genObtainArchetype);
      commandBuffer.SetComponent(
        obtainableEntity,
        new GenerateObtainable {
          Size = new float3(1) // Hardcoded size of 1x1x1
        }
      );
      commandBuffer.SetComponent(
        obtainableEntity,
        new ObtainableItem {
          ItemEntity = entity
        }
      );
    }
})
.Run();
```

Figure 5.12: Implementation of loading key item from database.

```
private EntityArchetype _eventArchetype;

protected override void OnCreate() {
  base.OnCreate();
  _eventArchetype = EntityManager.CreateArchetype(
    typeof(ItemPickedUp),
    typeof(GameEvent)
  );
}

protected override void OnUpdate(
  EntityCommandBuffer commandBuffer
) {
  var eventArchetype = _eventArchetype;
  Entities
  .ForEach((
     Entity entity,
     in CollidedWithPlayer collision,
     in ObtainableItem obtainableItem
  ) => {
    var eventEntity =
      commandBuffer.CreateEntity(eventArchetype);
    commandBuffer.SetComponent(
      eventEntity,
      new ItemPickedUp {
        ItemEntity = obtainableItem.ItemEntity,
        OwnerEntity = collision.PlayerEntity,
        TriggerEntity = entity
      }
    );
  })
  .Schedule();
}
```

Figure 5.13: Implementation of a pick-up event creation system with event entity archetype definition.

```
Entities
.ForEach((
  Entity entity,
  in CollidedWithPlayer collision,
  in DynamicBuffer<UnlockRequirement> requirementBuffer,
  in UnlockEntity unlockEntity
) => {
   var playerInventory =
    GetBuffer<InventoryItem>(collision.PlayerEntity);
   if (playerInventory.Length < requirementBuffer.Length){
     return;
   }
   var foundKeys = 0;
   for (var i = 0; i < requirementBuffer.Length; i++) {
     var found = foundKeys;
     var requirement = requirementBuffer[i];
     for (var j = 0; j < playerInventory.Length; j++) {
       var playerItemEntity = playerInventory[j].Entity;
       var isItemKey =
        HasComponent<ItemIsKey>(playerItemEntity);
       if (!isItemKey) {
         continue;
       }
       var keyComponent =
        GetComponent<ItemIsKey>(playerItemEntity);
       if (requirement.Value == keyComponent.Value) {
         foundKeys++;
         break;
       }
     }
     if (found == foundKeys) {
       break;
     }
   }
   if (foundKeys == requirementBuffer.Length) {
     var ueRef = unlockEntity.Entity;
     commandBuffer.AddComponent<DestroyEntity>(ueRef);
     commandBuffer.AddComponent<DestroyEntity>(entity);
   }
})
.Schedule();
```

Figure 5.14: Unlock system implementation.

5.6.1 Map generation

During the implementation of the procedural generation, I experimented with various techniques. The idea was to decide on one implementation from room generation and one implementation for path generation to demonstrate the modular generation system.

5.6.1.1 Room generation

The tested techniques for room generation include random DFS, cellular automaton and other methods derived from these two. The result uses BFS with a configurable chance of expanding to a cell. While this is not the best method, it provides suitable areas for testing various other systems, like combat.

The room generator saves each cell into a list and calculates a room's bounding box. The bounding box is useful to speed up room position detection. After all the rooms are generated, an algorithm uses a bounding box to identify possible intersecting rooms and then validates it with more expensive cell to cell comparison. Merging rooms is beneficial because the original algorithm may sometimes generate overlapping rooms that could mess up with spawning that relies on room definitions. It also reduces the need to check for paths in pathfinding phase.

5.6.1.2 Path generation

Path techniques include drawing a line and various pathfinding algorithms. I selected an algorithm that uses A* pathfinding because it offers the best quality/performance ratio. The algorithm guarantees that rooms stored directly after each other (index i and i+1) are always connected. Other paths have a configurable probability of appearance. When the algorithm creates a path from room A to room B, it first selects a random cell from each room (cell A and B for each room, respectively). After that, it uses my own A* implementation to find a path to a cell in room B.[1] The distance from cell A to cell B is calculated using Manhattan distance. The final path is reversed, so it is from start to end, rather than from end to start.

The algorithm selects a random cell because finding the closest cell is a costly operation. During testing, it took 100 milliseconds for each path to find the closest point. One of the perks of A* is that it can be easily altered, for example, by adding noise penalties to the pathfinding map, it can produce more curvy lines.

[1] A* is a popular graph search algorithm. It is a modification of Dijkstra's Algorithm optimized for a single destination.[21]

```
public int2 RequestSpawnPosition(
  ref Random random,
  SpawnMemory memory,
  int[,] cells,
  IReadOnlyList<DungeonSegment> segments,
  MapSpawnData spawnData
) {
  if (memory.SpawnPositions.Count >= cells.Length) {
    throw exception, no possible spawn positions
  }
  // Check positions until an empty one is found
  while (true) {
    var segmentIndex = random.NextInt(segments.Count);
    var segment = segments[segmentIndex];
    var positionIndex =
      random.NextInt(segment.Cells.Length);
    var position = segment.Cells[positionIndex];
    // Test if position is taken
    if (!memory.IsPositionTaken(position)) {
      // Record position as taken
      memory.AddPosition(position);
      return position;
    }
  }
}
```

Figure 5.15: Implementation of the random spawner.

5.6.1.3 Spawning

The spawner first selects x and y coordinate at random. Then it checks if the coordinate is empty if it is it returns that coordinate. Otherwise, it tries a different coordinate. The spawning relies on the fact that the random number generator eventually tries every possible combination, so there are no safety measures not to repeat coordinates. In the scenarios that can occur in the prototype, this is not an issue. The probability of hitting taken coordinate is very low.

5.6.1.4 Presenter

The presenter has several variations in the prototype. Before the editor was ready for use, I tested map generation algorithms inside sprites using Sprite presenter. It converted cell maps to textures and added them to Image component of Unity UI (uGUI). When I finished the map editor, I replaced it with Texture presenter and added map preview to the editor.

At first, I generated the playable areas with a Terrain presenter. However,

this proved to be a problematic solution as the terrain cannot create 90-degree slopes and has trouble drawing textures correctly on a steep slope. I could solve the texture problem with tri-planar mapping, but that would introduce a significant performance penalty. Instead, I decided to implement a Mesh presenter.

The Mesh presenter generates a three-dimensional model of the terrain. I used the core of the Mesh presenter from Joseph Hocking's article about Procedural Generation Of Mazes With Unity.[22] The algorithm was modified to use Unity Mathematics and extended to generate a physics collider.

Unfortunately, there is a problem with recalculating normals. After investigating the issue, I determined that fixing it for the prototype would take much time and make the code more complicated. A likely cause of this problem is in the Hybrid Renderer library although I could not verify it. Since this issue is more of an annoyance than gameplay breaking problem, I decided not to fix this issue as it could be just a bug inside Unity. This problem causes some vertexes to have randomly flipped normals. The built-in RecalculateNormals method, which should fix these problems has no effect.

An example of a part of mesh presenter is in Figure 5.16.

```
var dimensions = new int2(cells.GetLength(1), cells.GetLength(0));

// normalize cells to a format that can be understood by the generator
var normalizedCells = new int[dimensions.y, dimensions.x];
for (var y = 0; y < dimensions.y; y++) {
  for (var x = 0; x < dimensions.x; x++) {
    normalizedCells[y, x] = cells[y, x] > 0 ? 0 : 1;
  }
}
var meshResult = FromData(
  normalizedCells,
  _configuration.MapScale,
  _configuration.TerrainHeight
);
meshResult.Mesh.LowPolify();
var renderMesh = new RenderMesh {
  material = new Material(Shader.Find("Universal Render Pipeline/Lit")),
  mesh = meshResult.Mesh
};
commandBuffer.AddSharedComponent(entity, renderMesh);
commandBuffer.AddComponent(
  entity,
  new RenderBounds {
    Value = renderMesh.mesh.bounds.ToAABB()
  }
);
using var colliderVert =
  new NativeArray<float3>(meshResult.Vertices, Allocator.Temp);
using var colliderTris =
  new NativeArray<int3>(meshResult.Triangles, Allocator.Temp);
commandBuffer.AddComponent(
  entity,
  new PhysicsCollider {
    Value = MeshCollider.Create(
      colliderVert,
      colliderTris,
      new CollisionFilter {
        BelongsTo = (uint) CollisionFilterCategory.StaticEnvironment,
        CollidesWith = (uint) (
          CollisionFilterCategory.DynamicEnvironment |
          CollisionFilterCategory.DynamicCharacters |
          CollisionFilterCategory.Player
        )
      }
    )
  }
);
```

Figure 5.16: Part of Mesh presenter implementation.

5.6.1.5 Editor

An editor is a vital part of the procedural map generation. Without it, it would be tough to design a map. The map editor (also known as level creator) generates all editable fields using reflection. To update the editor, only fields in the data file need to be changed. The GUI will automatically adjust itself. Section 5.2.3 has more information about the automatic field generation. Without visual feedback, an editor would not be as useful, so the editor also generates an example of a map generated with the current configuration. The Texture presenter generates the preview; it does so asynchronously to avoid editor UI freezes. Figure 5.17 shows the GUI of the map editor.

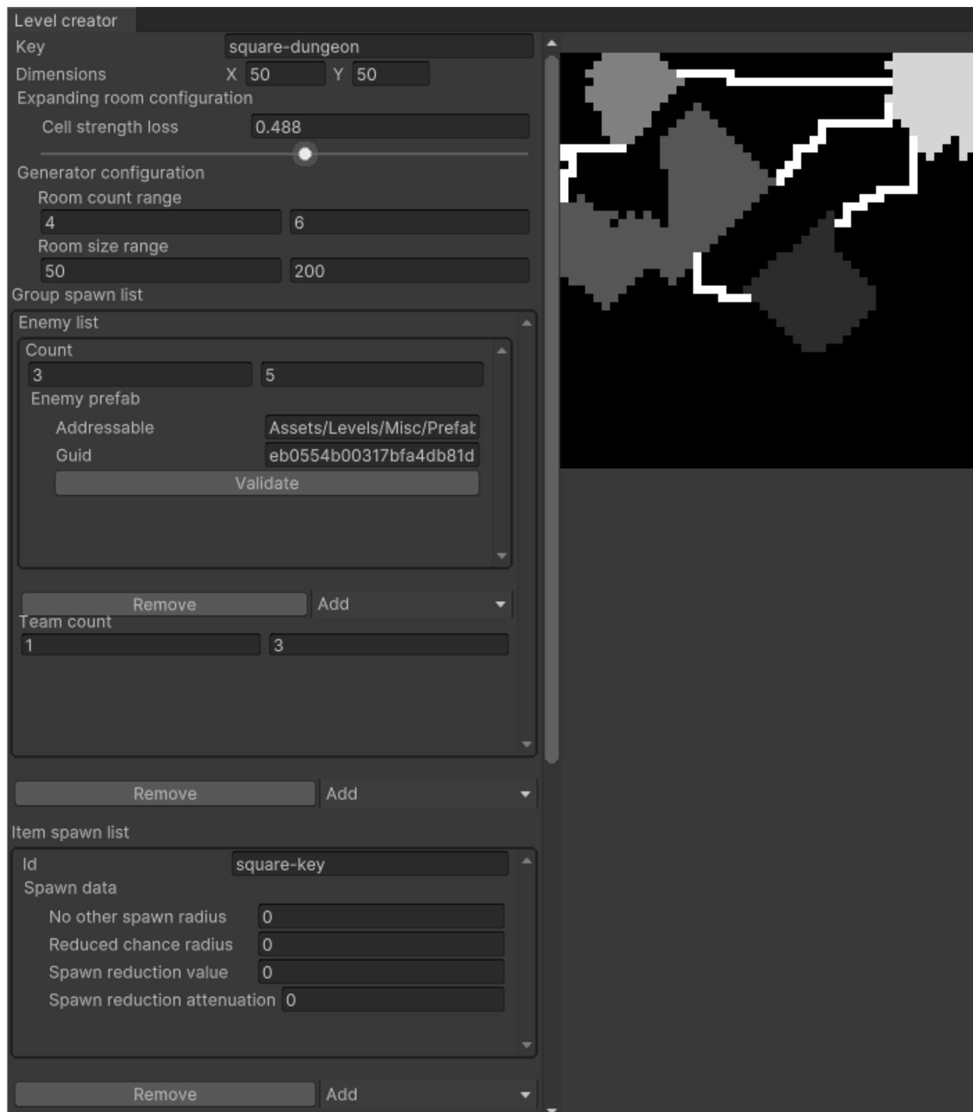


Figure 5.17: Procedural map editor

5.6.1.6 Performance

The map generation algorithm is high-speed. Even in the editor with debug mode enabled[1]. It is generated in several hundred milliseconds, making the loading times of new areas very fast.

[1]Editor's Debug mode significantly reduces the performance and can often result in worse performance than on mobile devices.

5.7 Artificial intelligence

The implementation of AI (artificial intelligence) consists of two main parts: sensors and behaviours. Sensors provide AI with information about its surroundings, and behaviours tell AI what to do based on this information.

5.7.1 Sensors

A component defines each sensor. It specifies its properties and makes it configurable. I chose the visual sensor for the demonstration of what sensors do.

The visual sensors definition component contains information such as field of view, distance, and filter. The filter improves the performance of the sensor by ignoring things in which the sensor is not interested. The visual sensor does not work like human eyes. Instead, it casts a sphere with a diameter equal to the size field of view has at the end of the visible distance. This was the best method, but there are two problems with it. The first problem is that the sphere will detect even objects outside of the field of view. The second problem is that it will detect even objects hidden behind other objects.

The first problem can be solved by validating if the object is within the FoV (field of view). I simplified the implementation to improve performance, so only the “collision point” is checked. As a result, some objects might be partially inside the FoV, but the algorithm will incorrectly discard them. However, solving this edge case would result in too high of a performance penalty.

The second problem can be solved similarly by sending a ray to validate if nothing obscures it from issue. It suffers from a similar problem as a single small object could obscure the ray, and the object would be incorrectly discarded. Luckily, the level design is done in a way that makes this problem very unlikely.

Visual two-dimensional representation of both of these problems is in Figure 5.18.

One final addition to the sensors is the processing of data provided by them. Working with raw data can be difficult and might become expensive with multiple behaviours. For that purpose, common use cases, such as calculating threat each visible character poses, are computed by a dedicated system.

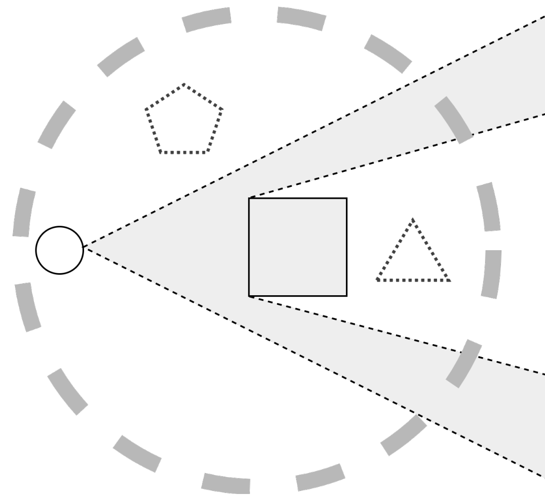


Figure 5.18: Example of two problems caused by sphere cast in two-dimensions.

The solid circle represents the character with the visual sense. The shape with a dashed outline that starts from the character represents the field of view. The sizeable dashed circle represents the area "seen" by the sphere (circle) cast in a single iteration. The dotted shapes represent objects with problems one and two. The solid rectangle is the only shape that should be detected.

5.7.2 Behaviours

All behaviours operate on exclusive entities. I divided behaviours into activatable and requestable. The former uses permanent entities that are activated with a tag component. The latter uses short-lived entities created for each request separately.

5.7.2.1 Activatable behaviours

The activatable behaviours are attached to an AI character. Every update all attached activatable behaviours calculate their selection priority and value. An example from combat behaviour of such calculation is in Figure 5.19. The system collects all behaviours and selects one by adding an ActiveBehaviour component. Behaviours with this component are allowed to modify the state of the character. The implemented combat behaviour tries to activate an attack each frame to demonstrate, that the combat system handles everything on its own and ignores invalid requests.

```
Entities
.ForEach((
  DynamicBuffer<ThreatData> threatDataBuffer,
  ref DynamicBuffer<SelectionResult> resultBuffer,
  in HasCombatBehaviour combatBehaviour
) => {
     if (threatDataBuffer.IsEmpty) {
       return;
     }
     resultBuffer.Add(
       new SelectionResult {
         BehaviourEntity = combatBehaviour.Behaviour,
         Priority = int.MaxValue,
         Value = float.MaxValue
       }
     );
   }
 )
.Schedule();
```

Figure 5.19: Combat behaviour selection.

5.7.2.2 Daily Routines

Daily routines need two parts: the pathfinder and routine itself.

I implemented the pathfinder as a requestable behaviour. For a behaviour to request a path, it needs to create an event entity (PathRequest) which contains information about the start and end points of the path. During the update, the pathfinder system queries all the unprocessed path events and finds a path for them. The system inserts the path into the event entity as a DynamicBuffer (PathCheckpoint) of three-dimensional points. It is much more challenging to check for a buffer's existence than a component's. Therefore the system adds PathFinishedTag component, to mark path entity as finished.

The routine is activatable behaviour, requiring at least two systems—selection system and behaviour system. I ended up implementing it as four systems, a Selection system, a Path Requester system, a Behaviour system and a Delay system. As the name suggests, the Selection system adds the behaviour to the selection buffer with the lowest priority and zero value. Once the system is activated, the Path Requester system requests a path if no path is available. The Behaviour system waits until a valid path is available. Once a path is available, it uses lerp function to move the character between a current checkpoint and the next checkpoint. Once the character reaches next checkpoint, it goes to the next until it reaches the end of the path. After reaching the end of the path, it destroys the path. The code of the Behaviour system

```
var deltaTime = Time.DeltaTime;
Entities
.ForEach(
  (Entity entity, ref ActivePath path, in ActiveBehaviour active) => {
    if (HasComponent<PathFinished>(path.Value)) {
      var pathBuffer = GetBuffer<PathCheckpoint>(path.Value);
      if (pathBuffer.Length > path.LastCheckpoint + 1) {
        var last = pathBuffer[path.LastCheckpoint];
        var next = pathBuffer[path.LastCheckpoint + 1];
        if (path.NewCheckpoint) {
          path.ChangePerSecond =
            MovementSpeed / math.distance(last.Position, next.Position);
          path.Time = 0f;
          path.NewCheckpoint = false;
        }
        path.Time =
          math.min(1f, path.Time + deltaTime * path.ChangePerSecond);
        var newPosition = math.lerp(last.Position, next.Position, path.Time);
        SetComponent(active.Entity, new Translation {Value = newPosition});
        if (path.Time >= 1f) {
          path.LastCheckpoint += 1;
          path.NewCheckpoint = true;
        }
      } else {
        commandBuffer.RemoveComponent<ActivePath>(entity);
        commandBuffer.DestroyEntity(path.Value);
      }
    }
  }
)
.Schedule();
```

Figure 5.20: Daily Routines Behaviour system.

is in Figure 5.20. Finally, the Waiter system waits for a specified amount of time before allowing the cycle to continue.

5.7.2.3 Requestable behaviours

Requestable behaviours are not attached to an AI character, but rather respond to requests. Each request needs to have a request component attached (every behaviour has a different request component). All requestable behaviours are asynchronous and not guaranteed to be completed before the next frame. The requester needs to store a reference to the request entity and check every frame if it is completed. The request is marked as completed when the behaviour removes the original request from the entity and replaces it with a result. An example of requestable behaviour is pathfinding. Because pathfinding has to rely on built-in NavMesh systems, it has to run on the main thread without Burst. Code example of the pathfinding behaviour is in Figure 5.21.

```
struct PathRequest : IComponentData {
// From position
  public float3 From;
// To position
  public float3 To;
// Entity which requested the path.
  public Entity Entity;
}

struct PathCheckpoint : IBufferElementData {
// Position of the checkpoint
  public float3 Position;
}

class PathfindingSystem : NextFrameSerialBufferSystemBase{
void OnUpdate(EntityCommandBuffer commandBuffer) {
  Entities
  .WithoutBurst()
  .WithNone<PathCheckpoint>()
  .ForEach(
    (Entity entity, in PathRequest request) => {
      var navMeshPath = new NavMeshPath();
      var foundPath = NavMesh.CalculatePath(
        request.From, request.To,
        1, // area mask
        navMeshPath
      );
      var pathBuffer =
        commandBuffer.AddBuffer<PathCheckpoint>(entity);
      if (foundPath) {
        pathBuffer
          .EnsureCapacity(navMeshPath.corners.Length);
        for (
          var i = 0; i < navMeshPath.corners.Length; i++
        ) {
          pathBuffer.Add(
            new PathCheckpoint {
              Position = navMeshPath.corners[i]
            }
          );
        }
      }
  })
  .Run();
}}
```

Figure 5.21: Implementation of requestable pathfinding behaviour.

5.8 Input

I chose the new Input System package to handle inputs. It replaces the older input system in Unity which has been troublesome for many years.[23] However, neither the old nor the new input system are yet compatible with Unity DOTS. Luckily, the integration into DOTS was not difficult.

To integrate the new Input System into DOTS, I first had to call Input System Update function manually to synchronise the update with DOTS systems (Figure 5.22). Next, I had to ensure that each system that uses inputs is called after the Input System update. Because the Input System is not compatible with DOTS, systems using it cannot be compiled with Burst, and cannot use jobs. Another issue is that the Input System uses Vector2[1] type instead of float2 from Unity Mathematics. I had to convert all vector inputs to float2. An example of a system using inputs is in Figure 5.23.

```
// Alternative to SystemBase for systems.
// It does not support Entities.ForEach syntax.
public class InputUpdateSystem : ComponentSystem {
  protected override void OnUpdate() {
    InputSystem.Update();
  }
}
```

Figure 5.22: Implementation of Input System update.

[1]Vector2 is Unity's built-in two-dimensional vector type

```
public class MoveInputIntentSystem : SystemBase {
  private PlayerMovementControls _inputActions;
  protected override void OnCreate() {
    base.OnCreate();
    _inputActions = new PlayerMovementControls();
    _inputActions.Movement.Move.Enable();
  }
  protected override void OnDestroy() {
    base.OnDestroy();
    _inputActions = null;
  }
  protected override void OnStartRunning() {
    base.OnStartRunning();
// Enable movement actions
    _inputActions.Enable();
  }
  protected override void OnStopRunning() {
    base.OnStopRunning();
// Disable movement actions
    _inputActions.Disable();
  }
  protected override void OnUpdate() {
    Entities
    .WithAll<UseInputDevices>()
    .WithoutBurst()
    .ForEach(
      (ref DirectionalMoveData directionData) => {
// Read joystick data as two-dimensional vector
        var input =
         _inputActions.Movement.Move.ReadValue<Vector2>();
// Convert two-dimensional input vector to
// three-dimensional velocity vector
        directionData.IntentVelocity =
         new float3(input.x, 0, input.y);
      }
    )
    .Run();
  }
}
```

Figure 5.23: Implementation of reading input data for character movement.

5.9 User interface

I chose UI Toolkit for the user interface implementation. UI Toolkit does not yet support DOTS[1], so it is not possible to compile UI systems with Burst and use them inside jobs. As a result of the missing support for DOTS, it also requires a GameObject to contain UIDocument MonoBehaviour. To allow usage inside DOTS, each UIDocument is contained within an ECS class component. While this introduces some limitations, it is not a problem, because most of the time, I use Entities ForEach function. An implementation example is in Figure 5.24.

User interface can be created either from code (Figure 5.6) or UXML (Unity Extensible Markup Language) (Figure 5.25) and USS (Unity Style Sheets) (Figure 5.26) files. Or a combination of both. There is a visual editor available for the latter. The main UI is composed of several smaller UXML files to make it easier to understand and maintain.

To create the prototype UI, I decided to use the visual editor because it provides the best experience. It supports all the essential functions I needed and immediately showed a preview of how the UI looks.

5.9.1 World-space user interface

One significant limitation I faced was the fact that UI Toolkit does not support world-space UI. In other words, it cannot place UI elements inside the three-dimensional world space. The only way to work around this issue is to use a different solution. Because uGUI (Unity UI) is built-in Unity and is easy to work within less complicated projects, I decided to use it. I needed to use the world-space UI to implement floating damage numbers when a character is hit to make it easier to find issues and give exact information to number-oriented players.

[1] DOTS support is coming in a future release.

```
// Class component so it can contain TextElement (class)
// without being shared component
public class FramePerSecondHud : IComponentData {
  public TextElement Text;
}

public class FramePerSecondHudSystem : SystemBase {
  protected override void OnUpdate() {
    Entities
    .WithoutBurst()
    .ForEach((
      // class component - no in/ref needed
      FramePerSecondHud hud,
      in FramePerSecondData data
    ) => {
      var millisecondsDeviation =
        (data.MaxFrameTime - data.MinFrameTime) * 100f;
      var milliseconds = data.AverageFrameTime * 1000f;
      hud.Text.text =
        $"{1f/data.AverageFrameTime:F1}fps{NewLine}" +
        $"{milliseconds:F2}ms{NewLine}" +
        $"{millisecondsDeviation:F1}ms␣~";
    })
    .Run();
  }
}
```

Figure 5.24: Implementation of frame per second UI system.

```
<ui:UXML xmlns:ui="UnityEngine.UIElements" xmlns:uie="
   UnityEditor.UIElements" xsi="http://www.w3.org/2001/
   XMLSchema-instance" engine="UnityEngine.UIElements"
   editor="UnityEditor.UIElements"
   noNamespaceSchemaLocation="../../../UIElementsSchema/
   UIElements.xsd" editor-extension-mode="False">
  <ui:Label text="FPS" display-tooltip-when-elided="True"
     name="FpsText" style="-unity-font-style:␣bold;␣font-
     size:␣12px;␣-unity-text-align:␣upper-right;␣color:␣
     rgba(255,␣255,␣255,␣255);" />
</ui:UXML>
```

Figure 5.25: Example of a UXML (Unity Extensible Markup Language) file.

```
.inspector-label {
  padding-left: 2px;
  padding-right: 2px;
}

.preview-image {
  min-height: 300px;
  min-width: 300px;
}

.fill-row > Button, ToolbarMenu {
  flex-grow: 1;
}

#unity-content-container {
  flex-grow: 1;
}
```

Figure 5.26: Example of a USS (Unity Style Sheets) file.

5.10 Animations

Animations are an essential part of each game; without them, it would be static. Unfortunately, the animation package for DOTS is in a very early preview and working with it is complicated. I decided to turn to Unity Mecanim Animation System, which is currently the default animation solution for Unity.

As with every GameObject system, this required some extra work to connect with Unity DOTS. I decided to use a class component—AnimatorProxy (Figure 5.27)— because the system can never use the Animator component outside of the main thread, and there is no need to modify it. Every system that needs to access the Animator can use either ForEach or EntityManager.

Humanoid animation system is an example of a system that uses Animator and its implementation in Figure 5.28.

```
public class AnimatorProxy : IComponentData {
  public Animator Animator;
  public float Offset;
}
public class AnimatorProxyAuthoring :
  MonoBehaviour, IConvertGameObjectToEntity {
  public void Convert(
    Entity entity,
    EntityManager dstManager,
    GameObjectConversionSystem conversionSystem
  ) {
    var animator = GetComponentInChildren<Animator>();
    dstManager.AddComponentData(
      entity,
      new AnimatorProxy {
        Animator = animator,
        Offset = animator.transform.localPosition.y
      }
    );
  }
}
```

Figure 5.27: AnimatorProxy component implementation with custom Authoring MonoBehaviour.

```
public class HumanoidAnimationSystem : SystemBase {
  private const double VelocityThreshold = 0.01;
  private static readonly int Forward =
    Animator.StringToHash("Forward");
  protected override void OnUpdate() {
    Entities
    .WithoutBurst()
    .ForEach((
      in AnimatorProxy animatorProxy,
      in BaseMoveData baseMoveData,
      in ExtraMoveData extraMoveData
    ) => {
        var planarVelocity =
          new float2(
            extraMoveData.ActualVelocity.x,
            extraMoveData.ActualVelocity.z
          );
        var distance =
          length(planarVelocity) / baseMoveData.MoveSpeed;
        if (distance < VelocityThreshold) {
          distance = 0f;
        }
        animatorProxy.Animator.SetFloat(Forward,distance);
      }
    )
    .Run();
  }
}
```

Figure 5.28: Implementation of a humanoid animation system.

Chapter 6

Testing

Testing is an essential part of software development; without it, the resulting program has a very high chance of being plagued with issues. Even prototypes should have some level of testing involved. The testing of this prototype proved to be more challenging than expected.

Initially, I planned to implement unit testing for database operations, modifiers, and some general-purpose systems. Unfortunately, this was not possible due to issues with the Entities library. I decided to abandon automated testing and instead only test functionality manually. The manual testing consisted of playing the prototype, trying various conditions that might occur during gameplay and validating whether they work or not. I also used assertions in some areas to catch potential problems.

After finishing the prototype, it was time to test and confirm its viability with several testers. However, the restrictions imposed by the government during the state of emergency made testing options limited. In the end, I decided to test everything remotely. I created a short scenario that the testers went through. During the testing session, the tester records the screen and audio (or are present in a voice call). Seeing what players did and thought in this form, allows me to see some common problems. Lastly, the testers rate the overall experience with the prototype and whether they found it enjoyable.

6.1 Compilation

The compilation for the testing of this project proved to be more challenging than initially expected. The first significant platform-specific problem was that it is not possible to open a database file directly. Instead, it needs to be copied to some location accessible by the application. Only then can it be used by the database. The database supports an option to run from memory. However, I could not make it work, as there were issues that seemed like in-memory corruption of the database. Similar issues were reported by other developers who use LiteDB, but unfortunately, they are difficult to reproduce.

Unity developed a scripting backend IL2CPP that tries to improve Unity projects' performance, security, and platform compatibility. Unfortunately, in this prototype, it led to significant degradation of performance. Instead, I decided to use its alternative Mono.

Unfortunately, during testing, the Burst compiler caused native crashes and had to be disabled. Disabling Burst led to significant degradation of performance and caused some devices, which initially ran the prototype at target framerate[1], to struggle to maintain even half the target framerate. Due to changes in the codebase, it was impossible to revert to an older version of Burst.

6.2 Testing scenario

Before proceeding with the instructions, do not forget to turn on a screen recorder.

By touching a dragging at the bottom right quarter of the screen, you can make the character move. There are no other interactable parts in the user interface.

Your character has awakened inside a barricaded building. Your first task is to reach a cellar and find a key inside it. Once you have the key, use it to unlock the door. After unlocking the door, your next task is to find the next key in the next area, which you can reach by a portal near the well. After finding the key and leaving the area, a portal will transport you near the square; however, there is a barrier preventing entry. To destroy/unlock this barrier, you need two keys (you should already have one from an earlier area). You can find the third and final key by entering the last procedurally generated area. The entrance is near the intersection of the barrier and the road. The last procedurally generated area has two levels, and only the second one contains a key. You can reach the second level by finding a portal inside the first level. Once you have gathered both keys, destroy/unlock the barrier and reach the centre of the square. When you reach the centre of the square, you have completed all the required tasks. Thank you for testing.

6.3 Results

The results from testing were mixed. Testers found the game concept to be interesting and fun. They also praised the modifier system as an interesting mechanic, which makes combat more interesting. On smaller screens, testers reported that the touch area for the joystick was too small. The movement system had only a single small issue with friction. The character slowly walked (slid) on a gradual slope and from testers perspective, walked by itself.

[1] Target framerate for this prototype is 60 FPS (frames per second).

The procedurally generated areas were difficult to orient, and testers sometimes struggled to find their target. The orientation in the dungeon was even more difficult because the camera was too close. The procedurally generated area with two levels also proved difficult for testers, because most of them expected to find a key in the first level and would spend minutes looking for it.

The technical aspect of the game was very problematic. All testers reported performance issues, and in some cases the prototype was unplayable. The technical problems did not end there. The HybridRenderer was the most problematic package of them all. Some devices experienced issues with lighting, and on some, it rendered nothing at all. With poor performance also comes battery drain and some testers reported rapid battery drain on their device.

The testers also reported flipped normals, which caused some triangles in the procedurally generated areas to be incorrectly lit. This issue was known, but after a short investigation, I determined there is no apparent issue in the code, and fix might be very complicated. Because I also determined it would be best to rework the game into 2D, which would not suffer from this problem.

6.3.1 Performance

Many testers noticed that the prototype ran at suboptimal framerates. Often at half or even less of the target framerate. There are several reasons for deficient performance. The primary cause is HybridRenderer, which has an enormous overhead.

A reference frame, taken from a debug build, took 37.19 milliseconds to render. The HybridRenderer overhead amounted to 13.28 ms. Compared to that, all the game's logic took 10.30 ms. It is important to note that physics took 6 ms to process. However, because the framerate is so low, two physics steps were executed in one frame. If the game ran at target framerate, the logic would only take about 3 ms. The actual rendering took 5.64 ms, which is as expected. By lowering the graphical settings, this can be reduced to 4 ms.

In some cases, more powerful SoC had much lower framerates than weaker SoC from a different manufacturer. I did not investigate this issue further, but the cause might be an incompatibility between Hybrid Renderer and specific SoC.

6.3.2 Testing conclusion

Testing showed significant performance issues with HybridRenderer. It is much more sensible for mobile devices to use 2D where the rendering takes a lot less power. A 2D rendering solution is currently in early development as part of Project Tiny, which offers the only “pure” DOTS experience in

Unity. In the future, they plan to expand this solution outside of Project Tiny, but it is not clear when this might happen. Any project with more complicated gameplay should avoid Project Tiny as there is no uncomplicated way to avoid its shortcomings. This leaves a hybrid solution, where rendering is taken care of by GameObjects. While this will not yield any performance benefits, it should not be difficult to convert it into pure DOTS rendering once it is available.

The flipped normals seem like it might be a problem with HybridRenderer rather than the mesh itself. I increased the camera distance to make it easier for players to orient themselves in the procedurally generated areas. I also reduced the size of the procedurally generated areas to reduce the chance of getting lost. Last but not least, I increased the touch are for the floating joystick.

I plan on converting this project into 2D a continuing the work on it in the future. Conversion to 2D should solve most problems testers had with the prototype. I also plan to continue implementing the logic in DOTS. Because the rendering tools are still lacklustre, I think using existing Unity rendering is better. Hopefully, by the time the game is ready, Unity will have a proper DOTS 2D rendering solution.

Conclusion

By the end of 2020, Unity DOTS is an exciting glimpse at the Unity engine's future. Unfortunately, it is far from ready for other than experimental projects. Unity laid the groundwork, but not much is yet built on it. However, not every part of DOTS is unstable. Burst, Job System and Mathematics are three parts of DOTS, which can be used today in existing projects. Together they provide significant improvements to parts of the code by optimizing it and providing safer and more straightforward multithreading.

The core of Unity DOTS is still in early stages of development. The most developed package, the Entities, has significantly improved over time. However, much work still lies ahead before it can be considered stable enough for widespread use. Physics package has all the fundamental tools for most projects. Sadly, it is mostly undocumented, and I would not recommend using Unity Physics for mobile games. Hybrid Renderer, the bridge between DOTS and the existing rendering architecture, is one of the most problematic parts of DOTS. It only supports basic features and has an extensive performance overhead. The Animation and Audio packages require a lot of extra effort from the developer to create even basic things, compared to existing Unity tools. Many features have not yet been released, such as user interface, asset management, NavMesh, and many others. I estimate it will take at least two to three years before DOTS will be a viable solution for a considerable number of Unity projects.

I succeeded in creating the prototype with all the requirements from the assignment. The player's goal is to collect keys to unlock new areas. The prototype has a physics-based movement system, which gives the player the ability to freely explore the environment. The player has no means of fighting back, so they must dodge enemies instead of engaging them. The modifier system makes the combat more engaging by enabling proper interaction between damage values of various attacks. It can also be expanded in the future to any property. The procedurally generated environments provide players with replayable value. The testers expressed concerns that the environments can

be disorienting at times because everything looks the same. Lastly, artificial intelligence gives non-player characters the ability to fight and make the world look less static with routines. The prototype runs on devices with Android 5.0 and newer.

Because Hybrid Renderer is computationally expensive, I think it would be better to convert the game into 2D instead of 3D. 2D would help with many problems such as battery drain and orientation in procedurally generated areas. Even though conversion to 2D would be a significant effort, most of the logic can be preserved as it is not specific to three dimensions. The mechanic of collecting keys is a good start, but the testers deemed it dull in its current state.

The prototype was a successful endeavour that explored the advantages and disadvantages of DOTS in its current state. Developers interested in becoming early adopters of DOTS can learn from it and make a better decision on whether and how to use DOTS in their current or future projects.

Bibliography

[1] Unity. Entity Component System package. [online], visited: 17.7.2020. Available from: https://docs.unity3d.com/Packages/com.unity.entities@0.16/manual/index.html

[2] Microsoft. Blittable and Non-Blittable Types. [online], visited: 17.7.2020. Available from: https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types

[3] Unity. GameObject conversion. [online], visited: 29.11.2020. Available from: https://docs.unity3d.com/Packages/com.unity.entities@0.9/manual/gp_overview.html

[4] Unity. Unity.Mathematics. [online], visited: 29.11.2020. Available from: https://github.com/Unity-Technologies/Unity.Mathematics

[5] Mechtley, A. Overview of physics in DOTS - Unite Copenhagen. [online], visited: 29.11.2020. Available from: https://youtu.be/tI9QfqQ9ATA

[6] Unity. Hybrid Renderer. [online], visited: 5.11.2020. Available from: https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.10/manual/index.html

[7] Unity. Project Tiny. [online], visited: 5.11.2020. Available from: https://docs.google.com/document/d/1A8hen2hLFY5FLkC5gd3JP2Z-IpHfnAX-CpYLK3aOdwA/edit

[8] Buck, J. Maze Generation: Prim's Algorithm. [online], visited: 31.1.2020. Available from: http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm

[9] Shiffman, D. *The nature of code*. United States: D. Shiffman, 2012, ISBN 0985930802.

[10] Conway, J. Game of Life. [online], visited: 7.9.2020. Available from: https://playgameoflife.com

[11] O'Leary, M. Generating fantasy maps. [online], visited: 9.9.2020. Available from: https://mewo2.com/notes/terrain/

[12] Unity. ScriptableObject. [online], visited: 24.9.2020. Available from: https://docs.unity3d.com/Manual/class-ScriptableObject.html

[13] Unity. PlayerPrefs. [online], visited: 24.9.2020. Available from: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

[14] Point, T. Software design basics. [online], visited: 26.10.2020. Available from: https://www.tutorialspoint.com/software_engineering/software_design_basics.htm

[15] Unity. Asset Database documentation. [online], visited: 14.11.2020. Available from: https://docs.unity3d.com/Manual/AssetDatabase.html

[16] Unity. Loading resources at runtime. [online], visited: 14.11.2020. Available from: https://docs.unity3d.com/Manual/LoadingResourcesatRuntime.html

[17] Unity. Addressables. [online], visited: 14.11.2020. Available from: https://docs.unity3d.com/Packages/com.unity.addressables@1.1/manual/AddressableAssetsGettingStarted.html

[18] Joachim_Ante. Addressables and DOTS. [online], visited: 14.11.2020. Available from: https://forum.unity.com/threads/subscenes-and-addressable-asset-system.861838/#post-5677804

[19] Fowler, M. Architecture. [online], visited: 26.10.2020. Available from: https://martinfowler.com/architecture/

[20] Unity. Entities and Physics samples. [online], visited: 15.11.2020. Available from: https://github.com/Unity-Technologies/EntityComponentSystemSamples

[21] Patel, A. Introduction to A*. [online], visited: 20.11.2020. Available from: https://www.redblobgames.com/pathfinding/a-star/introduction.html

[22] Hocking, J. Procedural Generation Of Mazes With Unity. [online], visited: 20.11.2020. Available from: https://www.raywenderlich.com/82-procedural-generation-of-mazes-with-unity

[23] Damm, R. Introducing the new Input System. [online], visited: 22.11.2020. Available from: https://blogs.unity3d.com/2019/10/14/introducing-the-new-input-system/

Appendix A

Glossary

Blittable types data types with identical memory representation in managed and unmanaged code.

Convex mesh a mesh is convex if, given any two points within the mesh, the mesh contains the line between them.

Dereference to access value or object located in a memory location stored within a pointer or another value interpreted as such.

Direct3D graphics API for rendering 3D vector graphics on Windows. It is part of DirectX.

DirectX collection of APIs for handling tasks related to multimedia (primarily game programming and video) on Windows.

IL2CPP scripting backend developed by Unity, which can improve performance, security, and platform compatibility in comparison to Mono.

Instantiation a process of creating a new instance.

LLVM a collection of compiler and toolchain technologies.

Managed code a computer program that can only be executed in VES (Virtual Execution System).

Mono Ecma standard compliant .NET Framework-compatible software framework. It includes a C# compiler and a Common Language Runtime. In 2016, Microsoft acquired the developers of Mono.

MonoBehaviour base class from which every Unity script derives.

OpenGL cross-platform API for rendering 2D and 3D vector graphics.

Pointer a variable whose value is the address of another variable.

Sprite two-dimensional graphical objects that can be placed inside a game world.

Appendix B

Acronyms

ACID atomicity, consistency, isolation, durability.

AI artificial intelligence.

API application programming interface.

ARPG action role-playing game.

BFS breadth-first search.

CPU central processing unit.

DFS depth-first search.

DOTS Data-Oriented Technology Stack.

ECS Entity Component System.

FoV field of view.

FPS frames per second.

GUI graphical user interface.

HDRP High Definition Render Pipeline.

HPC# High Performance C#.

ID identifier.

IDE integrated development environment.

IK inverse kinematics.

LTS long-term support.

PC personal computer.

RMDBS Relational Database Management System.

RPG role-playing game.

RTS real-time strategy.

SQL Structured Query Language.

SRP Scriptable Render Pipeline.

UI user interface.

URP Universal Render Pipeline.

USS Unity Style Sheets.

UXML Unity Extensible Markup Language.

VES Virtual Execution System.

Appendix C

Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
thesis.pdf ............................the thesis text in PDF format
prototype.apk .................. the compiled apk file for 64 bit ARM
src ...................................the directory of source codes
  code .......................................implementation sources
  thesis ............. the directory of LaTeX source codes of the thesis
```