ECSQL

A Domain-Specific Language for Manipulating Entity Component Systems

Aidan Hall

Department of Computer Science

University of Warwick

Supervised by Alex Dixon

Year of Study: 2024

29th April 2024

Abstract

In this project, we produced a framework for building video games that allows the developer to interactively manipulate the state of a game, like a database. We achieved this by creating a simple scripting language, with a query language embedded within it, called ECSQL. We used the Entity Component System design pattern to represent the state of games in a standardised, database-like way. This makes it possible for ECSQL code to apply arbitrary, programmatic transformations to all aspects of any game built using our framework, at run-time.

We created a small game using the framework, which allowed us to demonstrate the capabilities of ECSQL. These included the potential to implement interactive game development tools in just a few lines of code.

The end product is a sufficient proof-of-concept, and could be the basis for be a compelling alternative to game engines like Unity, although little effort was made to optimise its performance.

Acknowledgements

I would like to thank my supervisor, Alex Dixon, for helping to guide the direction of the project.

Keywords

C, Lisp, Video Games, ECS (Entity Component Systems), DSL (Domain-Specific Languages), Databases.

Contents

1 Introduction **6**

2 Background and Research **9**

2.1 Entity Component Systems 9

2.1.1 Definitions of Terms 10

2.1.2 Existing ECS Implementations 12

2.1.3 Feature Breakdown 16

2.1.4 Entity Relationships 17

2.2 Domain-Specific Languages 18

2.3 Lisp 19

2.3.1 Macros 20

2.3.2 Association Lists 21

2.3.3 Lisp Dialects 22

2.4 Potential Applications 23

2.4.1 Graphical User Interfaces 23

2.4.2 Console Commands 25

2.5 ECS Computational Models 25

3 Objectives **26**

3.1 Query Language 26

3.2 System Architecture 27
3.3 Requirements Analysis 28

4 Methodology **33**
4.1 Research 33
4.2 Design 34
4.3 Development 34
4.4 Testing 36
4.5 Tools 36

5 Design **38**
5.1 Entities 38
5.2 Components 40
5.2.1 Component Representation 40
5.2.2 Component Storage 42
5.3 Systems 44
5.3.1 System Scheduling 45
5.3.2 Entity Names 45
5.4 Lisp 46
5.4.1 Type System 46
5.4.2 Structs 48
5.4.3 Scopes and Closures 50
5.4.4 Macros 50
5.4.5 REPL 50
5.4.6 Error Handling 51
5.4.7 Syntax and Short-Hand Forms 51
5.4.8 Core Language and Special Forms 51
5.4.9 Macro System 53

5.4.10 Primitive Functions 54
5.4.11 ECS Lisp APIs 55
5.5 Domain-Specific Languages 56
5.5.1 ECSQL Query Language 57
5.5.2 Entity Initialisation 60
5.5.3 Primitive Argument Type Specifications 61
5.6 Asynchronous REPL 62

6 Implementation **63**
6.1 Entities 64
6.1.1 Entity Names 65
6.2 Components 66
6.2.1 Adding and Removing Components 67
6.2.2 Component-Column Mapping 67
6.2.3 Bootstrapping the `Storage` Component 68
6.2.4 Lisp Components 69
6.3 Queries & Systems 70
6.3.1 Query Compilation 70
6.3.2 Query Execution 71
6.3.3 Systems 73
6.4 Lisp 74
6.4.1 Object Representation 75
6.4.2 Parser and Printer 80
6.4.3 Memory and Addressing 80
6.4.4 Error Handling 82
6.4.5 Scopes 82
6.4.6 Evaluation 83
6.4.7 Macro Expansion 85

6.4.8 Documentation Strings 85

7 Project Management **87**

7.1 Project Progress 87

7.2 Risk Management 90

8 Results & Evaluation **92**

8.1 Example Application 92

8.1.1 C Systems 93

8.1.2 Scene 93

8.1.3 Queries & Lisp Systems 95

8.2 Possible Use-Cases 98

8.3 Requirements Evaluation 100

9 Conclusions **105**

9.1 Further Work 106

9.1.1 Lisp Implementation 106

9.1.2 Entity Relations 107

9.2 Self-Assessment 108

Bibliography **109**

A Lisp Primitives **115**

Chapter 1

Introduction

In the field of game development, it is common to use large, pre-existing frameworks called engines to build games. Engines like Unity [41] & Unreal [9] implement low-level functionality such as rendering pipelines and scene hierarchy management. They also provide development environment tools, including level editors and parameter viewers, that make the game development experience quicker and easier (see Figure 1.1). Most Engines adopt an Object Oriented Programming (OOP) model, at least for user code.

An alternative approach that has gained some popularity in game development recently is the Data-Oriented Design process [6]. Instead of building class hierarchies in an attempt to create a model of the world, Data-Oriented programs are written with a focus on what the actual data is, and how it is transformed [1], with the aim of removing unnecessary complexity.

One prominent Data-Oriented design pattern is the Entity Component System (ECS) [22]. It serves as a concrete alternative to OOP as a framework for building games, while remaining a sufficiently simple concept that there are numerous ECS library implementations, in a wide variety of languages [40, 29, 19]. There are even more ambitious projects such as Bevy [2],

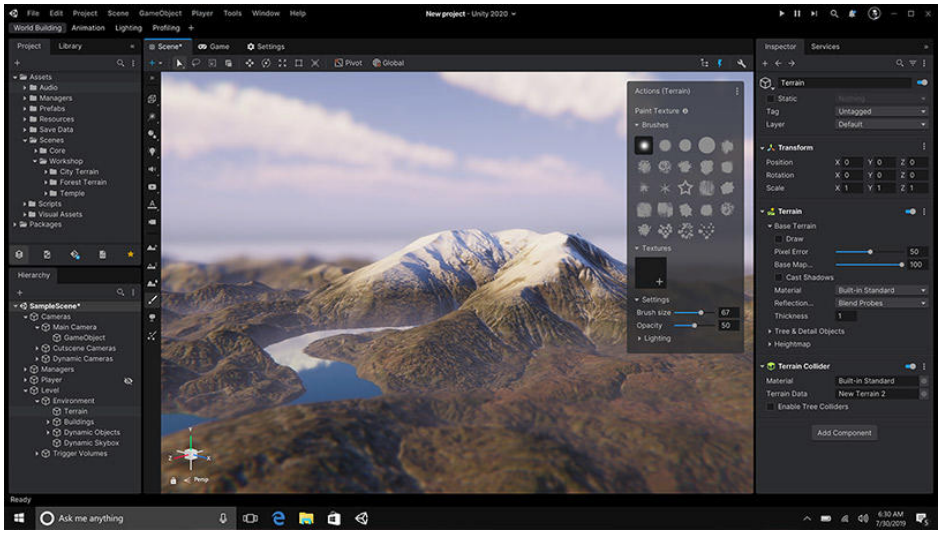

Figure 1.1: The Unity editor displaying the properties of some terrain, a scene hierarchy, and a preview of the scene.

an ECS-based game engine.

The main weakness of these libraries is that they lack most of the helpful interactive features that engines provide. Even Bevy lacks a scene editor at the time of writing. For most small game development teams, it would be infeasible to build all the development tooling of an engine, so they must either go without if they want to use an ECS library, or turn back to using game engines.

Our goal with this project was to produce a single general-purpose tool that could introduce a more fluid, interactive game development experience to developers using ECS libraries, and replicate some of the functionality of game engine editors, without the cost and complexity of implementing their many specialised tools individually.

To achieve this, we implemented a Domain-Specific Language (DSL), on top of a more general-purpose scripting language, that would provide a concise way to express queries about and transformations of the state of an

ECS-based game, in a manner similar to SQL. We refer to this DSL as ECSQL (the Entity Component System Query Language).

We believe our system would be most useful to small teams, or solo developers, who want to make games using an ECS library, but don't have the resources to build their own engine or tools. The expressive power of ECSQL would allow them to do this, because it would reduce the time required to implement each feature.

Furthermore, we believe any game developer could benefit from a system with the capabilities of the ECSQL language, because it provides such a novel way of interacting with and developing a game. It could even act as an effective complement to the traditional tools in large engines like Unity.

We discuss potential use-cases of the ECSQL system in more detail in section 8.2.

Chapter 2

Background and Research

In order to fully appreciate the motivation for our project, on even the fundamental level of its design, some context is required. This includes a basic background explanation of Entity Component Systems and Lisp, which we will provide in this chapter. We also discuss the findings of our initial research, including some potential applications of a system like ECSQL, and different computational models we considered before settling on creating a query language.

2.1 Entity Component Systems

The ECS design pattern is a way of structuring programs, most commonly used in video games, where objects are represented as abstract Entities. There is a lot of terminology required to discuss Entity Component Systems, so we first define these terms, then evaluate some existing solutions.

2.1.1 Definitions of Terms

Several of the important concepts in ECS design have names with general meanings, that are used to refer to unrelated concepts in other contexts. Throughout this project, we will capitalise the terms when referring to their specific meanings within the context of ECS.

Most of these definitions are based on those in the Unity Entities package documentation [39, Entity Component System concepts]. Other sources are cited where relevant.

Entity Something discrete in a game World, with its own set of data. They are represented by a unique ID. Each Entity has a set of Components.

Component A single logical piece of information about an Entity, such as health or position, represented as plain data.

Archetype A unique identifier for all the Entities in a World that have the same unique combination of Component types. It is common to store Component data for Entities with the same Archetype together [39, 29].

System Functions that perform some operation for each Entity matched by a Query. It is possible to represent Systems as Entities [29, Systems].

Query A way of specifying conditions an Entity must meet for a System to operate on it, and what Components of each Entity the System will use. A basic approach would be to specify which Components an Entity *must* have, and which ones it *must not* have. More sophisticated implementations can aggregate Component data from multiple related Entities into one entry in a Query's results [29, Queries].

Relationship A special type of Component used to express how Entities should interact [29, Relationships]. For example, A `ChildOf` Relationship could be used to represent a scene hierarchy.

Tag A Component containing no data. It conveys information about an Entity only by whether or not it is present [29, Queries].

World A collection of Entities, within which each Entity's ID is unique. They are analogous to scenes in general game engine terminology. A System also exists in a specific World, and operates on the Entities within it.

Dependencies A mechanism to control System scheduling. Each System specifies which Components it reads and writes, and possibly some other constraints, to prevent Systems from interfering with one-another while running.

Phases/Pipelines An alternative approach to controlling System scheduling [29, Systems]. A Pipeline is divided into an explicitly-ordered set of Phases. Each System has a Phase Component, that specifies which Phase it should run in. In the strictest form, there is the assumption that no Component type is read from and written to in the same Phase by different Systems. There can be multiple Pipelines, and they typically run at regular intervals, such as once per frame.

Generation A number included as part of an Entity's ID. It is incremented every time an ID is reused (after the last Entity with that ID was destroyed) [30]. This ensures persistent references to Entities are always valid.

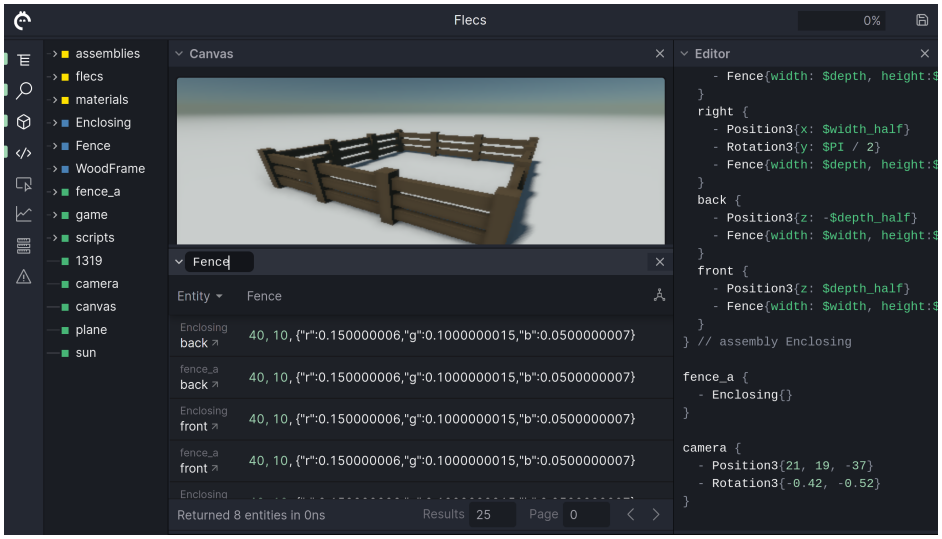

Figure 2.1: Flecs Explorer

2.1.2 Existing ECS Implementations

Since the primary goal of this project is to produce a new way of manipulating Entity Component Systems, we must review existing solutions to identify any gaps in the existing provision. We took the code examples in this section from the official documentation or tutorials for each solution.

Flecs

Flecs [29] was created by Sander Mertens, and is written in C. It appears to be the most conceptually advanced among the ECS implementations discussed here. Mertens has written numerous blog posts on advanced ECS topics, notably including Entity Relationships (see subsection 2.1.4), a distinctive feature of Flecs [24, 27, 30]. Another notable feature of Flecs is that it treats Component types as Entities, which makes it possible to create new ones at run-time.

Flecs has a DSL for creating Queries, Entities, and Component types. This integrates with the Entity Relationships system, allowing it to express

highly complex Queries, that can even select Components of multiple Entities at once, concisely. A Query in the DSL can be mapped almost directly to calls to the C API. Listing 2.2 shows a Query that matches Entities that have Position, but neither Velocity nor Speed.

The DSL is mainly restricted to selecting sets of Entities. It does not support System definitions, or Queries that apply a transformation.

Flecs has a web UI that allows interactive inspection of a Flecs World at runtime, including a tabular view of the results of Queries (see Figure 2.1).

Listing 2.1: How Systems are defined using the C++ API for Flecs.

```
flecs::system sys = world
  .system<Position, const Velocity>("Move")
  .each([](Position& p, const Velocity &v) {
    // each() runs the function on each Entity.
    p.x += v.x;
    p.y += v.y;
  });
```

Listing 2.2: A Query in the Flecs DSL.

```
Position, !{ Velocity || Speed }
```

Unity DOTS

Unity's Data-Oriented Technology Stack is "a combination of technologies and packages that delivers a data-oriented design approach to building games in Unity" [40]. The ECS [39] is part of it, alongside a JIT C# compiler and a parallel job scheduling system. The job scheduling system uses Dependencies to control the order of execution.

The ECS integrates with Unity's editor UI, with a similar interface to that of Unity's normal `GameObject`s (see Figure 2.2). In the code, there are `IComponentData` and `ISystem` interfaces, that Component and System types must implement, respectively. There is a Query builder class, and there are

UI elements for designing Queries.

The documentation includes high-quality explanations of basic ECS concepts, as well as more advanced ones, notably including Archetypes [39].

Figure 2.2: A comparison of the property editor user interfaces for GameObjects and ECS data in Unity (source: `https://unity.com/ecs`).

cl-fast-ecs

This is an ECS written purely in Common Lisp [19]. We have included it as an example that is, at least superficially, quite similar to ECSQL. Since all Entities, Components and Systems are defined in Lisp, they can all be redefined at run-time.

Listing 2.4 demonstrates its `ecs:make-object` function, that implements a tiny DSL of sorts for creating new Entities with certain Component values. As we discuss in section 2.3, Lisp is especially well-suited to creating *embedded* DSLs (see section 2.2), that integrate with code in the host language. For example, the code in Listing 2.3 uses a Lisp expression within the DSL code to assign the object a random position, which only works because the DSL

is embedded.

Listing 2.3: A cl-fast-ecs system.

```
(ecs:define-system move
  (:components-ro (speed)
   :components-rw (position)
   :arguments ((:dt single-float)))
  (incf position-x (* dt speed-x))
  (incf position-y (* dt speed-y)))
```

Listing 2.4: An application of `cl-fast-ecs`'s `ecs:make-object` function, which provides a concise way of creating an Entity with a given set of Components.

```
(dotimes (_ 1000)
  (ecs:make-object
   `((:position
      :x ,(float (random +window-width+))
      :y ,(float (random +window-height+)))
     (:speed :x ,(- (random 100.0) 50.0)
             :y ,(- (random 100.0) 50.0)))))
```

Bevy ECS

Bevy [2] is a game engine written in Rust, with an ECS at its core. Components are implemented as Rust `structs` that implement the `Component` trait. Systems are implemented as Rust functions with a `Query` template as an argument. It supports Systems that iterate over the relevant Entities in parallel.

Listing 2.5 is an example of a Bevy System. It takes a single argument, generated by the `Query` template. The `With` clause requires Entities to have the `Person` Component, without loading its data, which is useful for referencing Tag Components, which have no data. Since Queries are implemented as Rust templates/macros, there is no way to write new ones at run-time.

Listing 2.5: A Bevy System.

```
fn greet_people(query: Query<&Name, With<Person>>) {
    for name in &query {
        println!("hello {}!", name.0);
    }
}
```

2.1.3 Feature Breakdown

Now we have gotten an overview of some existing solutions, we can evaluate their feature-sets to determine where ECSQL fits in. We have chosen to compare them based on the following features, since they effectively illustrate the gap that ECSQL fills:

Query DSL A way of expressing an ECS Query outside the implementation language. Allows Queries to be written and executed at run-time.

Scripting Allows gameplay code to be written written in a high-level easy-to-use language.

Native API Provides a mechanism to interact with the ECS, in particular to define Systems, in a native/systems language such as C or Rust.

Run-time Definitions Allows all ECS features, including Entities, Components and Systems, to be defined and modified at run-time.

Data Manipulation Language (DML) Provides a run-time interface to define and run one-off commands that can apply arbitrary, bulk transformations to many Entities at once.

Table 2.1 provides a breakdown of each of the ECS implementations we have discussed, and ECSQL, in terms of whether or not they implement

Solution	Query DSL	Scripting	Native API	Run-time Definitions	DML
Unity ECS	×	✓	×	×	×
Bevy	×	×	✓	×	×
`cl-fast-ecs`	×	✓	×	✓	×
Flecs	✓	×	✓	×	×
ECSQL	✓	✓	✓	✓	✓

Table 2.1: Feature breakdown of various ECS solutions, including ECSQL.

each of the above features. With ECSQL, we aim to combine the run-time interactivity of `cl-fast-ecs` with the expressive power of Flecs' Query DSL.

2.1.4 Entity Relationships

Entity Relationships are special Component types that can express a relationship between two Entities. The basic idea is to make it possible to add a pair of Entities/Components as a single Component. For example, the Relationship `(A, B)`, when added to an Entity, would represent that Entity having Relationship `A` with Entity `B`. Mertens [27] explains how this simple concept can be used to represent a variety of constructs, including:

Scene Hierarchies Add Relationship `(ChildOf, Parent)` to each child Entity.

Inventory Contents Add Relationship `(Holds, Apple)`, with value n, to an Entity to represent it having n apples in its inventory. Notably, `(Holds, Orange)` would be a distinct Component type.

State Machines Some Relationships in Flecs (such as `ChildOf`) are exclusive, so each Entity can have that Relationship with at most one other Entity. If an Entity can have one of a finite set of states, such as $\{standing, running, jumping\}$, these could be represented with an exclusive `State` Relation: `(State, Standing)`, `(State, Running)`,

```
(State, Jumping).
```

One especially powerful feature of Flecs, enabled by Relationships, is the ability to use Joins in Queries. For example, the following Flecs Query would get the position of each Entity, and its parent, in a scene hierarchy:

```
Position($this), (ChildOf, $Parent), Position($Parent)
```

This is just one of the features that makes the implementation of Relationships in Flecs so complex [24].

2.2 Domain-Specific Languages

It is common for design patterns to be implemented as built-in features of new programming languages. Notable examples include procedure calls and OOP. Norvig [33] describes 3 levels at which patterns may be implemented in a language:

Informal Written by hand each time, implemented as prose.

Formal Implemented in the language, commonly with macros.

Invisible A fundamental part of the language, used implicitly.

Formal pattern implementations provide a more concise, expressive way of using a pattern than informal ones. This is because they save the programmer from thinking about the (often highly repetitive) code required to apply a pattern in a given context. They achieve this by generating that code automatically, at compile time.

A Domain-Specific Language provides a formal way to express a set of operations, constraints, or ideas, concerning a specific application. We can

think of a DSL as a set of complex design patterns, with a compiler or interpreter as the implementation. When a Formal implementation of a DSL compiles DSL code to code in the host language, it is said to be an *embedded* DSL. Embedded DSL code can directly interact with code written in the host language [14, p. 254]. For example, it could access variable definitions that are in scope in the surrounding host language code.

Flecs' Query DSL is not embedded, because C is not powerful enough for that to be possible. The ECSQL Query language (see subsection 5.5.1) is compiled, but the output is not Lisp code (see subsection 6.3.1), and has a separate interpreter implementation, so it is not an Embedded DSL by our definition. The `ecs-add*` (see subsection 5.5.2) language can use the result of evaluating Lisp expressions at run-time (with the `expr` form), and thus has to be embedded.

Embedded DSLs are also not to be confused with embedded scripting languages, such as Lua [17]. These languages typically have an API that allows function calls between themselves and the host language, but generally cannot make use of host language constructs. The interface to run scripting language code from the host language often relies on snippets of scripting language code stored in strings, which must be parsed and evaluated at run-time; this is the case for Lua, and the Flecs Query language.

2.3 Lisp

For reasons discussed in section 3.2, we chose to implement a basic Lisp interpreter as the basis for ECSQL, so we have included some necessary background information on it.

Listing 2.6: A Lisp function that computes $n!$.

```
(defun factorial (n)
  (if (<= n 1)
      1
    (* n (factorial (- n 1)))))
```

Lisp is a dynamically-typed language, that has the familiar set of built-in data types, with the addition of lists and symbols (e.g. `*`, `defun`). Lists are composed of nodes called `cons` cells. A `cons` cell is a pair of two values, the `car` and the `cdr`. In a list, the `car`s contain the values, and each `cdr` stores a reference to the next `cons` cell in the list, or a null terminator value. Each element of a list can be any type, including `cons` cells, so lists can represent arbitrarily nested expressions. We refer to this as *list structure*.

We represent lists visually using S-expressions: elements are separated with spaces, with parentheses as delimiters. A dot in an S-expression indicates that the next value will be the `cdr` of the last `cons` cell, so we can represent a single pair as `(car . cdr)`; the list `(a b c)` is equivalent to `(a . (b . (c . nil)))`.

2.3.1 Macros

Lisp code is represented using list structure and symbols (see Listing 2.6), so it is easy to write Lisp code that generates Lisp code, as shown in Listing 2.7. Such code is commonly written in the form of Lisp *macros*.

To greatly over-simplify, Lisp macros are functions that run at compile time and can generate Lisp code [13, p. 162]. Macros allow programmers to extend the language using the language. A recurring introductory example is `let` (see Listing 2.8), which adds local variable bindings to a Lisp that only supports lambda function application. For a deep dive into the potential of Lisp macros, see Graham [14] and Hoyte [16].

Lisp features a *quotation* operator, `'`. An expression following this op-

erator is not evaluated, so code can treat it as a value. The *quasiquote* or *backquote* operator, `, acts like ', with the additional feature that the *unquote* operator, ,, cancels it out. See Listing 2.7 for an example of how we can use these when writing macros.

Listing 2.7: A piece of Lisp code that generates a piece of Lisp code.

```
(cons '+ (list 1 2 (cons '* (cons 3 (cons 4 nil)))))
=> (+ 1 2 (* 3 4))
```

Listing 2.8: Local variable binding, implemented as a Lisp macro, derived from our definition in `util.lisp`.

```
* (defmacro let (binds . body)
    `((lambda ,(mapcar #'car binds)
        . ,body)
      . ,(mapcar #'cadr binds)))
* (macroexpand-1
   '(let ((a 2)) (* a a)))
((lambda (a) (* a a)) 2)
```

2.3.2 Association Lists

The flexibility of list structure means Lisp programmers can use it to represent a variety of data structures. One example is the association list [13, p. 51], which models a key-value map. Each element of the list is a pair, with the key in the `car` and the value in the `cdr`. The standard Lisp function `assoc` finds the first element of an association list with a given key (see Listing 2.9).

Listing 2.9: Association List Example

```
* (defvar names '((x . 3) (y . 2)))
* (assoc 'x names)
(x . 3)
* (assoc 'z names)
()
```

2.3.3 Lisp Dialects

There are a number of similar languages that are collectively referred to as *dialects* of Lisp. Notable examples include Common Lisp [13], Scheme [11] and Emacs Lisp [10]. They all share the basic traits of code being represented with S-expressions, and the general semantics of how expressions are evaluated, but there are a number of minor differences that make each distinct.

The most fundamental difference between Common Lisp and Scheme, the two most prominent dialects, is the way they handle namespaces [35, Ch. 2]. Common Lisp has entirely separate namespaces for functions and variables, even allowing a function and a variable with the same name to exist in the same scope. It is referred to as a Lisp-2 because of this (see Listing 2.10). By contrast, Scheme is referred to as a Lisp-1, because it has a single namespace for functions and variables (see Listing 2.11); there is little distinction between them, to the point that they are defined with the same operators (`let` for locals, `define` for globals).

Listing 2.10: Common Lisp's separate namespaces.

```
* (defun b (x)
    (* x x))
b
* (let ((b 5))
    (b b))
25
```

Listing 2.11: Scheme's single namespace.

```
* (let ((a 5)
        (b (lambda (x) (* x x))))
    (b a))
25
```

2.4 Potential Applications

We considered a few potential applications that could benefit from an interactive ECS manipulation language, before deciding to focus on the Queries and programmatic transformations of ECSQL.

2.4.1 Graphical User Interfaces

One idea we considered was to use Entity Component Systems to represent a Graphical User Interface (GUI) within a game.

This idea was explored in great depth in the Polyphony project [36]. Their approach was to represent widgets as Entities, with properties represented as Components For example, a button could have Components for its position, size and text, and to indicate what happens when it is pressed. The actual widget drawing and interaction is then implemented as Systems that operate on the set of widget Entities.

Expressing all information about the state of the GUI with Components is similar to the traditional "retained"[1] model of GUI implementation [5], but comes with a few notable advantages. These advantages are broadly the same as those of using ECS for the rest of a game: using Components instead of inheritance makes it simpler to reuse and compose features dynamically, and centralising computation into Systems can improve performance, and makes it easier to extend the functionality of the whole GUI at once (e.g. by creating one new System) [36, § 4].

It is also possible to use an "immediate mode" [5] approach, using draw calls in Systems to re-build the whole GUI every frame. In an immediate-mode system, the GUI itself does not have any persistent, internal state.

[1] Typically Object-Oriented

Instead, we can derive its appearance from the state of Entities within the game world.

For example, we have previously implemented a "health bar" System, that Queries for Entities with the `Position` and `Health` Components, then draws a health bar above each of them. This GUI feature has no internal state of its own, and is trivially enabled or disabled by starting or stopping the System. We have included a possible implementation of such a System in Listing 2.12. For a similar example in action, see the position labels System in Listing 8.7.

Listing 2.12: ECSQL code for a Health Bar System.

```
(ecs-new-system
 (Graphics) (and Pos Health) (pos health)
 ;; Full health bar
 (draw-rectangle (- pos #*v2(10.0 10.0))
                 #*v2((health-total health) 5)
                 black)
 ;; Remaining health bar
 (draw-rectangle (- pos #*v2(10.0 10.0))
                 #*v2((health-current health) 5)
                 red))
```

In a retained GUI system, it might have been necessary to create a health bar widget for every matching Entity, and copy the position from each Entity to its corresponding widget each frame.

With a more retained, extensional representation, and a language like ECSQL, it would be possible to build GUIs interactively, with a similar experience to Tcl/Tk [38]: the developer could evaluate an expression to create a new Entity representing a GUI widget, and immediately see it in the game window.

With a retained, ECS-based GUI, it could also be useful to apply the flyweight pattern [34, Ch. 3], using a single template Entity for each kind

`/give`	Add an item to a player's inventory.
`/gamemode`	Change a player's game mode, which determines what actions they can perform.
`/locate`	Find the nearest in-game structure of a specific type.

Table 2.2: Examples of Minecraft Commands.

of GUI element, with Components holding shared information, such as the colour and size. A GUI rendering System could read the Component values for this one Entity, and re-use them for all GUI Entities based on it[2]

2.4.2 Console Commands

Some games feature a developer console, accessible to the player. Examples include Minecraft [12] and games built with Valve's Source engine [42]. These provide a textual interface to issue simple, high-level commands that change the state or behaviour of the game in some way. We have listed some examples in Table 2.2.

A Query language like ECSQL would make it possible to implement some of these commands in a single line of code. As a result, the developer could create many such commands near-effortlessly. Another approach would be to give the players direct access to the Query language, but perhaps restrict the set of Entities and Components they can manipulate, so they can't completely break the game.

2.5 ECS Computational Models

[2]This is a natural use-case for Entity Relationships (see subsection 2.1.4).

Chapter 3

Objectives

In this chapter we set out and justify the objectives for the project, and specify a set of requirements a solution would have to satisfy to meet those objectives.

3.1 Query Language

The initial motivation for this project was a perceived gap in the provision of existing ECS solutions. From prior experience, we knew most had only an inexpressive API in a static, compiled language, with little or no support for directly interacting with the ECS's state at run-time (see subsection 2.1.2). We realised that a domain-specific language could mitigate this problem, by providing an expressive, interactive interface.

We chose to create a Query language because it maps well semantically to the structure of an Entity system. Mertens [31] and Martin [23] have both drawn a link between ECS and relational database systems: Entities act as primary keys, and a Component type is equivalent to a column in a database table. Martin [23] claims that "life is more fun if you embrace the

dynamicity of the query", suggesting that thinking of an ECS-based game like a database affords greater flexibility than would be possible with static architectures like OOP inheritance hierarchies. Consequently, we believed that taking this analogy as far as possible and creating a dynamic, run-time query language could allow for myriad interesting, novel applications.

Furthermore, we believed that such a language could be powerful and expressive enough to replicate many of the specialised utilities seen in more advanced game engines, and go beyond their capabilities in certain areas. For example, a parameter viewer could be implemented simply as a Query to get the values, and a function call to put them on the screen (see section 8.1).

There would also be a lot of potential value in applying bulk, programmatic transformations interactively, as seen almost exclusively in database systems (with languages like SQL). It would be challenging, if not impossible, to replicate this kind of functionality with specialised utilities, to the same level of expressivity and generality.

3.2 System Architecture

We can now identify the two primary objectives of this project: creating an Entity Component System library, and a Query language frontend to it that could be used at run time.

Since our primary aim was to create a Query DSL optimised for this specific application, we deemed it necessary to create the language implementation ourselves, rather than use an existing embedded scripting language such as Lua. This allowed us to integrate ECS functionality into the language at a fundamental level, in the type system, for instance (see subsection 5.4.1). This also gave us complete control over the execution model of the language,

allowing us to implement each feature in whatever way we deemed best.

We chose to implement the Query DSL within a basic Lisp implementation, with macros, in order to maximise the effectiveness of the solution, relative to the amount of effort we would need to put in. For one thing, we needed a scripting language to express transformations to apply to Entities matched by the Query language, and for another, it is fairly straightforward to create a basic Lisp implementation [8]. Furthermore, Lisp's macro system is especially suited to creating advanced embedded languages with a small amount of code (see section 2.3): there is even precedent for implementing Query languages within Lisp using macros [14, Ch. 19]. We also had substantial experience writing Lisp, including macros, prior to this project.

We chose to create the ECS ourselves, so we could ensure it would integrate elegantly with the language. For instance, we used Lisp S-expressions for the internal Query representation, allowing us to easily generate and manipulate it within Lisp. This made it possible to extend the Query language without having to modify the implementation in C.

3.3 Requirements Analysis

These are the original requirements we created for the specification. Since we wrote them, the focus of the project shifted away from performance considerations, and almost entirely towards the capabilities of the language itself. As a result, some of these requirements are no longer relevant. One notable change was that we decided to implement an interpreter rather than a compiler for ECSQL.

We have prioritised these requirements using the MoSCoW approach, specifying whether we **MUST**, **SHOULD**, **COULD** or **WILL NOT** at-

tempt to meet each one.

1. Construct an ECS:
   (a) Represent Entities with unique, automatically-generated IDs (**MUST**).
   (b) Represent Components as plain `struct` types (**MUST**).
   (c) Schedule System execution at run-time (**MUST**): This will allow the language to manipulate Systems.
   (d) Store Components according to Archetypes [39, Archetype concepts] (**COULD**): This could mean storing Component data for all Entities with the same Archetype together. This is a performance optimisation.
   (e) Represent Entity Relationships and Joins, in a similar manner to Flecs [29] (**SHOULD**). This would greatly increase the expressive power of the language.
   (f) Allow fast, multi-threaded access to the Entity store (**SHOULD**): This would be necessary to enable some of the parallelisation enhancements discussed below.
2. Implement a language that can manipulate the state and behaviour of an ECS-based game, with the following capabilities and features:
   (a) Create and delete Entities (**MUST**).
   (b) Add and remove Components of Entities (**MUST**).
   (c) Display and edit the contents of Components (**MUST**).
   (d) Select sets of Entities to apply a transformation to, based on Queries [29, Queries] and the values of their Component data (**MUST**).

(e) Start and stop Systems, and list which ones are running (**MUST**).

(f) Compile programs to LLVM IR (**SHOULD**): This would potentially yield much greater performance than bytecode, increasing the range of scenarios where it would be suitable to use the language.

(g) Compile programs to bytecode (**COULD**): A bytecode VM would probably be slower than native code generated by LLVM, but may be easier to work with depending on how the implementation works out.

(h) Define and run Systems and functions (**SHOULD**): Functions are a fundamental abstraction mechanism in any language, and Systems are of similar importance in ECSQL.

(i) Execute commands received asynchronously from a REPL (**SHOULD**): This would allow the user to type commands and see results with the game still running.

(j) Have a terse and ergonomic syntax (**SHOULD**): Users should be able to write code in the language quickly and easily, since the intention is for them to use it as an interactive development tool.

(k) Run Systems and general functions written in the host language (**SHOULD**): This would allow the user to arbitrarily extend the language to suit their purposes.

(l) Produce high-quality error messages (**SHOULD**): This language is primarily intended to be a convenient tool for developers, so it should be pleasant to use.

(m) Monitor the result of repeatedly running a specified query (**SHOULD**): This could let the programmer, for example, track the velocity of

a particular Entity over time.

(n) Enforce a strong type system (**SHOULD**): This is likely to be necessary, since it would allow the compiler to perform more safety checks, so they could be omitted during execution, making it more efficient.

(o) Compose systems together to build more complex ones (**COULD**): The general idea would be to have a set of fundamental transformation systems, which the programmer can compose to quickly produce interesting behaviour. The exact mechanism would depend on the details of how the language is designed.

(p) Parallelise programs automatically (**COULD**): There are three main possible approaches:

Data Parallelism Run a single System on multiple Entities simultaneously. ECS operations are naturally data-parallel on the level of Entities, so this could be reasonably straightforward.

System Scheduling Run Systems that will not interfere with one-another, most commonly by accessing the same Components, at the same time.

Optimised Codegen If a System performs highly intensive computations for every Entity, applying optimisations such as vectorisation in the ECSQL compiler could improve performance.

(q) Optimise code generation in the ECSQL compiler (**COULD**): The performance of the resulting LLVM code could possibly be improved by using vector intrinsic operations.

3. Write a set of tests (**SHOULD**): These will allow us to ensure the

code is working correctly. We could also include performance tests or benchmarks. These would allow us to assess the runtime efficiency of programs written with ECSQL compared to equivalent ones written without it, and the effectiveness of extensions intended to improve performance.

4. Create a demo game (**SHOULD**): The live demonstration in the presentation would be a lot more engaging with the example of a small game developed using the system.

5. Allow ECSQL code to be compiled at the compilation stage of the host language, and embed the result into a final executable (**COULD**): This could reduce the cost of translation at run-time, and possibly mean the language compiler itself could be omitted from the executable, reducing binary size.

6. Implement a profiler for ECSQL (**COULD**): This could break down how long functions called in ECSQL take to run. This would be useful for performance testing, since a profiler for the host language probably wouldn't be effective at profiling the performance of ECSQL code.

7. Support game state serialisation and deserialisation, including the state of Systems (**COULD**): This would be an easy way to save and reload game data (between sessions), and send it across the network.

8. Accept ECSQL commands from remote network hosts and execute them (**COULD**). This has many potential applications, although it could have serious security implications, for example, allowing remote code execution.

Chapter 4

Methodology

In this chapter we outline some of the methodologies we employed to complete this project.

4.1 Research

This was not a research-focused project, so we took a straightforward approach to research. We read up on some existing ECS implementations, and did a comparative analysis to help decide which features of each to incorporate into our ECS. The results of this can be seen in chapter 2. This research also helped inform the design of the ECSQL Query language, which is largely based on Flecs' Query DSL [29].

We determined that "industrial" Lisp implementations such as SBCL and GNU Guile would be too complex to inform our design. Instead, we focused on educational resources about Lisp implementation as sources of inspiration for the design, primarily Engelen [8] and Queinnec [35].

4.2 Design

For each step of the design process, we considered and analysed several approaches, sourced either from existing solutions or our own creativity. We then compared each possible solution, weighed up their pros and cons, and selected or synthesised a suitable approach to take for the implementation based on that comparison. See chapter 5 for how we applied this process in practice.

4.3 Development

By focusing on the **MUST** and **SHOULD** requirements in section 3.3, we were able to determine the scope of a Minimum Viable Product (MVP) for this project. It would comprise an ECS library, a Lisp implementation and a Query language that would allow a game developer to manipulate games created using the ECS library to manipulate the state of the Entities in the game with Lisp code.

We chose to implement the MVP using a Waterfall methodology [3, Waterfall]. This was the best choice, since we had a clear, fixed set of requirements, so we could create a schedule (see Gantt chart) and set of deliverables (see deliverable-oriented WBS) before carrying out the implementation. Additionally, since the product would not meet its most basic requirements until the entire MVP was implemented, an iterative/sprint-based methodology would not have been greatly beneficial at this stage.

In addition to the requirements of the MVP, we identified optional features that would improve the usability of the product, but were not essential to its success (identified as **COULD** requirements). We estimated each extension would take at most 1–2 weeks to implement, so we decided to adopt

an agile methodology while implementing them. This allowed us to take an iterative approach to development, incorporating regular feedback after implementing each extension (over a 1-week sprint) through weekly supervisor meetings.

This workflow ensured we avoided wasting time and effort, since we only researched and designed each extension during the sprint when we intended to implement it. As a result, we only had to do research and design for the extensions that actually made it into the final product.

The Agile workflow we adopted was as follows:

1. Do background research, and preliminary design work to determine the general approach required to implement the extension.

2. Hierarchically decompose the feature into a set of implementation tasks, filed as sections in the `README.org` file (see section 4.5).

3. For each task, write up detailed notes concerning how it is to be implemented, and any issues that need to be addressed in the implementation, underneath the corresponding heading.

4. Execute all implementation tasks for the feature.

5. Demonstrate the feature implementation to the project supervisor, to confirm it is sufficient.

We were also able to apply this workflow, to a lesser extent, during the MVP development phase, presenting the incomplete implementation to our supervisor each week as we progressively built more of the required functionality.

4.4 Testing

Lisp enables a somewhat unusual style of development that Graham refers to as "Interactive Programming" [14, § 3.4]. Instead of producing a static test suite, then compiling the code and running every test at once, the programmer can write one function at a time, and immediately call it (in the REPL) with a few sample inputs. Once an error is identified, the faulty code can be immediately updated and re-run in the same Lisp session, allowing very fast turnarounds. This somewhat eliminates the need for writing unit tests.

We were obviously able (and willing) to adopt this approach for testing the components of this project written in Lisp, some of which are among the most complex, so we did. Furthermore, since we created bindings to Lisp for most of the operations on the ECS anyway, we were able to adopt an only slightly diminished form of the same testing methodology for the ECS code as well, even though most of it was written in C. The same goes for all the Lisp functions that we implemented in C (including `concat` and `assoc`). The only part we couldn't adopt this approach for was the core Lisp implementation itself, but that was one of the easiest parts of the project, since we have extensive prior experience implementing interpreters and compilers. As a result, this testing methodology was sufficient for most of the project.

4.5 Tools

We used a few different software tools while working on this project.

Emacs This is our preferred development environment. It is also unquestionably the best editor for Lisp code, which we had to write a lot of in this project. Notably, it includes the `inferior-lisp` package, which

provides an excellent interface for interacting with Lisp REPLs. This made the experience of interacting with the REPL much smoother, which was especially valuable because it made the demo in our presentation more compelling. For example, its history function can cycle back through complete, multi-line Lisp expressions, which allowed us to re-run the same Query several times, with minor alterations, without having to re-type the whole thing.

GitHub Code hosting. Early on, we attempted to use the GitHub Projects Kanban board for project management, but we found it cumbersome to work with, so we switched entirely to Org mode.

Org Mode A plain-text note-taking and task management application [7]. Org Mode has a text markup syntax similar to Markdown, with built-in support for marking headings with a workflow state. We used TODO, DOING, DONE and CANCELLED states. This simple approach allowed us to rapidly generate and consume new backlog tasks, which was especially valuable while developing extensions. In addition, we were able to keep our research and design notes for each task immediately under its corresponding heading, which was a lot more efficient than having information about the same task stored in different places, in different formats. It can also track when the workflow states of tasks change, which we used to generate our burn-down chart (see Figure 7.2).

Chapter 5

Design

In this chapter we discuss the design of the complete ECSQL system. This includes the ECS, the Lisp interpreter, and the Query DSL. The overall architecture of the whole system is shown in Figure 5.1.

For the design of the ECS, we drew heavy inspiration from Flecs [29], and blog posts by Sander Mertens [25, 26, 30], the creator of Flecs.

Our Lisp implementation uses a similar architecture to the one described by Engelen [8], though ours is substantially more featureful.

5.1 Entities

As in most ECS libraries, we have chosen to represent Entities with unique, numeric IDs. IDs are generated sequentially, wrapping around when an upper limit is reached. We say an Entity is "live" if it has been assigned an ID, and has not yet been "destroyed". To ensure Entities' IDs remain distinct, we store the set of live Entities' IDs, and skip forward to the first unused one when creating an Entity.

One issue that can arise, when an Entity's ID is used to store a reference

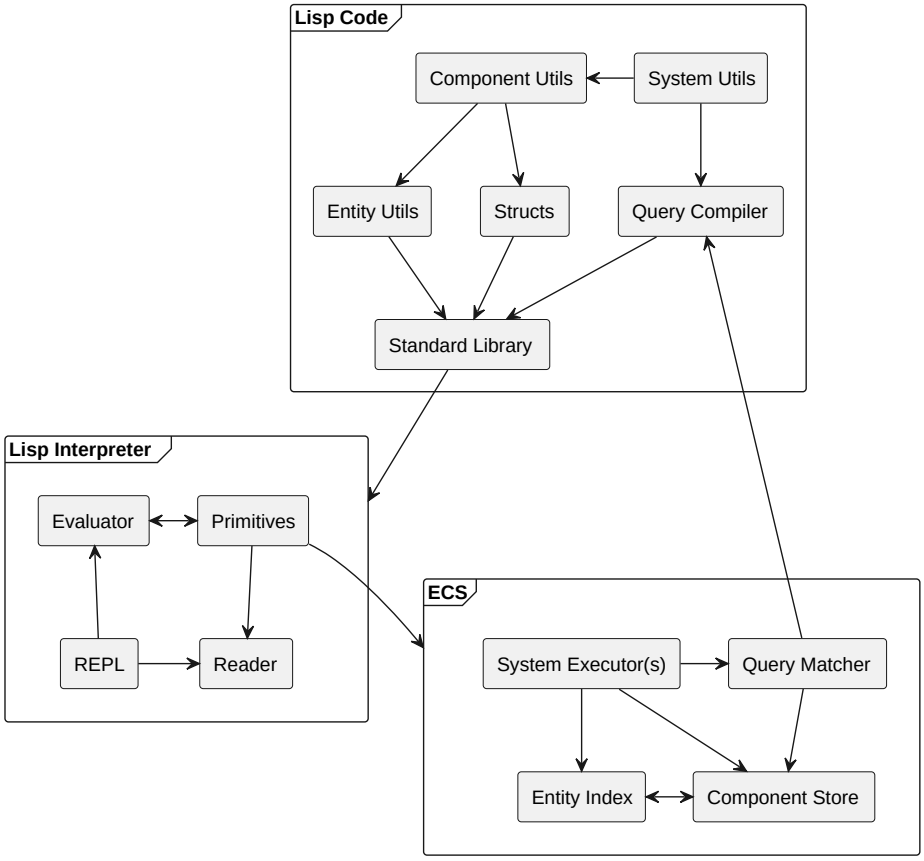

Figure 5.1: Architecture of ECSQL.

to it, is that if a referenced Entity is destroyed, this can leave a dangling reference. This isn't too hard to solve, since we can just check if the Entity is live before using the reference, but that doesn't work if another Entity is created with the same ID before the check. As mentioned in subsection 2.1.1, this is typically solved by adding a Generation count alongside the ID [30]. The Generation is incremented each time an ID is reused, so a particular ID + Generation pair will only refer to a single Entity throughout the whole lifetime of the program.

See subsection 6.4.1 for the exact memory layout of Entity IDs.

5.2 Components

The main design considerations for Components are how they are identified, how we keep track of what Components each Entity has, and how Component data is stored and accessed.

5.2.1 Component Representation

We have adopted an idea from Flecs of representing Components as Entities [30]. Mertens explains a few benefits of this approach, with three of his examples being directly relevant to this project:

Reflection Type information about a Component can be stored as a Component of that Component. For example, a `Position` Component could have a `Type` Component that indicates positions are stored as 2D vectors. We will need run-time type information for Components so Lisp code can manipulate them, so this is essential.

Support for scripting languages Since Components are Entities, and En-

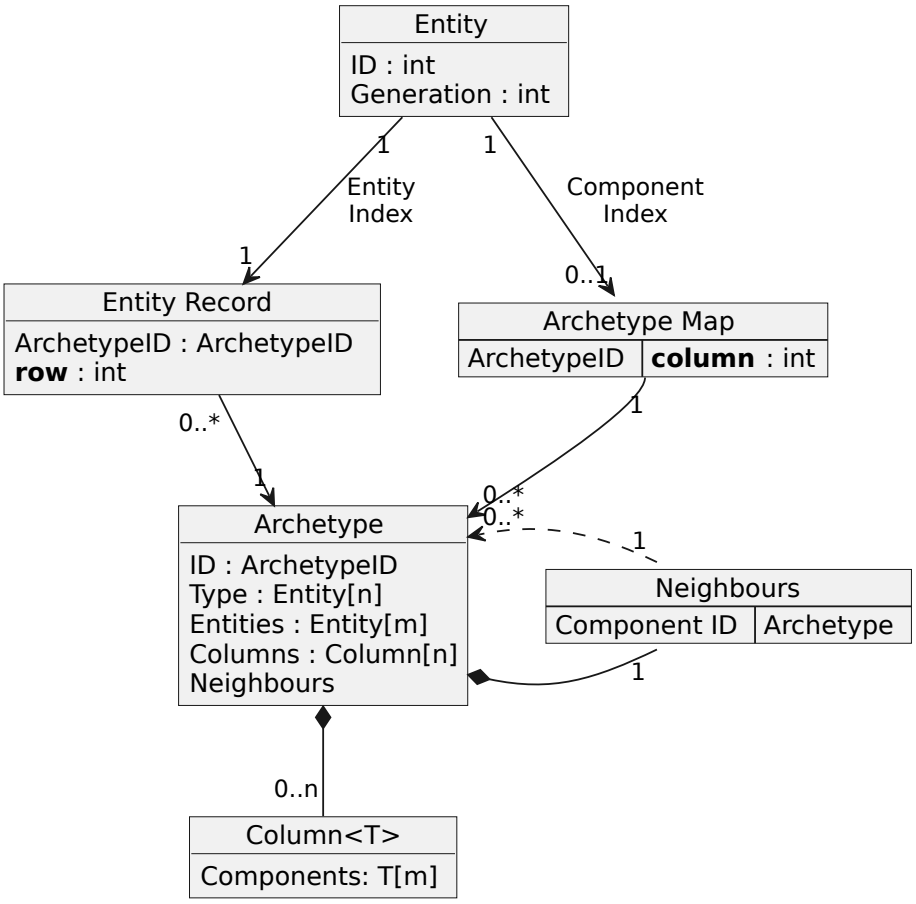

Figure 5.2: Architecture of the Component Store.

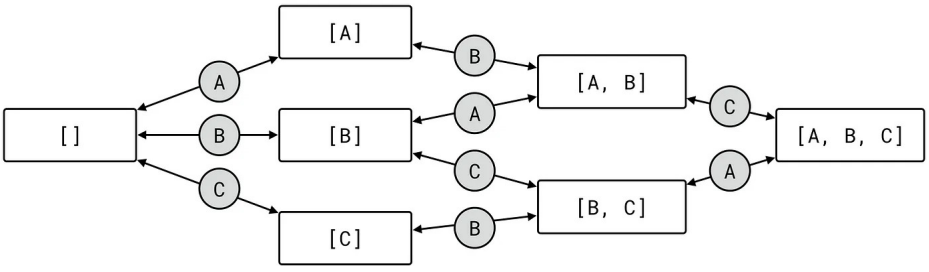

Figure 5.3: An example Archetype graph, created by Mertens [26].

tities can be created as run-time, so too can Components. This would allow us to define Components within Lisp.

Tooling Tools such as editors can use ECS Queries to get information about Components the same way as they could for "normal" Entities. A large part of the motivation for this project was to create tooling for ECS games, so this only expands the potential capabilities of the product.

Components that contain data, such as Position, have a Component called `Storage` added to them, that contains the size of that data. This also makes it easy to represent Tag Components as Entities that don't have the Storage Component.

5.2.2 Component Storage

The way Components are stored is a major contributor to the performance of an ECS implementation, so doing so efficiently has been the subject of much discussion [21, 26]. The main recurring point in these discussions is that the best way to maximise throughput is to store data (e.g. Components) contiguously in memory, and iterate linearly through it. This has highly predictable memory access and control flow behaviour, so the instruction and data caches are used efficiently, with minimal evictions.

Since our aim was not to break new ground in this field, we based our design (see Figure 5.2) on one created by Mertens [26], which is supposed to offer good performance, while remaining reasonably straightforward.

The design is Archetype-based, meaning all Component data for all Entities with the same Archetype (set of Components) is stored together. Each Archetype has a "type" vector, which contains the Entity IDs of the Components in that Archetype.

The values of a particular Component for all Entities in a given Archetype are stored in an array, called a *column*. Each of the Component values for a particular Entity are stored at the same position in each Column, that being its *row*.

We use two main data structures to track which Entities and Components are in which Archetypes:

Entity Index This stores the Archetype of each Entity, and the row in the Archetype where that Entity is stored.

Component Index For each Component, this stores the Archetypes it is in, and what Column within each Archetype its data is in, if any.

The process to get the value of a Component for a particular Entity is as follows:

1. Find the Entity's Archetype and row in the Entity Index.
2. Identify the column in that Archetype that stores the Component using the Component Index.
3. Use the row and column indices to obtain the address of the Component value within the Archetype.

To test whether an Entity has a Component, we look up what Archetype it is in using the Entity Index, and search for that Component in that Archetype's Type vector (an array of Component IDs).

This design also features what Mertens refers to as an Archetype graph (see Figure 5.3). The Neighbours member in each Archetype maps a Component (Entity) ID to the Archetype you would get by adding or removing that Component on the original Archetype. This makes it more efficient to

add/remove single Components of Entities, since we can just follow the corresponding reference in the Neighbours map. The mappings are added lazily, as Entities move in/out of the Archetype.

In Mertens' version, there are edge nodes (indicated by the circles in the diagram) that contain references to the two Archetypes at either end of the edge (labelled "add" and "remove"). This adds an unnecessary extra indirection when traversing an edge, so we directly store references to the neighbour Archetypes in the Neighbours map.

One part of Mertens' design we didn't implement was a mechanism to look up Archetypes directly using the "type" vector (of Component IDs). In Mertens' design, this is used when creating an Entity, to find its initial Archetype. We chose not to include this because the hash map library we used (klib) does not support hashing arrays. Instead, we create every Entity with no Components (and put it in the pre-defined "empty" Archetype), then add the Components one at a time, traversing the Archetype graph. This is undoubtedly less efficient, but creating Entities and adding/removing Components are relatively uncommon operations anyway, so we did not consider them important to optimise.

5.3 Systems

Systems are represented as Entities, with many of the same benefits as representing Components as Entities. We attach the following Components to Entities representing Systems:

`Query` A compiled ECSQL Query (see subsection 5.5.1), that selects which Entities the Systems runs on, and binds Component data as appropriate.

`System` A function that implements the System's behaviour.

`SystemData` Arbitrary data passed to the System function each time it is invoked. This allows us to create multiple instances of the same System that behave differently, depending on the value of this data.

5.3.1 System Scheduling

We can use a Query to find and run a set of Systems: `(select System Query SystemData)`.

One limitation of this approach is that we don't necessarily know what order Entities will be in when iterating over the results of a Query. The order Systems are executed in can affect the behaviour of the game in unexpected ways, so we must enforce some ordering on them. For example, the displayed position of a moving Entity will be different depending on whether its position is updated before or after it is drawn on-screen. On the other hand, the order in which other pairs of Systems are executed is irrelevant, so we only need a partial ordering for the results to be predictable.

We used a phased approach to enforce this partial ordering [29, Systems]. Each System is given one of a set of *Phase* Components. We can then use a Query to run the set of Systems with each Phase in a manually defined order. While we won't necessarily know what order Systems will run in within a Phase, we do know all Systems in one Phase run before all Systems in the next.

5.3.2 Entity Names

The API function `ecs-set-name`, inspired by Mertens [29], assigns a unique identifier to an Entity. This makes it easier for the user to refer to a specific Entity repeatedly, since they don't have to remember its ID, which may not

even be the same between sessions. This is especially useful for Components.

Entity names in ECSQL are Lisp symbols. This is convenient, because once a symbol is read in, we have a unique representation of it that is easy to compare (`==`), and makes Queries integrate better into the rest of the Lisp code in a project, since we don't have to use strings for Entity/Component names.

5.4 Lisp

The Lisp dialect we were most familiar with prior to starting this project was Common Lisp (see subsection 2.3.3), so we went with a similar design. We used Engelen [8] as a basis for the general architecture of the interpreter.

5.4.1 Type System

Our Lisp language has the following built-in types. Each of these names is a Lisp symbol, which is used to refer to that type in the implementation.

`nil` A null value. Terminates a list. The semantics of this differ between Lisp dialects. In our case, there is exactly one nil value, displayed as `()`, and we define a global variable, `nil` that has that value.

We have not included a Boolean type, adopting the Lisp convention of `()` (the nil value) representing "false", and all other values being considered true [8, p. 11]. This convention also entails using the symbol `t` to represent "true", when no other information is necessary.

`string` A string.

`character` A character, physically distinct from integers.

`i32` A 32-bit signed integer.

`f32` A 32-bit floating-point number.

`file` A file handle. We mainly included this so we could implement `load` in Lisp.

`vector` A fixed-length array of Lisp objects.

`undefined` An undefined value, used to indicate error conditions. This is mostly unnecessary, since we have proper error handling support (see subsection 5.4.6), but could theoretically be used for non-fatal errors.

`symbol` An object with a textual name. Each symbol is equal to itself, and not equal to anything else.

`pair` A pair of two values, referred to as the `car` and `cdr`. Linked list nodes are pairs, where the `car` stores the value at that node, and the `cdr` stores a reference to the next node.

`primitive` A built-in Lisp function, written in the host language (C). Functions should usually only be implemented as primitives for performance, or to integrate with a native code library (such as our ECS).

`closure` A function written in Lisp. See subsection 5.4.3.

`entity` An ECS Entity (ID). Technically, we could have represented Entities using other types like integers, pairs or structs, but this approach has some concrete advantages:

- We can restrict what operations are allowed on Entity objects, and how they are created, to make it harder for "invalid" Entity

objects (that do not refer to a live Entity) to be created. For example, we can disallow adding numbers to an Entity ID to produce a different one.

- The Lisp printer can display them differently from other data, so the meaning of their value is clear.
- The entire Entity ID can fit within our object representation (see subsection 6.4.1).

5.4.2 Structs

In addition to the built-in types, our language supports user-defined data structures, or structs, with a similar set of operations to structs in C. Struct members can be any previously-defined type, including struct types. See Listing 5.1 for an example of how these can be used.

Defining a struct type will define the following functions:

Constructor (`make-{struct}`) Creates a new struct with initial values for all struct members.

(Nested) Getters and Setters (`{set-,}struct-{member}`) Functions to access not only the immediate members of the struct, but also the members of any child structs (e.g. `player-pos-x`). The nested accessors make accessing child struct members much more concise than it would be otherwise, since we only need one function call, rather than multiple nested calls.

Copier (`copy-{struct}`) Copies the values of all members of a struct to another of the same type.

Complete Setter (`set-{struct}`) Assigns new values for all struct members at once. Technically, we could achieve this behaviour by creating a new struct and copying its values across, but this is more convenient and saves an unnecessary allocation.

Type Predicate (`{struct}p`) Tests whether a value is of that struct type.

Printer (`print-{struct}`) Allows the value of a struct to be printed at the REPL.

Listing 5.1: Lisp struct definition and use.

```
* (defstruct v2
    (x f32)
    (y f32))

* (defstruct player
    (rotation f32)
    (name string)
    (pos v2)) ; Recursive structs

* (defvar a
    (make-player 90.0 "Aidan" (make-v2 -20.0 42.1)))
#*player(90.0 "Aidan" #*v2(-20.0 42.1))

* (player-name a)
"Aidan"

* (set-player-pos-x a 99.9)
* a
#*player(90.0 "Aidan" #*v2(-20.0 99.9))
```

We included structs in our design in order to represent ECS Component types. Components that need to be used in C code must have a static C type, so the constraints on the struct members' types must be preserved when working with Components in Lisp. As a result, we enforce strict typing for struct members, so each is always of a known type.

5.4.3 Scopes and Closures

For simplicity, we chose to only support global function scope. Lambda expressions in Lisp create lexical closures; to implement these correctly we had to support local variable scope, in addition to global variable scope. Macros, like functions, can only have global scope.

Closures allow Lisp functions to capture the scope in which they are defined, and retain access to it whenever they are called. For example, in Listing 5.2, the `adder` function returns a function that captures the value of `n`, so calling it later will add 5 to the argument. They are integral to the way Lisp code is written, so our language must implement support for them.

Listing 5.2: Closure Example

```
* (defun adder (n)
    (lambda (m)
      (+ n m)))
adder
* (defvar add5 (adder 5))
add5
* (funcall add5 6)
11
```

5.4.4 Macros

Our macro system is fairly typical. Refer to any of the resources on Common Lisp we have cited for more information.

5.4.5 REPL

A read-eval-print loop:

1. Print a prompt to indicate the process is waiting for input.

2. Read a Lisp expression from the user.

Shorthand	Expansion	Description
`'x`	`(quote x)`	Quotes the next form, so it is not evaluated.
`` `x ``	`(quasiquote x)`	Backquotes the next form [13, p. 399].
`,x`	`(unquote x)`	Unquotes the next form (counterpart to `` `x ``).
`#'x`	`(function x)`	Access the definition of `x` in the function namespace.
`#/x`	`(macro x)`	Access the definition of `x` in the macro namespace.
`#\x`	Character x	The next character after `#\` is read in as a character object.
`#*x init`	`(make-x . init)`	Allows the printed representation of structs and a few other complex types to be correctly read in.

Table 5.1: Lisp Shorthands

3. Evaluate it.

4. Print the result.

5. Repeat.

5.4.6 Error Handling

As mentioned in subsection 5.4.1, we have the undefined type to represent an invalid value. We decided this was insufficient for cases where errors occur in deeply nested expressions, so our language also has the `wrong` primitive function. It stops evaluation, prints an error message, and resumes execution at the REPL.

5.4.7 Syntax and Short-Hand Forms

Our Lisp has a fairly standard syntax, with the addition of a few convenient short-hands forms. These function similarly to Common Lisp's read macros [13, p. 399]. A few of the notable ones are listed in Table 5.1.

5.4.8 Core Language and Special Forms

This is the main function that evaluates a Lisp form. See algorithm 1.

Algorithm 1: Basic Lisp Evaluator

```
Input: expr
if expr is a cons cell/pair then
    if car of expr is a special form keyword then
        Handle it specially;
        return special result.
    else
        Call function named in car of expr with cdr of expr as
         arguments;
        return result of application.
    end
else
    return expr.
end
```

Most list forms have a function in the first position, in which case the function is applied. There is a set of special forms, however, that cannot be implemented as functions, and would be at best impractical to implement as macros. Instead, they must be hard-coded into the evaluator. They are as follows:

`(quote form)` Return form without evaluating it.

`(function name)` Look up name in the function namespace.

`(macro name)` Look up name in the macro namespace.

`(progn body...)` Evaluate the body forms in sequence and return the result of the last one.

`(lambda params body...)` Create a closure (a callable Lisp function object) that captures the scope in which it is defined, with the supplied parameters and body.

`(and args...)` Evaluate each argument form in sequence. Stop and return

`nil` if one returns `nil`. If none return `nil`, return the result of the last argument form.

`(or args...)` Evaluate each argument form in sequence. Stop and return the result of the first one to not return `nil`. If all return `nil`, return `nil`.

`(if cond then else...)` If cond evaluates to a true value, evaluate the then form, otherwise evaluate the else forms.

`(while cond body...)` Repeatedly execute the body forms, as long as the cond form returns a true value.

`((lambda params body...) args...)` Execute body in the current context, with the addition of the names in the params list bound to the results of evaluating the corresponding args.

`(setq var val)` Assign var the value obtained by evaluating val.

5.4.9 Macro System

Our Lisp interpreter has a dedicated *macro expansion* step, before normal evaluation begins. We chose to include this because a large part of the benefit of a dedicated macro system over calling `eval` at run-time is that macros are only expanded once, at compile time Graham [13, p. 162]. Considering our intentions to make heavy use of macros to implement the DSLs in this project, it would be unwise to repeat the computation required to expand a macro every time we evaluate a piece of code containing one.

See algorithm 2 for a simplified overview of a macro expansion procedure. Of critical importance is the fact that the arguments are *not* evaluated before we pass them to the macro. This is, in large part, what distinguishes macros

from normal functions, allowing them to do things like compile DSL code into Lisp.

Algorithm 2: Macro Expansion Procedure

```
Input: form
if form is a list then
    head ← macroexpand(car(form));
    args ← cdr(form);
    if head is a macro name then
        macro ← The macro with name head;
        return macro(args).
    else if head is quote. then
        return args.
    else
        Apply macro expansion to each element of args;
        return cons(head, args).
    end
return form.
```

5.4.10 Primitive Functions

In order to do anything useful with this language, we must include a set of basic "primitive" Lisp functions, implemented in C. These include mathematical, string, list and ECS (see Table 5.2) operators, as well as fundamental Lisp operators like `funcall` and `eq`.

Most of these functions have a restricted set of valid argument types, so to save the effort of implementing type checking in every primitive function, we have defined a small DSL to express these restrictions, using Lisp data (see subsection 5.5.3).

Function	Operation
`ecs-new`	Create an Entity.
`ecs-add`	Add a Component to an Entity.
`ecs-has`	Check if an Entity has a Component.
`ecs-set`	Set the value of a Component for an Entity.
`ecs-get`	Get the value of a Component for an Entity.
`ecs-remove`	Remove a Component from an Entity.
`ecs-destroy`	Destroy an Entity.
`ecs-new-component`	Create a new Component.
`ecs-set-name`	Create an identifier for an Entity.
`ecs-entity`	Find the Entity with a certain ID.
`ecs-lookup`	Find the Entity with a certain name.

Table 5.2: Primitive ECS Lisp API

5.4.11 ECS Lisp APIs

The API in Table 5.2 provides an interface for Lisp code to perform ECS operations. Each maps almost directly to a function that we had to implement in C. This API could be used as shown in Listing 5.3.

Listing 5.3: Example use of the primitive ECS API.

```
* (defvar a (ecs-new))
#*entity(25 0)
* (ecs-new-component 'v2)
#*entity(26 0)
* (ecs-set-name (ecs-entity 26) 'C)
t
* (ecs-add a (ecs-lookup 'C))
Adding new archetype link.
()
* (ecs-set a (ecs-lookup 'C) (make-v2 2.0 -3.5))
#*v2(2.000000 -3.500000)
* (set-v2-x (ecs-get a (ecs-resolve 'C)) 5.0)
5.000000
* (ecs-get a (ecs-lookup 'C))
#*v2(5.000000 -3.500000)
```

This API is suitable for C code, which tends to be verbose and low-level anyway, but we felt we could create a better API for user code. The

Function/Macro	Operation
`ecs-resolve`	Get the Entity with a given ID *or* name.
`ecs-add*`	Add and set multiple Components (see subsection 5.5.2).
`defcomponent`	Define a named, documented Component, and make that name a Lisp variable.

Table 5.3: High-Level ECS Lisp API

additional functions/macros are listed in Table 5.3. They make manipulating the ECS in Lisp much more elegant, as shown in Listing 5.4.

Listing 5.4: Example use of the high-level ECS API.

```
* (defcomponent B f32)
#*entity(24 0)
* (defcomponent C v2)
#*entity(25 0)
* (describe 'C)
C
symbol
A Component. Stores data of type v2.
()
* (defvar a
    (ecs-add* (ecs-new)
              (B 10.5)
              (C 2.0 -3.5)))
#*entity(26 0)
* (list B C (ecs-get a B) (ecs-get a C))
(#*entity(24 0) #*entity(25 0) 10.500000 #*v2(2.000000 -3.500000))
```

5.5 Domain-Specific Languages

Although the main DSL in this project is the ECSQL Query language, due to the flexibility of Lisp's list structure, we were able to implement a couple more DSLs for other parts of the project.

5.5.1 ECSQL Query Language

ECSQL Queries serve two purposes: they act as a predicate that an Entity must satisfy for it to match, and they describe a set of Components that must be "bound", so their values can be used in a System. The grammar of ECSQL is illustrated in Listing 5.5. The predicate and binding meanings of a Query can be extracted out into separate expressions, of the forms shown in Listing 5.6 and Listing 5.7.

A single Component name or ID on its own means that an Entity must have that Component to match, and that the value of that Component is bound. The meanings of the `and`, `or` and `not` expressions in a predicate are self-evident. Concerning binding, `and` inherits its child Queries bindings in sequence, `or` inherits the bindings of its first child Query that matches, for each Entity/Archetype, and `not` binds nothing.

An `opt` expression inherits its child Query's bindings but has no predicate, so the bindings will only apply if a given matched Entity actually has the relevant Components.

Finally, a `has` expression inherits its child Query's predicate, but has no bindings. This is useful for Queries that match Tag Components, but have no data. It is inspired by Bevy's `With` predicate (see section 2.1.2).

As an alternative to that long-winded prose explanation, Table 5.4 presents the dual meanings of each Query form in a more structured way.

Listing 5.5: BNF Grammar for the ECSQL Query language.

```
<query> ::= (and <query> <query>*)
          |  (or <query> <query>*)
          |  (not <query>)
          |  (has <query>) ; Matches without loading data.
          |  (opt <query>) ; Optional
          |  <component>
```

Query Form	Predicate	Bindings
`<component>`	"Has this Component."	This Component.
`(and <query>*)`	"All child Queries match."	Bindings of each child Query, in sequence.
`(or <query>*)`	"Any child Query matches."	Bindings of first matching child Query.
`(not <query>)`	"The child Query does not match."	None.
`(has <query>)`	"The child Query matches."	None.
`(opt <query>)`	None.	Same as child Query, for each Component that is present, or null for Components a matched Entity doesn't have.

Table 5.4: ECSQL Query binding and predicate condition rules

```
<component> ::= <symbol>  ; An Entity/Component name.
              |  <integer> ; An Entity/Component ID.
```

Listing 5.6: BNF Grammar for ECSQL predicates.

```
<pred> ::= (and <pred> <pred>*)
         |  (not <pred>)
         |  (or <pred> <pred>*)
         |  <integer> ; An Entity/Component ID.
```

Listing 5.7: BNF Grammar for ECSQL binding lists.

```
<binds> ::= (<binding> . <binds>)
          |  ()

<binding> ::= <id> ; An Entity/Component ID.
            |  (or <id> <id>*)
            |  (opt <id>)
```

The primary, interactive interface for writing ECSQL Queries is, fittingly, the `ecsql` macro. It is called as shown below. The `bindings` argument is a list of names of variables that are bound to the values of bound Components in the Query, and the `body` is evaluated for each Entity that the Query matches. We also provide the `select` macro that just compiles a Query, for use with C Systems, and the `ecs-new-system` macro for creating Systems in Lisp. The latter has an interface that is mostly the same as `ecsql`, with an

additional parameter to add Components to the System Entity (most often a Phase; see subsection 5.3.1).

```
(ecsql <query> <bindings> <body>...)
(select <query>...)
(ecs-new-system <components> <query> <bindings> <body>...)
```

The following examples may help with understanding the `ecsql` interface. Consider this ECS world:

Entity	Component A (`symbol`)	Component B (`f32`)
27		3.0
28	`cool`	1.0

Both Entities have Component B, but only Entity 28 has Component B. In this context, we can use `and` and `opt` to get the following result. Since Entity 27 doesn't have a value, `opt` binds `a` to `()` when the Query runs on it.

```
* (ecsql (and (opt A) B) (a b) (print (list entity a b)))
(#*entity(28 0) cool 1.000000)
(#*entity(27 0) () 3.000000)
```

Next, since only Entity 28 has Component A, the Query `(or A B)` will bind Component A for Entity 28, but Component B for Entity 27. The bound value of `x` has a different type in each match, but this works naturally because of Lisp's dynamic type system.

```
* (ecsql (or A B) (x) (print (list entity x)))
(#*entity(28 0) cool)
(#*entity(27 0) 3.000000)
```

Finally, we can write Queries that bind no values at all!

```
* (ecsql (and (has B) (not A)) () (print entity))
#*entity(27 0)
```

These examples mainly illustrate the power of the Query language itself. A large part of the power of the ECSQL system comes from the fact that

code in the body of an `ecsql` expression can manipulate the values of Components, and what Components an Entity has. See section 8.1 for some more compelling examples, operating on a more interesting ECS world.

5.5.2 Entity Initialisation

It is common to add multiple Components to an Entity at the same time, especially when creating it. To make it easy to perform this repetitive operation, we have defined the `ecs-add*` macro. It is similar to the `ecs:make-object` function in `cl-fast-ecs` (see section 2.1.2). While not huge, it does implement a simple DSL, a BNF grammar for which is shown in Listing 5.8.

Listing 5.8: BNF Grammar for `ecs-add*`.

```
<start> ::= (ecs-add* <entity> <entry>*)
<entry> ::= <component>
          |  (<component> initialisers...)
          |  (<component> = <value>)
          |  (<component> : <entity>)
          |  (expr <expression>)

<component> ::= <id> | <name>
```

Each entry form adds the corresponding Component. In addition, the list form ones have the following behaviours:

`(<component> ...)` Initialises the Component using the remaining arguments. If the Component is stored as a struct, the corresponding constructor is called.

`(<component> = <value>)` Assigns the Component the result of evaluating the expression after the = sign. This is useful for initialising a Component with a pre-defined struct.

Primitive	Arguments	Type Specification
`quit`	None.	`()`
`funcall`	At least one argument of any type.	`(t . t)`
`+`	Any number of floats or integers.	`(* (or f32 i32))`
`aset`	Vector, index, value.	`(vector i32 t)`
`length`	A list, vector or string.	`((or pair vector string nil))`

Table 5.5: Example Primitive Argument Type Specifications

`(<component> : <entity>)` Copies the Component value from the Entity after the colon. This makes it possible to create "template" Entities, with standard Component values that can be copied when instantiating new "real" Entities.

`(expr <expression>)` Add the result of evaluating the Lisp expression `<expression>` at run-time. This makes it possible to add the value of a variable as a Component. There is no way to set the value in this case, since we can't know the Component type at compile time.

5.5.3 Primitive Argument Type Specifications

This DSL provides an expressive way to declare the types of arguments a primitive function may accept. This power is necessary, since many of the primitives take variadic argument lists, with complex constraints. The BNF grammar is shown in Listing 5.9. We have included some examples of type specifications for our primitive funtions in Table 5.5.

Listing 5.9: BNF Grammar for Primitive Type Specifications

```
<typespec> ::= (or <type> <type>*) ; One of these must match.
             |  (* <typespec>)      ; Any number of repetitions.
             |  <typeseq>           ; Match each typespec in order.
             |  ()                  ; No more arguments.
             |  t                   ; Any type.
             |  <type>              ; Next arg is of type <type>.
```

5.6 Asynchronous REPL

Of the additional, optional features we added, the most valuable was an asychronous REPL. We didn't intend to implement parallel scheduling, so the whole system would run in a single thread. However, we needed the REPL to wait for user input, while the rest of the game kept running.

We considered using non-blocking I/O, but we had already implemented the Lisp reader in a way that was not compatible with that approach. The solution we went with was to run just the REPL in a dedicated thread, with a lock on the Lisp memory allocator, and no other protection. Given how infrequent commands run from the REPL are, this approach worked with only the occasional minor bug, which we deemed adequate, especially considering how simple it was to implement.

Chapter 6

Implementation

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.
>
> Greenspun [15].

In this section we discuss how we implemented the ECSQL system, and how we came to make certain decisions about what approaches to take.

We chose C as our implementation language. It made sense to use a systems language, since a primary aim of the project was to tackle usability issues with ECS libraries written in these languages. We used C over a more featureful language like C++ because we did not expect to benefit from most of those additional features, and believed they would only distract us.

In addition to the C standard library, we used Klib [4] for dynamic arrays and hashmaps, and Raylib [37], which provides I/O, a drawing API, and a basic vector maths library.

In retrospect, we created unnecessary work for ourselves by using C in-

stead of C++, especially when using Klib's verbose hashmap API. On the other hand, it did make deciding how to implement certain features easier, since C provided far fewer options than C++ would have.

6.1 Entities

As mentioned in subsection 5.4.1, Entities are one of the built-in types of Lisp objects. As such, we decided to work with Entities in C stored as Lisp objects. This obviously came with some reduction in type safety, since any type of Lisp object could be passed to ECS functions that expect an Entity. On the other hand, this made interoperating with Lisp code much easier, since we could pass Entities back and forth, without having to convert them between the Lisp object type and a dedicated Entity type.

We used two data structures to track Entity liveness: the live set and the Generation map. The live set simply stores the IDs of all Entities that are currently alive. The Generation map stores the Generation value for each Entity ID. To save space (and time at startup), we fill in the Generation map lazily, by initialising the Generation for an Entity ID to 0 the first time it is requested.

When an Entity is created, its ID is added to the live set. We can then produce a Lisp object to represent it. This contains its ID, and the corresponding Generation value. The code works approximately as shown in Listing 6.1. The `next_entity` variable allows the search for a free Entity to start from where the last one left off, rather than first checking all the smaller ID values, which are much more likely to be in use.

To destroy an Entity, we remove its ID from the live set and increment the corresponding Generation count.

Listing 6.1: A Function to Create an Entity.

```
Object new_entity_id(World *world) {
  u32 id = world->next_entity;
  /* id <- next available ID from world->next_entity */
  world->next_entity = id + 1;

  /* Add id to live set. */
  kh_put(live, world->live, id);
  /* ... */

  return ENT_BOX(id, *ecs_generation(world, id));
}
```

The live set is primarily used to check if an ID is available when creating an Entity. To test if a "complete" Entity (with ID and Generation) is alive, we check if the Generation map's entry for that ID matches the Generation stored in the Entity (see Listing 6.2). We increment the Generation when we destroy an Entity, so only live Entities can satisfy this check.

Listing 6.2: Entity Liveness Test

```
bool ecs_alive(World *world, Object entity) {
  return *ecs_generation(world, entity.id) == entity.gen;
}
```

6.1.1 Entity Names

A name is a unique, human-readable identifier for a particular Entity. The fact that they must be unique, in addition to the fact that we need to be able to look up an Entity by its name, means that it actually wouldn't make much sense to implement names as Components. Instead, we use a hash map from names to Entities. This is efficient because names are symbols, each of which has a unique 64-bit representation in our Lisp environment (see subsection 6.4.1), so we can use integer hashing.

We store the Entities that names map to with their Generation counts, and automatically remove a name if the Entity it references is no longer alive when attempting to look it up.

6.2 Components

We implemented the Component store essentially as described in the design, with a few complications. It is Archetype-based, with a concrete data structure representing each Archetype (see Listing 6.3).

The `Column` type is a generic dynamic array, in which the size of elements is determined at run-time. This is less efficient than an implementation where the element size is known at compile time, but this trade-off was necessary because we don't know what Archetypes will exist at compile-time, and some Components aren't even defined until run-time.

Listing 6.3: Our Archetype type definition.

```
typedef struct Archetype {
  ArchetypeID id;
  Type type;
  kvec_t(size) component_columns;
  kvec_t(EntityID) entities;
  kvec_t(Column) columns;
  khash_t(archetype_edge) * neighbours;
} Archetype;
```

The `ArchetypeID` type acts as a unique identifier for a specific Archetype, which we can expose to code outside the Component store without making the implementation of Archetypes public.

The Entity list (`entities`) is effectively another Column that every Archetype has, and always stores the Entities themselves.

6.2.1 Adding and Removing Components

The processes of adding and removing a Component are mostly the same, to the point that both are thin wrappers over the same function: `move_entity()`. This function can move an Entity from any Archetype to any other Archetype, and copies across the values of Components that are in both. It adds the Entity to the new Archetype before removing it from the old one so it can copy the Component data.

The Component data for all Entities in an Archetype is stored entirely in contiguous arrays, so adding an Entity to an Archetype is as simple as increasing the length of each Column by 1, and putting the Entity's ID and Components in the last row.

Removing an Entity from an Archetype is a little more complicated, since doing so can create a hole in the contiguous rows of Component data. To fix this, we move the Entity in the last row into the row that was vacated, along with all its Component data. This requires us to update that Entity's row value in the Entity Index. The alternative would be moving every Entity after the hole back by one, which would obviously be less efficient. This approach was suggested by Morlan [32].

6.2.2 Component-Column Mapping

Not every Component contains data. We should only allocate Columns to the ones that do. We represent these allocations using an array (`component_columns`) in each Archetype; it contains the index of the Column allocated to each Component, or -1 if the Component doesn't have data. We took this simple approach, rather than using something like a hash map, because Archetypes generally have fewer than 10 Components, so anything more complicated

would have been excessive.

If a Component should contain data, we add the `Storage` Component to it. `Storage` contains the size of the data required for a Component, which is used as the element size in the corresponding Column when initialising an Archetype.

6.2.3 Bootstrapping the `Storage` Component

The technique described in subsection 6.2.2 works for most Entities and Components, but has a slightly involved set-up process. The `Storage` Component contains the size of the Component it is added to, so *it must be added to itself*. This creates a cyclic dependency, since `Storage` must be initialised before it can be added to Entities, but it has to be added to itself as part of that initialisation!

Our solution for this is as follows:

1. Create the `Storage` Entity, which will initially be in the "empty" Archetype, since it has no Components.

2. Create the Archetype for Entities containing only `Storage` (`[Storage]`). Since `Storage` doesn't have the `Storage` Component, this will contain no Columns.

3. Manually add the Column for `Storage` (with the correct element size) to that Archetype, and point the appropriate (only) entry in `component_columns` at it.

4. Add the `Storage` Component to itself as normal, and it will move into the `[Storage]` Archetype. Set the value of its newly-created `Storage` Component to `sizeof(struct Storage)`, and we can safely add `Storage` to other Entities that we want to represent Components.

6.2.4 Lisp Components

We did not consider it essential to be able to access every Component's data in Lisp. For the ones that do need to be accessible from Lisp, we created the `LispComponentStorage` Component (see Listing 6.4). It contains all the information necessary to give Lisp code access to the value of a Component value stored in the ECS.

Listing 6.4: `LispComponentStorage` Component.

```
enum LispComponentStorageType {
  STORE_OBJECT, STORE_STRUCT, STORE_UNBOXED};

struct LispComponentStorage {
  enum LispComponentStorageType type;
  u16 struct_id;
  size size;
  enum ObjectTag object_type;
};
```

There are three storage formats for Components:

Struct A struct, with the struct type ID in the `struct_id` member, and the struct's size stored in `size`.

Object A full, boxed Lisp object, with the Object type tag in `object_type`.

Unboxed A single, raw value, such as an integer or floating-point number. The Object type tag for these is also stored in `object_type`.

All Components with the `LispComponentStorage` Component are also expected to have `Storage`, so the `size` member may seem redundant. On the other hand, it allows us to exchange Component data between Lisp and C without reading `Storage`, which saves an ECS lookup in Lisp primitives like `ecs-get`.

6.3 Queries & Systems

Our implementation of Queries is split between C and Lisp code. The Lisp code mainly serves to provide a frontend to the C code, that is more suitable for interactive use.

6.3.1 Query Compilation

The `translate-predicate` function in `query.lisp` compiles a Query of the form defined in Listing 5.5 into the separate predicate and binding list forms defined in Listing 5.6 and Listing 5.7. The function is somewhat involved, and performs a substantial amount of list structure manipulation, so we benefited greatly from implementing it in Lisp. It is also only called at most once per Query, so its performance is not of critical importance, so we had little reason to implement it in C.

We now consider Listing 6.5 as an example, to illustrate the behaviour of `translate-predicate`. The Query `(and (has A) (opt B) C)` is compiled, generating a pair containing the predicate and the binding list forms[1].

The binding list, `((opt #*entity(26 0)) #*entity(29 0))`, contains references to Entities B (26) and C (29). Component A (25) is not bound, since it is wrapped in a `has` expression in the Query.

The predicate, `(and #*entity(25 0) (and) #*entity(29 0))`, only references Entities A & C, because B was inside an `opt` expression. The empty `(and)` predicate is generated because the Query `(opt B)` has no requirements, and `(and)` happened to be a conveninent way to express that.

The generated function, in the second argument to `ecs-do-query`, takes

[1]In this expansion, we have used the dotted pair notation to clearly distinguish the binding list and predicate, although this is not how it would normally appear when printed, because the predicate, a list, is in the `cdr` of the pair.

the Entity and the bound Component values as arguments. The implementation of `ecs-do-query` determines what values these should be, based on the Query (see subsection 6.3.3).

Listing 6.5: ECSQL Macro Expansion.

```
* (list A B C)
(#*entity(25 0) #*entity(26 0) #*entity(29 0))
* (ecs-add* (ecs-new) A (B 2.0) (C 'cool))
#*entity(27 0)

* (ecsql (and (has A) (opt B) C) (b c)
         (print (list entity b c)))
(#*entity(27 0) 2.000000 cool)

* (macroexpand-1
   '(ecsql (and (has A) (opt B) C) (b c)
           (print (list entity b c))))
(ecs-do-query
 '(((opt #*entity(26 0)) #*entity(29 0))
   . (and #*entity(25 0) (and) #*entity(29 0)))

 (lambda (entity b c)
   (print (list entity b c))))
```

6.3.2 Query Execution

The function `ecs_do_query` in `query.c` takes a compiled Query and a System function pointer, and calls the function for each Archetype that matches the Query's predicate. It also generates an `EcsIter` object based on the binding list. This contains an array of the Column numbers of the Components that are bound, providing an easy way for C System functions to access the matched Component data, as shown in Listing 6.6.

In `move`, `Pos` is the first Component bound in the Query, and `Vel` is second, so the system can access them by passing 0 and 1 respectively to `ecs_iter_get`. We adapted this API from Flecs [29]. Notice that, since we

know the types of the Components in this context, we can cast the (void) pointer returned by `ecs_iter_get` to that type, allowing us to access the Component data in a very clean way.

A System function, or `SystemFunc`, implements a transformation to apply to the Entities matched by a Query. A C System function is any function with the same type as `move` (see Listing 6.6). A Lisp System function takes an Entity as its first argument, followed by the (values of the) Components it uses. They aren't necessarily attached to Systems, and may only run once, as is the case with the functions generated by the `ecsql` macro.

Listing 6.6: A C System function.

```
typedef void(SystemFunc)
            (LispEnv *lisp, struct EcsIter *iter, void *data);
/* Query: (select Pos Vel) */
void move(LispEnv *lisp, struct EcsIter *iter, void *data) {
  struct Vector2 *poss = ecs_iter_get(iter, 0);
  struct Vector2 *vels = ecs_iter_get(iter, 1);
  size N = ecs_iter_count(iter);
  float delta = GetFrameTime();
  for (size i = 0; i < N; ++i) {
    poss[i].x += vels[i].x * delta;
    poss[i].y += vels[i].y * delta;
  }
}
```

Since `ecs_do_query` only runs on C System functions, we need a wrapper for it to work with Lisp System functions. The approach we take is to call `ecs_do_query` with a special C System function called `lisp_run_system`, and pass the Lisp function (in the form of a Lisp object) as the `data` parameter. The C function constructs a Lisp argument list, and calls the passed Lisp function for each Entity in an Archetype. We chose to perform the iteration in C rather than Lisp for efficiency.

The `lisp_run_system` does most of the work of determining the types and sizes of the Components bound for a whole Archetype up-front. The only work it needs to do for each Entity is to put its ID and bound Components

into the argument list, and call the Lisp System function.

Previously, we used a Lisp macro to generate `ecs-get` calls that ran for every Entity, so `lisp_run_system` only needed to pass the Entity ID to the Lisp System function. This was unnecessarily inefficient, since it would have to perform the same type checks and logic to get the right Component value for every Entity in an Archetype.

6.3.3 Systems

A System is an Entity with the `Query`, `System` and `SystemData` Components. The `ecs_new_system` function is a convenient wrapper that takes the values for those Components, and adds them to a new Entity, as shown in Listing 6.7. That listing also shows how we add Phase Components (see subsection 5.3.1), using the `ecs_add` API. We add the same Components to Systems written in Lisp, such as the functionally equivalent one in Listing 6.8; note how concise this definition is, especially in comparison to the equivalent C code, spread across Listing 6.6 and Listing 6.7.

Listing 6.7: Movement System initialisation.

```
Object move_system = ecs_new_system(lisp,
    LISP_EVAL_STR(lisp, "(select␣(or␣RelPos␣Pos)␣Vel)"),
    move, NULL);
ecs_add(world, move_system,
        ecs_lookup_by_name(world, SYM(lisp, "Physics")));
```

Listing 6.8: Lisp Movement System initialisation.

```
(ecs-new-system
 (Physics)
 (and (or RelPos Pos) Vel)
 (pos vel)
 (let ((delta (get-delta)))
   (set-v2 pos
           (+ (v2-x pos) (* (v2-x vel) delta))
```

```
(+ (v2-y pos) (* (v2-y vel) delta)))))
```

System scheduling works as described in subsection 5.3.1 (see Listing 6.9). The "manual" function call to execute a Phase has the advantage of making it simple to integrate ECSQL-based Systems with code from other libraries, such as the begin/end drawing calls from RayLib, in this example.

Listing 6.9: The main loop of a basic ECSQL game.

```
Object physics_query =
     LISP_EVAL_STR(lisp,
      "(select␣System␣Query␣SystemData␣(has␣Physics))");
Object graphics_query =
     LISP_EVAL_STR(lisp,
      "(select␣System␣Query␣SystemData␣(has␣Graphics))");
ecs_do_query(lisp, physics_query,
             run_matching_systems, NULL);
/* ... */
while (!WindowShouldClose()) {
  ecs_do_query(lisp, physics_query,
               run_matching_systems, NULL);
  BeginDrawing();
  ecs_do_query(lisp, graphics_query,
               run_matching_systems, NULL);
  EndDrawing();
}
```

This approach was extremely simple to implement, and sufficient for our use-case. It would also be amenable to parallelisation, since Systems with the same Phase Component could theoretically run concurrently.

6.4 Lisp

As explained in section 3.2, our plan was to use the power of Lisp, especially its macro system, to implement a high-quality Query language with minimal wasted effort. We also decided to apply this thought process to the Lisp implementation itself. Macros are such a powerful language feature that

Lisp "can be bootstrapped up from essentially nothing" [16, p. 13] using them. This meant we could implement an evaluator for an excruciatingly simple language in C, and build the rest of the features of a useful language with macros.

6.4.1 Object Representation

To support dynamic typing, it must be possible to represent a value of any type with a fixed-length value. We researched two existing approaches. Engelen [8, p. 7] uses NaN boxing, with a 64-bit value either holding a valid double-length floating point value or a pointer to a value in memory, stored in the unused bits of a NaN floating point value. Queinnec [35, p. 391] uses the least significant bit to indicate whether a 32-bit object contains a 31-bit integer, or a pointer to an object in memory.

We chose to use a tagged union representation (see Listing 6.10 and Listing 6.11). In this scheme, we use a 5-bit "type tag" to represent the type of the object, and use the remaining 59 bits for type-specific data. This has the advantage that we only need the immediate value of an object to determine its type, so type checks don't need to read memory.

One issue with the C type definition that we couldn't fix is that only integer types can be bit fields. As a consequence, we have to represent the `tag` as a `u8`, even though we have an enumeration type, `ObjectTag`, for this purpose.

Listing 6.10: Our Lisp Object Memory Layout

```
+-------------------------------+-------------------------------+
<-----------------------Data (59b)-------------------------><typ>
<-------Entity (32b)----------><---Gen (16b)-->00000000000<typ>
00000000000000000000000000000<---------Integer (32b)---------><typ>
00000000000000000000000000000<----------Float (32b)----------><typ>
<-----------Index   (43b)----------------><-Metadata(16b)><typ>
```

Type	Metadata	In Memory
`string`	Length	Characters
`symbol`	Name Length	Characters
`pair`	Length (2)	The `car` and `cdr`.
`closure`	Length (2)	Body (pair)
`vector`	Length	Contents
`struct`	Struct type	Struct members

Table 6.1: In-Memory Lisp Object Representations

Listing 6.11: Lisp Object C Type Definition

```
enum ObjectTag {
  OBJ_NIL_TAG = 0, OBJ_STRING_TAG, /* ... */
  OBJ_CLOSURE_TAG, OBJ_ENTITY_TAG
};
#define OBJ_TAG_LENGTH (5)
typedef union Object {
  u64 bits;
  struct {
    u8 tag : OBJ_TAG_LENGTH;
    u64 val : 59;
  };
  struct {
    u8 : OBJ_TAG_LENGTH; /* enum ObjectTag */
    u32 metadata : 16;
    /* Must be signed to support indirect addressing */
    i64 index : 43;
  };
  struct {
    u16 : 16;
    u16 gen;
    EntityID entity;
  };
} Object;
static_assert(sizeof(Object) == sizeof(u64));
```

For types that can't fit within these 59 bits, we store an index into Lisp memory (see subsection 6.4.3), with some metadata specific to each type. The types, metadata, and format in memory are listed in Table 6.1.

We initially tried to use the full 59 data bits to represent floating-point

numbers and signed integers, but ran into some strange bugs, so we restricted those to 32 bits. The numbers of bits used to represent Entities and in-memory object metadata were chosen mostly arbitrarily.

At first, we did not have an explicit type definition for the Lisp Object type. Instead, we just defined `Object` as an alias for `u64`, and used shifting and masking operators to pack and unpack the contents of the object. We did this because we couldn't work out how to implement this representation using bit fields. That approach was extremely error-prone, and we were fortunately able to convert to this implementation later on.

We treat the `val` struct member as a typeless buffer, and directly copy the bytes of object data in and out of it. This approach was retained from the code that worked with the old type definition, and we saw no reason to change it.

It is important for the Lisp `eq` operator to be efficient, since it is used often, both in the implementation, and actual Lisp code. This representation allows us to implement `eq` with a literal equality test in C, which is about as cheap as possible.

Listing 6.12: Lisp `eq` operator.

```
static inline bool EQ(Object x, Object y) {
  return x.bits == y.bits;
}
```

Symbols

We represent symbols by storing string objects that contain their names, then replacing the string type tag with the symbol type tag. This allows two symbols with the same name to remain distinct, by storing the name string in two distinct locations in Lisp memory. However, this is usually undesirable.

The default behaviour (for example, when reading), is to "intern" symbols, adding them to the Lisp environment's symbol table [13, § 8.4]. A symbol's name is looked up in the symbol table, and added if it is not yet there. A symbol table entry contains the unique "interned" symbol with a given name.

There are situations when Lisp programmers want to generate a symbol that has certainly never been used before, for example, when writing certain macros [13, p. 166]. Our implementation makes this simple, since we can just replace the type tag of a newly-stored string with that of a symbol. This is how our `make-symbol` Lisp primitive is implemented.

Closures

Closures are callable Lisp functions. A closure is represented as a list, containing the lexical context in which it was created, the function argument list, and the function body.

Primitives

Primitives are Lisp functions implemented in C. Each primitive has a name symbol, and the actual representation of a primitive function object is that symbol, with the type tag replaced by the primitive type tag. Primitive identifier objects are used as keys in a look-up table that stores the argument type-spec and function pointer for each primitive.

Structs

Listing 6.13: A 2D Vector Lisp Struct

```
(defstruct v2
  (x f32)
```

```
(y f32))
```

When a struct type is defined in Lisp, it is assigned an ID, which is stored in the metadata fields of structs of that type. This allows us to have up to 2^{16} Lisp struct types, which is more than enough.

We initially implemented structs as vectors of Lisp objects, one for each member. This was easy to implement, but was memory-inefficient since we didn't need all the type information for every struct member, because each member has a known, fixed type.

In our current implementation, Lisp structs have the same memory layout as C structs with equivalent type declarations. As a result, Lisp and C code can work with structs stored in the exact same format. This is extremely valuable because it allows Lisp Systems to directly manipulate data stored in the ECS Component store.

To match C's struct memory layout, struct members are packed together, in the order they were declared in the struct, with padding inserted where necessary so that each member has adequate alignment. Where possible, struct members are stored as just their raw data. For example, the struct in Listing 6.13 occupies 8 bytes, 4 for each float.

Each getter and setter method performs the necessary boxing and unboxing operations to abstract away the underlying representation of struct members.

We mostly implemented support for Lisp struct definitions in `struct.lisp`. The main part of this feature implemented in C was the `--struct-*` helper primitives, which perform low-level manipulation of Lisp object representations, which is not possible within the Lisp code itself (see Appendix A). We took this approach because implementing this feature involved *a lot* of complex Lisp code generation, which would have been much more challenging to

implement without using Lisp macros.

We store metadata for each Lisp struct type, such as its size, and the types of its members, as a Lisp vector. This is easier to work with in Lisp code than an opaque C data structure, and we rarely need to use it in C anyway. Ideally, we would represent this metadata as a struct, but this isn't possible because we need to use it in the code that implements structs in the first place.

6.4.2 Parser and Printer

The lexer and parser were mostly trivial to implement, at scarcely 100 lines each. They operate on `FILE*` streams, so the same code works for reading input from the user (standard input), files and even strings thanks to the POSIX `fmemopen()` function.

The `print` family of functions are meant to produce a printed representation of a given object. We implemented a printer in C and Lisp, primarily to demonstrate how much more concise an equivalent function can be in Lisp. The Lisp printer does fall back to the C printer for primitive types like integers, which it cannot print.

6.4.3 Memory and Addressing

Lisp memory is addressed with 43-bit values (see subsection 6.4.1). These values are used as indices into a large, contiguous array of 64-bit memory *cells*. Most Lisp data is stored in increments of 64 bits (the size of 1 object), so allocating data like seemed like a reasonable approach. This does waste up to 7 bytes per allocation that isn't an integer number of cells, but those are a minority.

Memory is allocated linearly. There is a pointer to the start of the free region, and each time an allocation is requested, the pointer is moved forward by the amount requested (plus some padding for alignment). This was easy to implement, though there is currently no way to reclaim memory that is no longer in use, so eventually the system runs out of memory and halts. Fortunately, the memory allocation rate is low enough that this doesn't happen for a decently long while, especially if there are no Lisp systems running. If we had more time, an obvious first step would be to implement a garbage collector (see section 9.1).

One major problem with this approach was that it provided no mechanism for referencing data stored outside Lisp memory, since 43 bits is insufficient to store a pointer. This would have meant Lisp code could not directly manipulate Component data stored in ECS Columns. We considered copying the data back and forth each time we wanted to change it, but this would have been awkward to implement, even if we hid it behind primitives and macros. Instead, we implemented a scheme for storing full-size pointers in Lisp memory.

Though 43 bits is insufficient to store a pointer, it was more than we needed for indexing Lisp memory. Additionally, we only needed to use positive values to represent indices into memory. Given these properties, we devised a simple scheme for storing pointers: If an index is negative, negate it, and the cell at that (positive) index contains a pointer to the data. This adds one check to every normal memory access, but considering the check and branch should compile to around 2–3 instructions, the extra computation is inconsequential, relative to the other, much less efficient parts of the Lisp implementation.

6.4.4 Error Handling

We implemented the `wrong` primitive described in subsection 5.4.6 using C's `setjmp` and `longjmp` functions. These are the closest thing to exceptions in the language. We call `setjmp` at the start of the REPL function, with a `jmp_buf` (that stores where `setjmp` was last called) stored in the Lisp environment data structure. When an error occurs, the `wrong` function is invoked, either in C or Lisp, and it calls `longjmp` to unwind the stack back to the start of the REPL function.

This design makes the simplifying assumption that Lisp errors only occur in the REPL. They are certainly more likely to happen there, but a more robust solution would have been preferable, since the assumption is broken by running the REPL in a separate thread from the rest of the game (see section 5.6).

6.4.5 Scopes

We have global scopes/namespaces for variables, functions, macros and structs. These are all implemented as hash-maps from symbols to the relevant type for each (data for variables, primitives or closures for functions and macros, struct metadata for structs). The `defname` primitive provides a mechanism for Lisp code to add a definition for a given symbol to any of these global namespaces.

Local variable scope is implemented as a stack of association lists, itself represented as a list, with the top of the stack at the front of the list. Variable lookup works by first searching for the name in local scope, top to bottom, then searching in global scope. For example, the code in Listing 6.14 would have a local scope represented as shown in Figure 6.1.

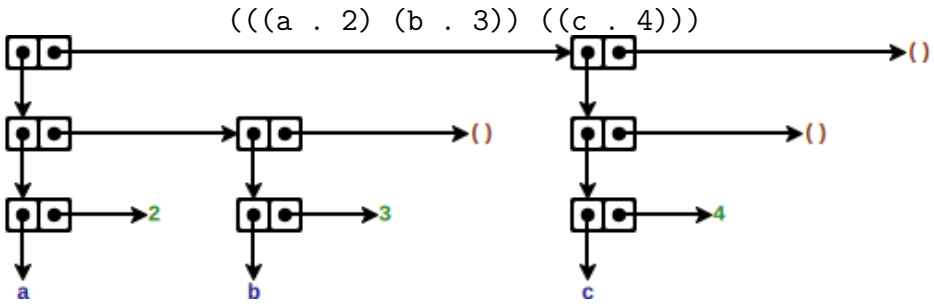

Figure 6.1: The list structure of a local variable scope.

Listing 6.14: Lisp variable scope example.

```
(let ((c 4))
  (let ((a 2) (b 3))
    (* a b c)))
```

When a closure is created, it is given a reference to the local scope at that point. Since scopes are built with cons cells, allocated in Lisp memory, they can persist after the function where they were created. This made it trivial to implement lexical closure scope correctly.

6.4.6 Evaluation

The `lisp_evaluate` function attempts to evaluate any Lisp object of any type. There are 4 main cases for how objects are evaluated: literals, variables, applications and special forms. A literal is anything that is neither a list nor a symbol, and evaluates to itself. Symbols are interpreted as variable names, and are looked up as described in subsection 6.4.5. They evaluate to the bound variable value, in the current context.

The special list forms are listed in subsection 5.4.8. Each is evaluated according to the rules in that section.

If a list form isn't special, the evaluator treats it as a function application. The first element of a function application form is the name of the function,

and the remaining elements are its arguments. The function application evaluation procedure is shown in algorithm 3. The arguments are evaluated in the current context, and the called function is evaluated in a separate context.

Algorithm 3: Function Application Form Evaluation

```
Input: fname;                              /* Function name.          */
Input: argforms;                   /* Argument form list.          */
Input: context;            /* Current evaluation context.          */
fn ← value for fname in function namespace;
arglist ← result of evaluating argforms in the input context;
if fn is a primitive then
    if arglist matches fn's typespec then
        return result of passing argforms to fn's C function.
    else
        Raise an exception;
    end
else
    bindings ← List of pairs of fn's parameter names and elements
      of arglist;
    if binding failed then
        Raise an exception;
    end
    return result of evaluating fn's body with bindings as the lexical
      context.
end
```

For argument list binding, we use a trick from Engelen [8, p. 15]: Binding initially pairs up elements of the parameter and argument lists; when the last element of either is reached, if it is not nil, the remainder of the other list is bound to it. As a result, we can use the dot operator to define functions that take variable numbers of arguments, or bind the elements of a list to multiple function parameters. The following examples illustrate this functionality.

```
* (defun max (a . as)
    (if (and as
             (< a (max as)))
```

```
        (max as)
      a))
* (max 1 2 3)
3
* (let ((args (list 1 2 3)))
    (funcall #'+ . args))
6
```

6.4.7 Macro Expansion

We kept the macro expansion procedure shown in algorithm 2 simple for illustrative purposes. There were additional issues we needed to tackle to produce a correct implementation.

First, macros can expand into expressions containing more macro expressions, so we have to iteratively expand each expression. We compare an expression to the result of applying macro expansion to it, and stop when they are the same.

We also encountered an unexpected exception to the rule of macro expansion: the argument lists of `lambda` expressions are never expanded, even if they aren't quoted. In Common Lisp, `lambda` is a macro Graham [13, p. 402]. We could have implemented a `lambda` macro of our own, that could generate a form that would protect the argument list from expansion, such as a closure. However, we found it more straightforward to simply hard-code the exception into the macro expansion code itself.

6.4.8 Documentation Strings

One useful piece of feedback we got from evaluation day was that this system would be easier to use with a tutorial or documentation. In response, we implemented support for documentation strings [13, p. 100]. The `describe`

function allows the user to access the documentation string of any object that has one (see Listing 6.15).

Since there is some common information that every documentation string for certain types of object would contain, such as the types of struct members, we implemented some macros to automatically generate portions of the documentation strings for those objects.

We implemented this feature entirely in Lisp.

Listing 6.15: Documentation strings example.

```
* (describe 'v2)
v2
symbol
A struct type.

Members:
- x: f32
- y: f32

* (describe 5)
5
i32
* (describe 'describe)
describe
symbol
Prints some information about the supplied OBJECT.
This comprises its value, its type, and its
docstring if it has one.

Function arguments: (object)
```

Chapter 7

Project Management

In this chapter, we evaluate how we managed the development of the project. We also perform a risk evaluation, in the context of the approach we decided on.

7.1 Project Progress

In November, we produced the schedule shown in Figure 7.1 for the Progress Report. At that point, we still expected the language to have limited expressive power, and a static type system, so we thought it would make more sense to implement the ECS first. However, after deciding to use Lisp as the basis for the language, we realised it would make more sense to implement the Lisp interpreter first; for one thing, this allowed us to use Lisp objects to store Entities and Entity names. As a result, we immediately deviated from the original timeline.

Based on Figure 7.2, we believe we worked at a consistent pace throughout the development of the project. We ended up slightly behind the expected progress line (with an expected completion date of the 15th of March, the

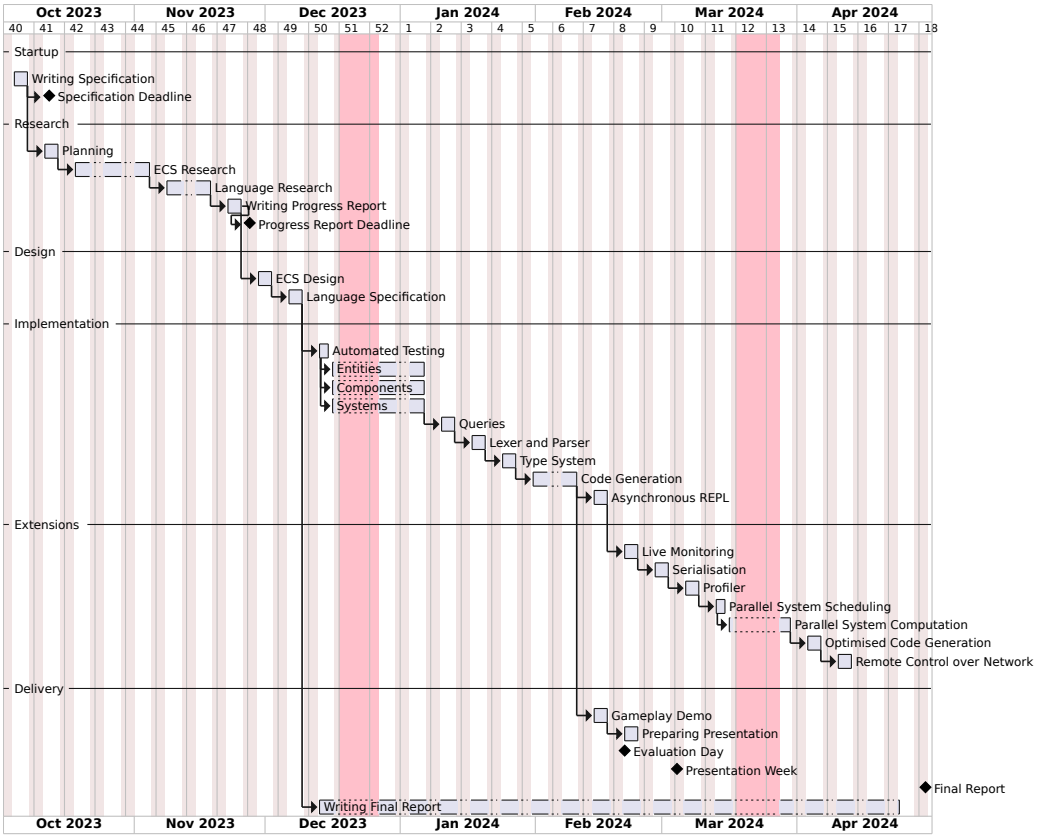

Figure 7.1: Project Timetable, created in November

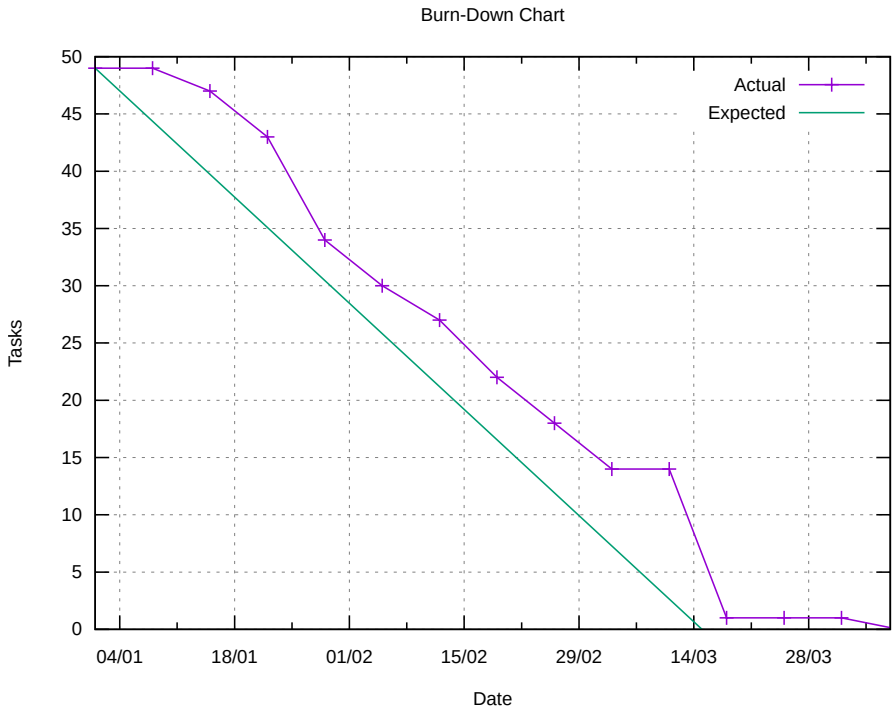

Figure 7.2: Project Burn-Down Chart.

Factor	Harm	Probability	Risk	Mitigation
Hardware Failure	3	3	9	Daily Backups
Schedule Slippage	1	5	5	Flexible scope
Regression Bugs	3	4	12	Automated testing
Remote Code Execution	5	2	10	Input validation

Table 7.1: Risk Management Plan Summary

day of our presentation) because we were more preoccupied by other commitments in week 1 of term 2, and we were still planning our approach at that point. This graph is only an approximation, and the sudden drop around the 14th of March happened because we looked back at our still-open TODO items in the week before the presentation, and realised a lot of them were either already complete, or no longer relevant, so we closed almost all of them at once. We took a break after the presentation, hence the delay before the we closed the final task.

7.2 Risk Management

At the start of the project, we performed a risk analysis, and established a plan to mitigate or prevent each risk factor (see Table 7.1). We rated the harm and probability of each risk factor on a 1–5 scale, and calculated the risk of each factor by taking the product of these ratings. We can now evaluate the success of this plan.

We didn't encounter any notable hardware failures, and we didn't implement a network interface for ECSQL, so we didn't have to implement our plan for preventing remote code execution. However, on a related note, it would still make sense to take precautions like removing the REPL from the release build of a game, to prevent players from gaining complete control over it.

As explained in section 4.4, we didn't implement proper automated testing, so it was slightly harder for us to detect bugs. Fortunately, the interactive development approach we adopted instead meant that when we did find a bug, we could fix it quickly, in most cases. One occasion when this approach didn't work as well was when we discovered a critical bug on the morning of evaluation day. However, that bug occurred because we were rapidly producing new code, and stopping to create tests would have slowed us down, which might have prevented us from fixing it before our time slot.

We already explained the flexible scope of our project in section 4.3. Thanks to it, we were able to accommodate an unexpectedly high workload during term 1, by simply dropping some of the less valuable extensions, and moving the whole development timeline later. As a result, we carried out most of the development work in term 2, when we had a much lower workload outside the project, and could consequently make rapid progress. This also gave us time to carry out rigorous research and design for the MVP, which was essential to the Waterfall methodology we used to implement it. Overall, this risk mitigation plan was a success.

Chapter 8

Results & Evaluation

We now consider the results of the project, with a focus on theoretical and concrete use-cases for the system we have produced. We also evaluate the success of the project, with regard to our objectives.

8.1 Example Application

The aim of this project was to produce a game development tool, so we decided the best way to demonstrate the result was to create a "game" using it. This demo has the following features:

- 2D particle simulation, with inelastic collisions.
- Three major Entity categories ("species"), distinguished with Tag Components: `Dwarf`, `Elf` and `Goblin`.
- Default parameters for instances of each species.
- An initial scene, defined in Lisp (see `examples/planets-scene.lisp`).
- Parameter viewing functionality.

- Support for scene hierarchies, with relative motion.

To compile and run the system, run the following commands:

```
/ecsql $ mkdir cmake && cmake -S . -B cmake
/ecsql $ cmake --build cmake -j
/ecsql $ cmake/Ecsql
```

8.1.1 C Systems

The demo includes the following set of C Systems, implemented in `main.c`:

Movement Moves Entities with `Pos` (position) and `Vel` (velocity), as shown in section 6.3.

Edge Collisions Bounces Entities off the edges of the screen.

Entity Collisions Bounces Entities off each-other. This is a "pairwise" System, a variant that has two Queries, and runs a System function on each distinct pair of Archetypes where one Archetype matches each Query.

Point Gravity When middle click is held, all Entities will accelerate towards the position of the mouse cursor.

Drawing Draws a circle at the position of each Entity that satisfies the Query `(and Pos (opt Colour) Radius)`. For Entities with no `Colour`, it defaults to white.

8.1.2 Scene

Initially, there are no Entities that match the drawing System's Query, so the window is blank. We can load in an initial set of Entities by loading the

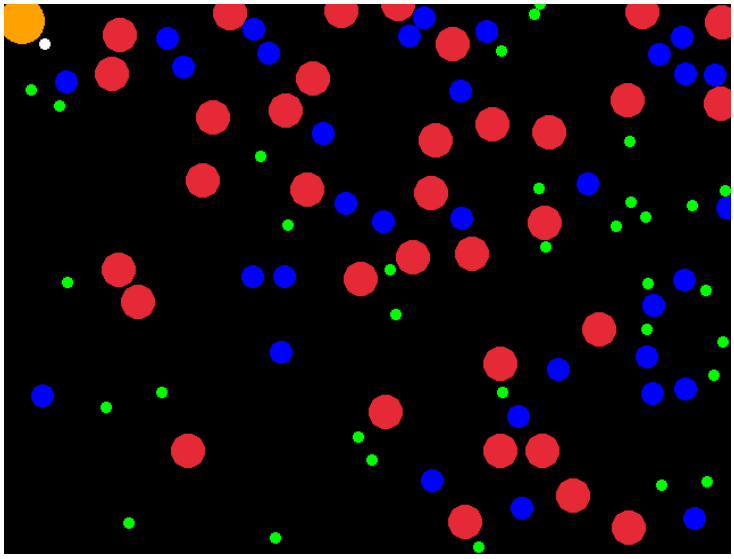

Figure 8.1: Initial Demo Scene

`demo.lisp` script, which runs all of the scene setup code for this subsection, the result of which is shown in Figure 8.1.

Listing 8.1: Loading `demo.lisp`.

```
* (load "examples/demo.lisp")
```

The red, blue and green circles in Figure 8.1 have the `Dwarf`, `Elf` and `Goblin` Tags respectively. These Components are still Entities, so we can add any Components we like to them, as shown in Listing 8.2.

Listing 8.2: Creating the Goblin template.

```
(defvar col-green (make-colour 0 255 0 255)) ; RGBA
(ecs-add* (defcomponent Goblin nil)
          (Colour = col-green)
          (Radius 5.0)
          (Mass 0.5))
```

When we then create "normal" Entities, they can inherit the values of these Components using the `:` operator in the `ecs-add*` DSL (see subsection 5.5.2). The code in Listing 8.3 creates 100 Entities, each at an offset

position from the last, cycling between creating a `Dwarf`, an `Elf` or a `Goblin`. Each Entity inherits `Radius`, `Colour` and `Mass` from its species Tag Component. We also add the value of `species` as a Component, using the `expr` form.

Listing 8.3: ECS Scene Creation.

```
(let ((species-vec (vector Dwarf Elf Goblin)))
  (dotimes (i 100)
    (let ((species (aref species-vec
                         (% i (length species-vec))))
          (screen-width (get-screen-width))
          (screen-height (get-screen-height)))
      (ecs-add* (ecs-new)
                (expr species)
                (Pos (+ 0.0 (% (* 20 i) screen-width))
                     (+ 0.0 (% (* 30 i) screen-height)))
                (Vel (* 20.0 (+ 1 i)) (* 15.0 (+ 1 i)))
                (Bounce 0.8)
                (Radius : species)
                (Colour : species)
                (Mass : species)))))
```

8.1.3 Queries & Lisp Systems

The main novel feature of ECSQL is the ability to run one-off, programmatic Queries that can arbitrarily modify the state of Entities. Given its importance, we will highlight some examples.

Listing 8.4: Colour all Dwarves and Elves yellow (see Figure 8.2).

```
(ecsql (and Colour (has (or Dwarf Elf))) (colour)
       (set-colour colour
                   255 255 0 255))
```

Listing 8.5: Move all Entities towards (200, 200) (see Figure 8.3).

```
(ecsql (and Pos Vel) (pos vel)
       (set-v2 vel
               (- 200 (v2-x pos)) (- 200 (v2-y pos))))
```

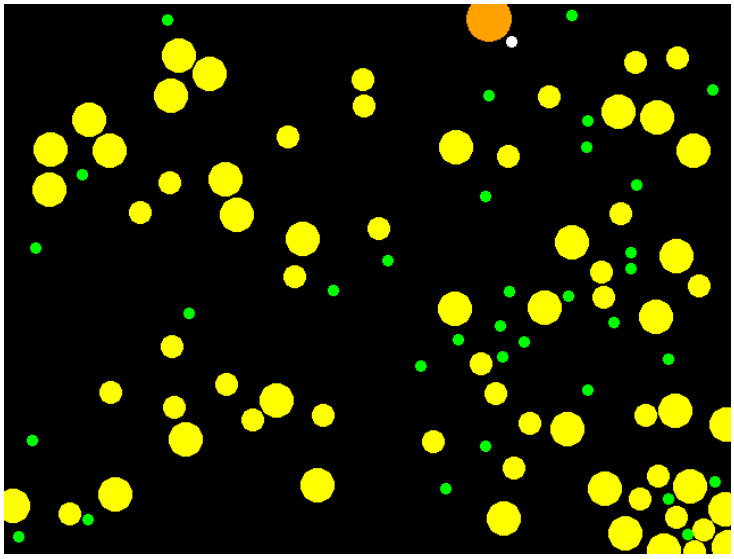

Figure 8.2: Dwarves and Elves coloured yellow.

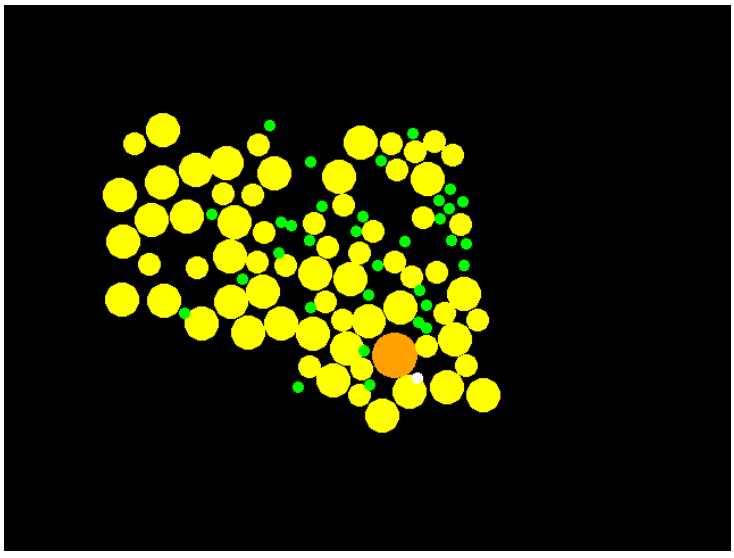

Figure 8.3: Entities Converging to (200, 200).

We already showed a complete example of a movement System in Listing 6.8, but we can define a wide variety of Systems in Lisp. The `Pos2RelPos` System in Listing 8.6 allows us to create Entities with positions relative to other Entities. For example, the small white Entity is at the same position relative to the large orange Entity (named `Sun`) in every example screenshot in this section. This is achieved using the `Parent` Component, which holds a reference to the parent Entity, and the `RelPos` Component, which stores the Position of an Entity relative to its `Parent`. This System sets the value of the Entity's normal `Pos` Component, so it will be displayed at the correct position by the drawing System.

Of note is that the movement System in Listing 6.8 actually uses `RelPos`: with the Query `(and (or RelPos Pos) Vel)`, it will bind `RelPos` if an Entity has it, or `Pos` if not. This means a child Entity's `Vel` Component is also considered relative to the parent Entity.

Listing 8.6: Scene Hierarchy in Lisp.

```
(defun hierarchy-pos (e)
    (if (and (ecs-has e Parent) (ecs-has e RelPos))
        (let ((pos (ecs-get e RelPos)))
          (v2-add pos (hierarchy-pos (ecs-get e Parent))))
      (ecs-get e Pos)))
(ecs-new-system
 (Physics (name 'Pos2RelPos))
 (and Pos (has RelPos))
 (abs)
 (copy-v2 abs (hierarchy-pos entity)))
(ecs-add* (ecs-new) ; Child Entity
          (Parent = (ecs-resolve 'Sun)) ; Big orange Entity
          Pos
          (RelPos 20.0 20.0)
          (Radius 5.0))
```

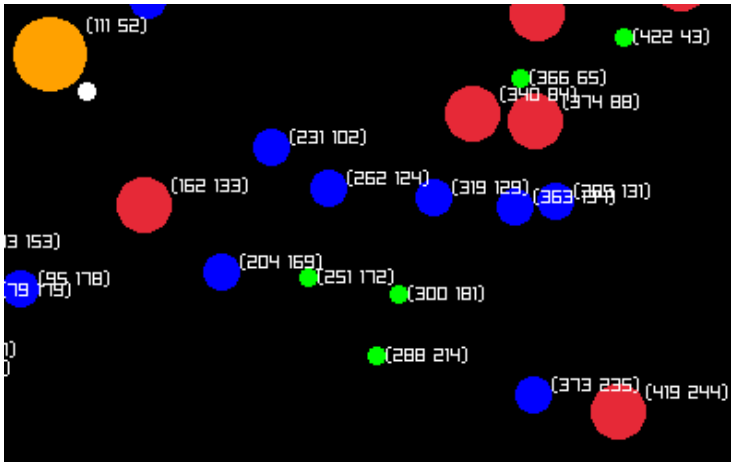

Figure 8.4: Entity Position Labels.

Listing 8.7: Display the `Pos` of each Entity (see Figure 8.4).

```
(ecs-new-system
 (Graphics) (and Pos Radius) (pos radius)
 (draw-text (to-string (list (floor (v2-x pos))
                             (floor (v2-y pos))))
            (+ (v2-x pos) radius)
            (- (v2-y pos) radius) 8))
```

Listing 8.8: Display an Entity's ID when the user clicks it (see Figure 8.5).

```
(ecs-new-system
 (Graphics) (and Pos Radius) (pos radius)
 (when (and (is-mouse-down 'left)
            (point-in-circle
             pos radius
             (make-v2 (float (get-mouse-x))
                      (float (get-mouse-y)))))
   (draw-text (concat "Clicked: "
                      (to-string (ecs-id entity)))
              10.0 10.0 20)))
```

8.2 Possible Use-Cases

Considering the examples in section 8.1, we have identified some potential use-cases for ECSQL, or a system like it, in the context of a small team

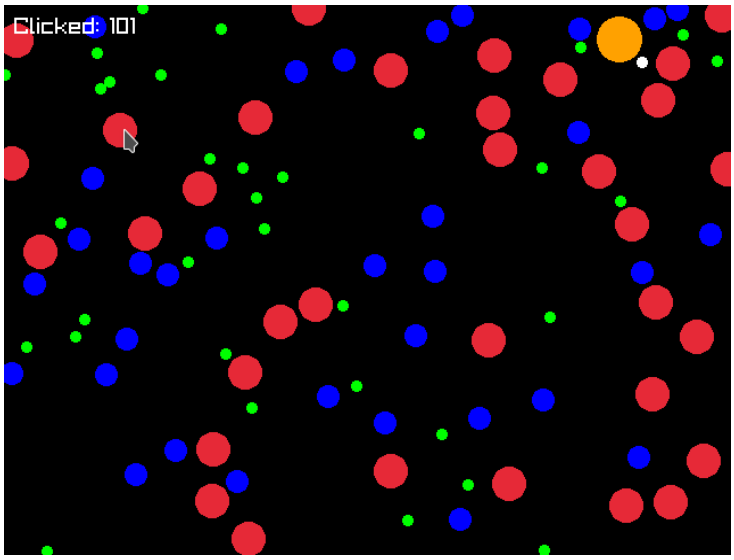

Figure 8.5: Displaying the ID of the clicked Entity.

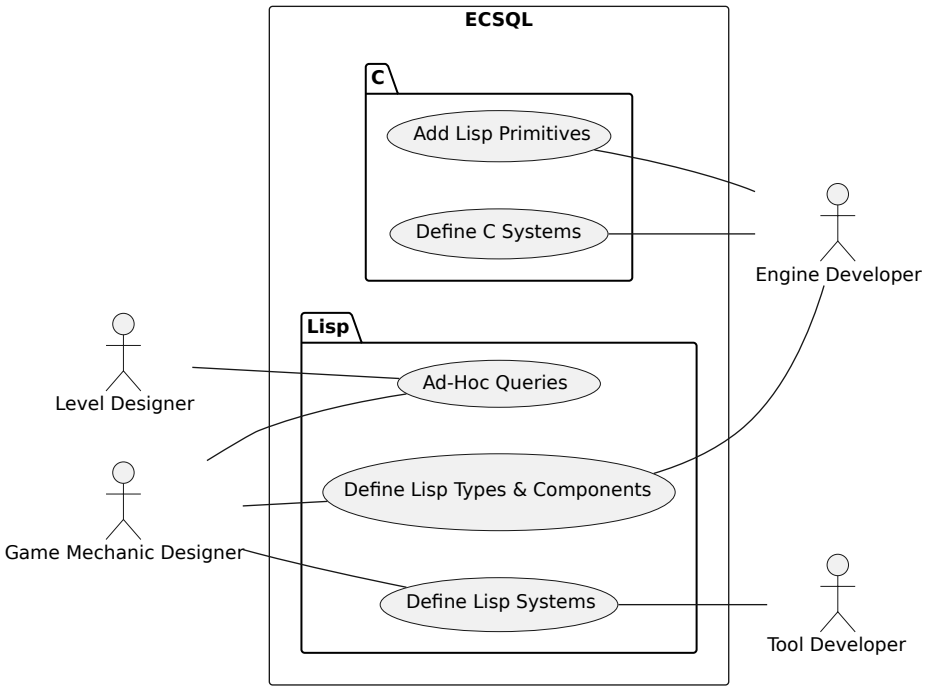

Figure 8.6: Potential Use-Cases of ECSQL in a Game Studio.

developing a game (see Figure 8.6).

Engine Developer These are the only users who should have to write native code; this would most likely be to write Systems that need to run efficiently, or to add primitive functions to the Lisp environment.

Level Designer Less technically-oriented team members like level designers probably wouldn't want to use a textual language like ECSQL. However, it is at least more approachable than C, and they could still benefit indirectly from tools implemented using ECSQL.

Tool Developer As we demonstrated in section 8.1, ECSQL allows us to implement interactive development tools concisely. The lower performance compared to native code would be inconsequential in this case, since it would only have to run on the developers' workstations.

Game Mechanic Designer These users could use Lisp Systems to experiment with the design of a game mechanic; they could write an implementation, test it, then re-write it, and see the change without having to restart the game. The high expressive power of Lisp makes it especially suitable for this style of iterative design, because the developer can implement changes quickly.

8.3 Requirements Evaluation

We have evaluated the extent to which our project has met the requirements set out in section 3.3 in Table 8.1.

Table 8.1: Requirements Evaluation

Requirement	Met	Relevant	Comment
1a (M)	✓	✓	Entities have unique IDs.
1b (M)	✓	✓	Lisp & C structs, or single values.
1c (M)	✓	✓	Systems are executed based on which of a set of Phase Components they have (see subsection 2.1.1). These Components can be manipulated like any other at run time.
1d (C)	✓	✓	Entities are stored in Archetypes.
1e (S)	×	✓	While it is possible to store Relationships, and add them as Components, advanced Relationship functionality (such as Joins) seen in Flecs [27] was not implemented.
1f (S)	×	✓	No consideration was given to making Entity store accesses thread safe.
2a (M)	✓	✓	It is possible to create and delete Entities within Lisp.
2b (M)	✓	✓	Adding Components is made concise with the `ecs-add*` macro.
2c (M)	✓	✓	One-off Queries and Systems can display the values of Components in the terminal or game window, and manipulate them programmatically.
2d (M)	✓	✓	Filtering by Component values is easily achieved by wrapping the code of a transformation in an `if` block.

Requirement	Met	Relevant	Comment
2e (M)	✓	✓	Systems are Entities so they can be listed with Queries. They can be started and stopped by adding and removing Phase Components.
2f (S)	×	×	Native compilation was not implemented.
2g (C)	×	×	Bytecode compilation was not implemented.
2h (S)	✓	✓	It is trivial to define Systems and functions (*at run-time*) within the Lisp environment.
2i (S)	✓	✓	See REPL implementation.
2j (S)	✓	✓	Due to Lisp being a functional language with macros, the code can be highly terse and ergonomic (at least for those familiar with Lisp).
2k (S)	✓	✓	There are simple APIs for defining new Lisp primitives and Systems in C.
2l (S)	×	✓	The quality of error messages varies, and they do not show where an error occurred.
2m (S)	✓	✓	We can monitor the result of a Query by writing a System in Lisp that displays its results (see Requirement 2c).
2n (S)	×	×	The language has a dynamic type system, with no explicit provision for compile-time type checking.

Requirement	Met	Relevant	Comment
2o (C)	×	×	"Compose systems together to build more complex ones" was not a sufficiently concrete Requirement for an implementation to be designed. Common behaviour between Systems can simply be extracted into functions.
2p (C)	×	✓	Systems are executed entirely serially.
2q (C)	×	×	There is no ECSQL compiler.
3 (S)	×	✓	There is no formal test suite.
4 (S)	✓	✓	Though the bouncing balls demo could hardly be called a game, it served its purpose of demonstrating the capabilities of ECSQL in the presentation.
5 (C)	×	×	There is no ECSQL compiler, so there would have been no benefit to embedding ECSQL code into native binaries.
6 (C)	×	✓	There is no profiler.
7 (C)	×	✓	We can print the values of many Component types, so full game state serialisation wouldn't be that much more work to implement.
8 (C)	×	✓	Sending/receiving ECSQL commands over the network was not implemented.

We met all of our **MUST** requirements, and six of our **SHOULD** requirements. Of our unmet requirements, five were written with the assumption

that we would implement a compiler for a statically-typed language (as opposed to an interpreter for a dynamically-typed one), and others were either too vague to implement, or ended up not being relevant, given the direction of the project took. As a result, we would not necessarily consider it a failure that we did not meet them. These results are summarised in the table below.

Rating	Must	Should	Could
Total	8	12	9
Relevant	8	10	5
Completed	8	6	1

In addition to the features specified in the requirements, we added the following features to improve the final product:

Components and Systems as Entities This approach composed well with the required Entity operators to make the whole system more flexible.

Documentation Strings Our Lisp implementation supports documentation strings for functions, macros, variables, structs, and Components. Where applicable, these are automatically generated with macros.

Lisp Standard Library We created a small standard library of Lisp functions and macros, based on the ones in Common Lisp. Some notable ones are `reduce` (fold), `case` (switch) and `gensym` (generates a *new* symbol).

On the whole, we consider the project to be a success.

Chapter 9

Conclusions

We have achieved our initial goal of creating a general-purpose, interactive game development tool. ECSQL allows developers to create and manipulate Entities as they see fit, incrementally develop new features for their games, and even implement more high-level, specialised tools at an otherwise infeasibly fast pace. Since all the state of a game is in a standardised format (represented with Entities and Components), developers can apply these powerful operations to all aspects of their games. The high-level, concise syntax of Lisp allows them to access all this power easily, and the REPL lets them do all this without even restarting the game.

This system is ECS-based, so it comes with the same limitations as most ECS frameworks face. For one thing, it would have limited applicability outside games. It also lacks the flexibility of a full-on programming paradigm, such as OOP [28]. On the other hand, within the narrow field of game development, as we have shown, it is tremendously useful.

Most popular Entity Component System libraries and Lisp implementations have been heavily optimised. By comparison, our implementations are highly naive, with little or no attempt at optimisation, beyond the basic

architecture of the Entity store and Lisp object representation. Their performance would almost certainly be orders of magnitude slower. On the other hand, ECSQL fills such a different niche from these existing systems that the use-cases where we could draw a direct performance comparison would not make use of its unique features. Due to this, we did not consider it worthwhile to analyse its performance in comparison to existing solutions[1].

Though we were too busy in term 1 to spend much time on the project, we still made the mistake of spending too much time on research. If we had focused on collecting just the necessary information to start developing, rather than exploring all potential applications and capabilities of the system, we might have had time to implement some more extensions, or perhaps even a basic Lisp compiler.

9.1 Further Work

Based on our unmet requirements, and analysis of the final product, we have identified some areas where further work would improve the system.

9.1.1 Lisp Implementation

Our Lisp implementation has obvious flaws. To make this system usable for even semi-serious applications, we would have to remedy them.

Garbage Collection We currently make no attempt to reclaim unused Lisp memory. Our first, most important step to improve the system would be to implement a garbage collector. We researched the simplest ap-

[1]To put it another way, we are not aware of any existing solutions to the same set of problems that ECSQL solves.

proaches, and would have attempted to implement a mark-sweep collector, if any [18, Ch. 2].

Performance We could have implemented a compiler, targeting either a Bytecode VM or native machine code, to improve performance.

Standard Conformance A "commercial" implementation of a system like could benefit from conforming to a standard, such as Common Lisp [13], so users could benefit from existing libraries.

On the other hand, the ways our language differs from Common Lisp are generally in service of this specific application: for example, the built-in Entity type and static member types for structs make interoperating with C "engine" code much easier.

There are also game engines like Godot [20], with custom scripting languages that integrate with their specific functionality, so this situation isn't unprecedented. We also could have used an existing embedded Lisp interpreter, and an existing ECS, but then they could not have integrated together as well.

9.1.2 Entity Relations

We found the Entity Relations feature in Flecs [29, Relationships], and its potential applications Mertens [27], compelling, so we attempted to implement this feature into ECSQL. However, we only implemented a small subset of the features required for them to be useful, and they would not have been a novel contribution, so we chose not to make further progress towards supporting them.

Relationships provide an elegant way to represent some important concepts in games, such as hierarchies, and would have provided a powerful way

of writing Queries that manipulate whole sets of related Entities at once. As such, this feature would have greatly expanded the capabilities of ECSQL, so we would have liked to implement it if we had more time.

9.2 Self-Assessment

We have implemented an Entity Component System library with an Archetype-based Component store, an interpreted Lisp system with lexical scope and macros, and an ECS-based Query language that integrates the ECS library and Lisp together. We created a small "game" using the system that allowed us to illustrate its capabilities in an application.

Limitations, such as the unoptimised Lisp interpreter and lack of garbage collection, make the final product unsuitable as a basis even for moderately complex games. Despite these, it still effectively demonstrates a novel set of capabilities that would make it worth refining into a more complete system. Even in its current state, the product could be of use to a developer that wanted to create a simple game, either as a prototype to test new mechanics, or in a small time window like a game jam.

Bibliography

[1] Mike Acton. 'Data Oriented Design and C++'. In: *CppCon 2014*. 11th Sept. 2014. URL: https://youtu.be/rX0ItVEVjHc?si=mMuHCnULl3e2TSyx (visited on 08/04/2024).

[2] Carter Anderson. *Bevy Engine*. 10th Aug. 2020. URL: https://bevyengine.org (visited on 19/11/2023).

[3] Atlassian. *Agile Project Management*. 2024. URL: https://www.atlassian.com/agile/project-management (visited on 08/04/2024).

[4] Attractive Chaos. *Klib: a Generic Library in C*. 2008. URL: http://attractivechaos.github.io/klib (visited on 11/04/2024).

[5] Sean Barrett. 'Immediate Mode GUIs'. In: *Game Developer* (Sept. 2005), pp. 34–36. URL: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/GD_Mag_Archives/GDM_September_2005.pdf (visited on 29/04/2024).

[6] Jessica D. Bayliss. 'The Data-Oriented Design Process for Game Development'. In: *Computer* 55.5 (May 2022), pp. 31–38. ISSN: 0018-9162. DOI: 10.1109/MC.2022.3155108.

[7] Carsten Dominik and Bastien Guerry. *Org Mode*. 2003. URL: https://orgmode.org/ (visited on 08/04/2024).

[8] Robert A. van Engelen. 'Lisp in 99 lines of C and how to write one yourself'. 19th July 2023. URL: https://raw.githubusercontent.com/Robert-van-Engelen/tinylisp/main/tinylisp.pdf (visited on 26/11/2023).

[9] Epic Games. *Unreal Engine*. 1998. URL: https://www.unrealengine.com/en-US/unreal-engine-5 (visited on 10/10/2023).

[10] Free Software Foundation. *GNU Emacs Lisp Reference Manual*. Version 29.2. 2024. URL: https://www.gnu.org/software/emacs/manual/html_node/elisp/index.html (visited on 10/04/2024).

[11] Steven Ganz et al. *Revised^7 Report on the Algorithmic Language Scheme*. report. R^{7}RS Authors, 13th Feb. 2021. URL: https://standards.scheme.org/official/r7rs.pdf (visited on 10/04/2024).

[12] Duncan Geere. *How To to Use Commands in Minecraft*. 22nd Sept. 2023. URL: https://www.minecraft.net/en-us/article/minecraft-commands (visited on 29/04/2024).

[13] Paul Graham. *ANSI Common Lisp*. English. London; Englewood Cliffs, N.J; Prentice Hall, 1996. ISBN: 0133708756.

[14] Paul Graham. *On Lisp. Advanced Techniques for Common Lisp*. Prentice Hall, 1993. ISBN: 0130305529. URL: http://paulgraham.com/onlisp.html (visited on 14/11/2023).

[15] Philip Greenspun. 'Philip Greenspun's Research'. 2017. URL: https://philip.greenspun.com/research/ (visited on 06/04/2024).

[16] Doug Hoyte. *Let Over Lambda. 50 years of Lisp*. Lulu.com, 2nd Apr. 2008. ISBN: 1435712757. URL: https://letoverlambda.com/ (visited on 14/11/2023).

[17] Roberto Ierusalimschy and Waldemar Celes Luiz Henrique de Figueiredo. *Lua*. 1993. URL: https://www.lua.org/ (visited on 29/04/2024).

[18] Richard Jones, Antony Hosking and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. English. Boca Raton, FL: CRC Press, 2012. ISBN: 1420082795.

[19] Andrew Kravchuk. *Gamedev in Lisp. Part 1: ECS and Metalinguistic Abstraction*. 17th Oct. 2023. URL: https://awkravchuk.itch.io/cl-fast-ecs/devlog/622054/gamedev-in-lisp-part-1-ecs-and-metalinguistic-abstraction (visited on 07/04/2024).

[20] Juan Linietsky and Ariel Manzur. *Godot Engine*. 14th Jan. 2014.

[21] Adam Martin. *Data Structures for Entity Systems: Contiguous memory*. (The images have been broken since late 2022.) 8th Mar. 2014. URL: https://t-machine.org/index.php/2014/03/08/data-structures-for-entity-systems-contiguous-memory/ (visited on 19/09/2022).

[22] Adam Martin. *Entity Systems are the future of MMOG development – Part 2*. 11th Nov. 2007. URL: https://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/ (visited on 08/10/2023).

[23] Adam Martin. *Entity Systems are the future of MMOG development – Part 3*. 22nd Dec. 2007. URL: https://t-machine.org/index.php/2007/12/22/entity-systems-are-the-future-of-mmog-development-part-3/ (visited on 08/10/2023).

[24] Sander Mertens. *A Roadmap to Entity Relationships*. 14th July 2023. URL: https://ajmmertens.medium.com/a-roadmap-to-entity-relationships-5b1d11ebb4eb (visited on 19/11/2023).

[25] Sander Mertens. *Building an ECS #1: Where are my Entities and Components*. 6th Aug. 2022. URL: https://ajmmertens.medium.com/building-an-ecs-1-where-are-my-entities-and-components-63d07c7da742 (visited on 09/04/2024).

[26] Sander Mertens. *Building an ECS #2: Archetypes and Vectorization*. 14th Mar. 2020. URL: https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9 (visited on 09/04/2024).

[27] Sander Mertens. *Building Games in ECS with Entity Relationships*. 7th Apr. 2022. URL: https://ajmmertens.medium.com/building-games-in-ecs-with-entity-relationships-657275ba2c6c (visited on 19/11/2023).

[28] Sander Mertens. *ECS: From Tool to Paradigm*. 19th Feb. 2021. URL: https://ajmmertens.medium.com/ecs-from-tool-to-paradigm-350587cdf216 (visited on 21/11/2023).

[29] Sander Mertens. *Flecs: Fast Lightweight ECS*. Version 3.2. 2018. URL: https://www.flecs.dev/flecs (visited on 13/11/2023).

[30] Sander Mertens. *Making the most of ECS identifiers*. 22nd July 2020. URL: https://ajmmertens.medium.com/doing-a-lot-with-a-little-ecs-identifiers-25a72bd2647 (visited on 18/11/2023).

[31] Sander Mertens. *Why it is time to start thinking of games as databases*. 6th June 2023. URL: https://ajmmertens.medium.com/why-it-is-time-to-start-thinking-of-games-as-databases-e7971da33ac3 (visited on 19/11/2023).

[32] Austin Morlan. *A Simple Entity Component System*. 25th June 2019. URL: https://austinmorlan.com/posts/entity_component_system/ (visited on 11/04/2024).

[33] Peter Norvig. 'Design patterns in dynamic programming'. In: *Object World* 96.5 (1996).

[34] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN: 0990582906. URL: http://gameprogrammingpatterns.com (visited on 11/10/2023).

[35] Christian Queinnec. *Lisp in Small Pieces*. English. 1st paperback. New York; Cambridge, U.K; Cambridge University Press, 2003. ISBN: 0521545668.

[36] Thibault Raffaillac and Stéphane Huot. 'Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model'. In: *Proc. ACM Hum.-Comput. Interact.* 3.EICS (June 2019). DOI: 10.1145/3331150.

[37] Raysan. *Raylib*. Nov. 2013. URL: https://www.raylib.com/ (visited on 11/04/2024).

[38] Tcl Community. *Tcl Developer Xchange*. 7th May 2022. URL: https://www.tcl.tk/ (visited on 29/04/2024).

[39] Unity Technologies. *Entities Package Manual*. Version 1.1. Unity Technologies. 13th Sept. 2023. URL: https://docs.unity3d.com/Packages/com.unity.entities@1.1/manual/index.html (visited on 08/10/2023).

[40] Unity Technologies. *Unity Data-Oriented Technology Stack (DOTS)*. 2022. URL: https://unity.com/dots (visited on 19/11/2023).

[41] Unity Technologies. *Unity Engine*. 8th July 2005. URL: https://unity.com/products/unity-engine (visited on 10/10/2023).

[42] Valve Developer Community. *Developer Console*. 22nd July 2023. URL: https://developer.valvesoftware.com/wiki/Developer_console (visited on 29/04/2024).

Appendix A

Lisp Primitives

Our Lisp implementation includes a large number of primitives, implemented in `src/lisp/primitives.c`. There are three main categories of primitives:

Standard Library These functions are integral parts of the language; they implement core functionality like vector and string construction. E.g. `eq`, `/`.

Integration These act as wrappers around functions from external libraries. E.g. `fopen`, `draw-text`.

Internal Functions that implement necessary functionality, but are not expected to be called by the user. E.g. `defname`, `--struct-allocate`.

We have listed most of our "Standard Library" and "Integration" primitives in Table A.1. These are the ones that language users are expected to use, so we have written Lisp documentation strings for them.

Table A.1: User-Facing Lisp Primitives

`primitive (arguments)`	Behaviour
`* (numbers...)`	Multiply a list of numbers.
`+ (numbers...)`	Add a list of numbers.
`/ (number divisors...)`	Divide the first argument by each of the remaining arguments. With one argument, divide 1 by it.
`- (numbers...)`	Subtract from the first argument all remaining arguments. With one argument, negate it.
`cons (car cdr)`	Create a new pair, with `car` and `cdr` as its components.
`car (list)`	Obtain the `car` of `list`, or nil if `list` is nil.
`cdr (list)`	Obtain the `cdr` of `list`, or nil if `list` is nil.
`quit ()`	Quit Lisp.
`symbol-name (symbol)`	Get the name of the supplied symbol as a string.
`intern (string)`	Obtain the canonical symbol with the given name name.
`make-symbol (string)`	Produce a new, uninterned symbol with the given name.
`make-string (n c)`	Produce a string length N, with every character being C.

`primitive (arguments)`	Behaviour
`make-vector (n v)`	Produce a vector length N, with every element being V.
`vector`	Produce a vector containing the arguments.
`aref (vector n)`	Get the nth element of the vector.
`aset (vector n value)`	Set the nth element of the vector to the supplied value.
`eq (a b)`	Return t iff the arguments are bit-for-bit the same.
`eql (a b)`	Return t iff the arguments are equal, handling numbers and strings specially.
`assoc (key list)`	Returns first key-value pair in `list` whose car is eq to `key`, if any, else nil.
`length (object)`	Returns the length of the given list, vector or string.
`to-string (form)`	Returns the printed representation of the argument as a string.
`type-of (object)`	Returns the symbol representing the type of the argument.
`type-tag (object)`	Returns the Object type tag of the argument (a number). All structs have the same type tag (`OBJ_STRUCT_TAG`).
`funcall (fn args...)`	Apply the first argument (function) to the remaining arguments.

`primitive (arguments)`	Behaviour
`apply (fn args... arglist)`	Apply the first argument to the remaining arguments. The last argument is a list of arguments to pass to the function.
`eval (form)`	Evaluate the argument form.
`read-stream (file)`	Read one Lisp object from the supplied `file`.
`fopen (file r/w)`	Open a file with the given r/w setting.
`getc (file)`	Read a single character from an open file.
`macroexpand-1 (form)`	Expand the top-level macro in the argument form, if there is one.
`macroexpand (form)`	Recursively expand out all macros in the argument form.
`wrong (message arg)`	Signal an error, displaying a message and the value of the second argument.
`size-of (type)`	Return the number of Bytes necessary to store elements of the argument type in a struct.
`type-spec-matches (form spec)`	Returns t iff the supplied form matches the supplied type spec.
`structp (object)`	Returns t iff the argument is a struct.
`struct-metadata (type)`	Returns reflection data about the given struct type.
`ecs-new ()`	Create and return a new ECS entity.

`primitive (arguments)`	Behaviour
`make-entity (id generation)`	Produce an Entity object with the given id and generation. Not guaranteed to be a live entity.
`ecs-pair (relation target)`	Produce a Relation object with the given `relation` and `target`. Not guaranteed to be a valid Relation wherein `relation` and `target` are both alive.
`make-relation (rel t)`	Produce a Relation object with the given Relation `r` and target Entity `t`. Not guaranteed to be a valid Relation wherein `relation` and `target` are both alive.
`ecs-entity (id)`	Returns the Entity with the argument ID, if alive. Otherwise, nil.
`ecs-destroy (entity)`	Destroy the supplied Entity.
`ecs-get (entity component)`	Obtain the value of `component` for `entity`. The `entity` must have `component`, and `component` must have LispStorage, or an error is raised.
`ecs-set (e c v)`	Set the value (`v`) of Component `c` for Entity `e`. The `entity` must already have `component`, and `component` must have LispStorage. The `value` must be of `component`'s LispStorage type (see ecs-storage-type).

`primitive (arguments)`	Behaviour
`ecs-set-name (entity name)`	Set the name of `entity`. Names are not Components, and are used to find Entities with ecs-lookup.
`ecs-lookup (name)`	Obtain the Entity with the given name, if it exists.
`ecs-has (entity component)`	Returns t iff `entity` has `component`.
`ecs-add (entity component)`	Add `component` to `entity`.
`ecs-id (entity)`	Returns the `id` of `entity`.
`ecs-gen (entity)`	Returns the generation of `entity`.
`ecs-relation (relation)`	Returns the Relation type of `relation`
`ecs-target (relation)`	Returns the target Entity of `relation`
`ecs-new-component`	Creates a new Component that stores values of type `type`.
`ecs-do-query (function query)`	Run `function` on every Entity matching `query`. This is the backend to ecsql, which you should probably use instead.
`ecs-register-system (f q)`	Create a new System (Entity) with the given System function `f` and Query `q` added. This is the backend to ecs-new-system, which you should probably use instead.
`ecs-storage-type (component)`	Obtain the type of Lisp Object stored by `component`. If `component` is a Relation, this will be the same as (ecs-storage-type (ecs-relation `component`)).

`primitive (arguments)`	Behaviour
`get-mouse-x ()`	Get the X coordinate of the mouse cursor on the game window.
`get-mouse-y ()`	Get the Y coordinate of the mouse cursor on the game window.
`get-screen-width ()`	Get the width of the game window in pixels.
`get-screen-height ()`	Get the height of the game window in pixels.
`get-delta ()`	Get the duration of the last frame as a floating-point number.
`draw-text (text x y size)`	Draw `text` at the given X and Y coordinates, at the given `size`.
`is-mouse-down (button)`	Returns t iff mouse button `button` is currently held down. Allowed values of `button`: `left`, `right` (symbols).
`is-mouse-pressed (button)`	Returns t iff mouse button `button` was just pressed. Allowed values of `button`: `left`, `right` (symbols).

We have documented a few of our "Internal" primitives in Table A.2.

Table A.2: Internal Lisp Primitives

`primitive (arguments)`	Behaviour
`defname (ns name value)`	Add a mapping from `name` to `value` in namespace `ns`, which must be one of `globals`, `functions`, `macros` or `structs`.
`ecs-do-query (q f)`	Run function `f` on each Entity matched by Query `q` (which is represented in predicate+bindings form).
`--struct-register (name)`	Generate a new struct type ID (see subsection 6.4.1), and create a mapping from that ID to the supplied name (a symbol).
`--struct-allocate (id cells)`	Allocate the given number of memory cells (64 bits/cell) to store a struct object with the supplied ID.

In addition to the struct registration and allocation primitives, we have getter and setter primitives for accessing the contents of a struct as raw data, through a pointer, or as a boxed object. These are named as shown below.

`--struct-{get,set}-{vec,val,object}`

These functions are deliberately type-unsafe, effectively just providing thin wrappers around `memcpy`. We took this approach because the necessary type information required to perform struct operations is only created when Lisp structs are defined, at run-time. The macros in `lisp/struct.lisp` determine the correct parameters to pass to these functions automatically. Not even *I* am supposed to use them directly.