Asterism: Operational Logics as a Game Engine Engine

Joseph C. Osborn, Cynthia Li, Katiana Wieser
Computer Science Department
Pomona College
joseph.osborn@pomona.edu

Abstract

Game development is challenging even for experienced programmers, and game engine programming carries the added difficulty of creating a flexible, generic API with suitable performance. Part of this difficulty is that both game programs and game engines act in many ways like programming languages and their standard libraries, with the final game being built in terms provided by the platform on which it is built.

In this work, we synthesize perspectives from platform studies and operational logics to devise Asterism, a *game engines engine*: types, abstractions, and candidate implementations not only for features common across game engines, but also for the "connective tissue" between disparate game systems. Game engines defined in Asterism function analogously to domain-specific languages in which individual games are coded.

1 Motivation

Programming videogames from scratch can be challenging even for experienced programmers. The extra effort needed to build game content editing and auditing tools like animation and level editors, AI players, and automated testing tools compounds this difficulty. Many game developers therefore turn to fully-featured engines like Unity or Unreal, which have mature user interfaces and marketplaces full of additional tools—or else designers use specialized game-making tools like Bitsy, PuzzleScript, or Twine.

It is very difficult to make a new game engine for all the same reasons it is difficult to make games, multiplied by the generality that engines are expected to afford. This is because, like games, game engines are *sui generis* software products—unique unto themselves. It is difficult to share all but the lowest-level features (e.g., loading assets from disk, abstracting over controller inputs, or compiling shader programs) from engine to engine, because there is no standardized, shared theoretical foundation on top of which engines are built. This is not to say that game engines don't sometimes share certain characteristics or architectural decisions (**?**), but they each invent these notions for themselves. A notable exception in this area is *physics engines*, any given example of which might present a well-defined interface that can be plugged into a variety of game engines.

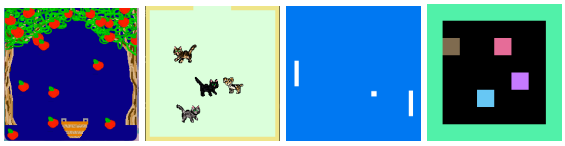

Figure 1: *Apple Catching*, *Clowder*, and *Paddles* (three games made with the same composition of OLs), plus *Extreme Dungeon Crawler*, a simple `boxsy` game.

Taking Unity and Unreal as examples, Unity settles on an architecture where GameObjects carry sets of Components which are updated each frame, while Unreal's similar Actor and Component architecture has substantially different affordances due to the different semantics of subclassing and instancing in Unreal. The metaphors are in some sense compatible, but only by taking them to a very high level of abstraction. They function more like design patterns and less like an ontology for games and game engines.

In this work, we investigate the problem of building a *game engine engine*, noting that the design of a game engine and a game on top of it is a project in successively more specific domain-specific languages (DSLs): from a general purpose programming language to a set of engine-specific data structures, protocols, and other commitments down to the level of a particular game program in terms of which game rules and content are expressed. Each successive layer imposes language-like restrictions and affordances on the layers above. We showcase a few Asterism games in Figure 1.

As our theoretical contribution, Asterism builds on the game studies formalism of *operational logics* and their compositions, with individual engines defined as compositions of such logics (called OLs henceforth to avoid ambiguity around the term *logic*). OLs are not mathematical logics, but ways of characterizing how a player comes to mentally model observations of phenomena due to a system with respect to the inputs they are providing to a system. In other words, an OL is the linkage of an hypothesized process with observed events supporting a particular gameplay experience. Importantly for this work, a particular set of OLs have been labeled as foundational and these can be identified with mathematical logics and formalisms (**?**), and these are the

ones around which Asterism is designed.

Each composition, or engine, is something like a new, game-genre-specialized programming language: it defines some base terms and allows for their combination and elaboration into game rules and data. We also show an implementation of Asterism in the Rust programming language (with at least one implementation each of collision, control, entity-state, linking, physics, and resource logics), two distinct engines defined in Asterism, and one game in each of those engines. Our code is available under the Non-Violent Public License (NVPL+) and is made available on GitHub[1] or by correspondence with the authors.

2 Related Work

Game programs are characterized by tight loops over sets of interesting objects performing similar computations. Examples include iterating through all colliders to check overlaps, iterating through all enemies updating their AI behaviors, or enumerating and updating simulated objects according to a spatial partitioning scheme or a schedule of objects that need updating.

The code responsible for organizing each type of data and performing the computation is often called a *system*: a collision system, AI system, inventory system, dialogue system, and so on. Often, some state machine governs handoffs between different systems, and each system itself has some state to indicate what sort of processing it should perform on a given frame. This kind of structure is seen to separate concerns in a way which is relevant to game programs, and it has been reified in so-called entity-component systems under the umbrella project of data-oriented design (**?**). This approach dovetails well with OLs, the formalism underwriting Asterism.

A complete game design must include not only its rules and instantial assets like graphics and level configurations, but also the terms in which those rules are defined—in the same way that a high-level game engine gives an ontology in which specific games are implemented. The Gemini project is one recent approach to specifying a game design in (formal) logical terms (**?**). Like the earlier BIPED and Ludocore systems (**?**), it uses Answer Set Prolog as a specification language. Also like BIPED, Gemini games can be judged against various criteria or transformed into game programs that humans can play. It is also typical of game-making tools (e.g., PuzzleScript, Bitsy, Flickgame), as distinct from engines, to define explicit textual or UI-driven "languages" for defining games.

The main obstacle to generality for Gemini—and a key reason for extensive duplication in its proceduralist reading rules—is that it enforces the choice of a particular set of operational logics in a particular arrangement, but it does not treat these logics or their fundamental operations as first-class objects. Specific integrations of logics are considered as units, which leads to awkward situations like separately defining feedback loop detection for resource growth, the amount of a particular color drawn on the playfield, and per-entity health. This issue is not unique to Gemini by any means, so the search for more orthogonal sets of primitives is a key motivation of the present work.

Not only these systems, but also the VideoGame Description Language (**?**) commit to particular compositions of OLs, i.e., a particular definition of what games are made of. Having a common ontological framework for diverse classes of games is extremely valuable, and this work can be seen as extending that effort into the space of procedural programming languages and simultaneously allowing for transfer across different game *schema*—not only arcade-style games, but also role-playing games, sports games, puzzle games, and so on.

3 Modeling Game Engines

Since their initial development and further exposition (**???**), *operational logics* (OLs) have enjoyed broad use and inspired several approaches to game studies. An OL (e.g., collision logic) is a combination of an *abstract process* (overlap detection) with its *communicative roles* (objects in space) in a game, connected through an ongoing *game state presentation* (sprites matching hit-boxes) and supporting a *gameplay experience* (stuff can happen when things touch) (**?**). Besides their direct application in describing specific games (**?**), OLs underlie several approaches to understanding how games communicate ideas (**??**) and a variety of projects in player and game modeling and game generation (**??????**).

OLs let us view games not as *bags of mechanics* but as assemblages of abstract *operations* from diverse *logics*. So far this has mainly been done on a case-by-case basis: OLs have been used to describe individual games or certain broad classes of games (e.g., *graphical logic games* comprising collision, physics, resource, and control logics).

In this work, we start from the core idea of *platform studies*—that platforms make certain ontological commitments regarding the games that best *fit* them, whether due to technical or social reasons—which applies equally to game consoles like the Atari 2600 (**?**) and more fully-featured software frameworks like Flash (**?**) or Unity. Instead of focusing on particular features of these (hardware or software) game engines, we want to draw attention to the *operational logics* they reify: in the case of the Atari 2600, that takes the form of hardware support for moving sprites, collision tests, controller mappings, and resource counters; for an engine like Unity, we see collision, physics, resource, control, camera, and persistence logics made primary. Notably, many popular game making tools (to varying extents, e.g. Game Maker, Bitsy, Unity's or Godot's standard library) privilege the arcade-game style of so-called *graphical logics*: characters moving about in simulated continuous spaces with resources (a composition of collision, control, entity-state, linking, physics, and resource logics).

Composing Operational Logics

OLs compose together into games or game platforms in three main ways: *structural syntheses*, joint *operational integrations*, and shared *communication channels* (**?**). This innate compositionality is a key reason for the effectiveness of OLs in the present work.

[1] https://github.com/faim-lab/asterism

The most fundamental connections between OLs are when they jointly produce a game's ontology of concepts: player characters, enemies, projectiles, inventories, equipment, rooms. *Structural syntheses* map game notions between distinct OLs: When we have a concept of a game character which interacts with the physics, collision, control, and entity-state logics, that is a structural synthesis at work. A game like *Super Mario Bros.* might be defined in terms of syntheses representing enemies, Mario, interactable blocks, levels, and so on. Several common syntheses exist, but the space of structural syntheses is not bounded and under-explored or new syntheses might represent areas of novelty in game design (since game genres may be thought of as popular, conventional assemblages of operational logics (**?**)).

The term *synthesis* here does not refer to program synthesis, but to the idea that a new concept is composed by unifying some (aspects of) existing concepts. This includes but is not limited to concept co-occurrence: for example, a game's inventory may have consist of items laid out spatially in "bags", while the crafting mechanics work only in terms of the number of input and output items. Both the numerical view and the spatial view are the inventory from the point of view of their respective systems.

Structural syntheses can be seen as a set of logical relations between the terms of different OLs: e.g., the colliders and positions of a collision logic, the physics state and bodies of a physics logic, and the resource pools and quantities of a resource logic.

Once OLs are structured together, game mechanics are built by combining their operations. For example, "lose ten health when you touch a wall" or "regain full health when you touch a powerup" share a similar template: "When you touch something, something else might happen." These are *operational integrations* of collision and resource logics. Addressing the infinite space of possible mechanics by combining elements from a fixed set of OLs gives useful constraints on automated game analysis, game generation, and game design in general. Not all such integrations have the "when X, do Y" format—in the crafting example from before, an item should only be craftable if there is a space big enough for it to fit in the spatial inventory; crafting uses up some resources and produces new ones; and so on. We view OLs as defining both predicates and actions, and these can be combined arbitrarily across OLs to define operational integrations. Structural syntheses define a kind of grammar in which operational integrations can be established.

OLs can also overlap by sharing *communication channels,* where OLs provide and can make use of communicative affordances. For example, games with distinct characters or sprites commonly share the space around the sprite as a channel for other information.

Game engine engines

Game-making tools like Bitsy that are intentionally constrained to specific genres or types of games can yield highly user-friendly programming environments, bringing elements of play to the experience of software development. They achieve this by fixing particular structural syntheses and sets of possible operational integrations, and therefore constraining the space of authorial interventions. More general tools like GameMaker or Unity also commit to particular syntheses to varying degrees, but give an escape hatch in the form of a general-purpose programming language in which programmers can implement or reimplement aspects of the game however they like.

Game engines have some important commonalities with games: they commit to particular structural syntheses, communication channels, and operational integrations. However, whereas a game is *complete* in the sense that all its definitions, rules, and instantial assets like images, music, and 3D models are given, game engines and game-making tools provide *schemata* which are to be filled in by a game designer. Similarly, a given game could in principle be *modded*, changing the appearance and behavior of the characters in its world, and potentially changing even the world and the scripted events of the game—but the OLs and their integrations would remain the same. Each game therefore has some base "platform"—the particular composition of OLs in terms of which the game proper is defined.

A game engine is therefore a composition of OLs: A set of logics, a set of structural syntheses between and communication channels shared among these logics, and some mechanism for defining operational integrations. An individual game will define data in terms of these syntheses and mechanics as new operational integrations.

It may be helpful to consider the metaphor of Lego bricks: there are combinatorially many different types of building blocks (studs, flats, blocks, poles, and so on, customized by length, width, and color); but they share a common set of interfaces by which they can connect and support each other. This commitment to a particular method of composing block designs limits the range of possible constructs to those which are *sensible* (e.g., there are no unstable connections due to mixing Lego bricks and Mega Bloks). A particular composition of OLs functions similarly: we have carved off a space where certain ideas are easily expressed, and we can offer individual compositions as a kind of kit to game makers while also admitting the creation of entirely new sorts of game engines.

Our goal with Asterism is to make the task of defining new game-making tools as compositional and modular as the task of defining new DSLs—this motivates our project of finding shared engine-level concepts across distinct game engines, so that tools can be written in terms of these concepts rather than being tied to particular game engines. In pursuit of that goal, we will show that game engines can be defined as compositions of OLs, casting the creation of new types of game engine as the composition of OLs, and allowing for greater portability of tools, AI support, and other interventions between game engines.

To sum up, a game engine engine should provide a set of components with which game engines can be built. To take advantage of an existing type checker and compiler toolchain, we show an example of such a game engine engine in the Rust programming language, defining types and computations in the base programming language rather than in a higher-level DSL.

We are particularly eager to draw connections to recent work on games and their abstractions, using more or less abstract versions of particular operational logics to admit modular, abstraction-refinement approaches to game AI and game design support (**?**).

4 Asterism

Asterism is a library written in the Rust programming language. It provides two key concepts: first, a data-oriented query table mechanism for efficiently processing lists; and second, types and interfaces describing OLs. As a convenience, Asterism also provides example implementations of various OLs (collision, control, entity-state, linking, physics, and resource logics) and a declarative sprite animation system. Of these, the main contribution in this work is the set of Rust *traits* (something like type-classes) representing OLs, their associated concepts, and their compositions.

The first main component of Asterism, the *query table* system, acts as a kind of blackboard for sharing data between OLs and processing events to define operational integrations. While Asterism defines the notion of an OL and a mechanism for sharing information across OLs and through time, a game engine defines specific structural syntheses, communication channels, and/or operational integrations while a game *per se* defines mainly operational integrations.

There are two types of query tables in Asterism: *output tables* and *condition tables*. An output table is a collection of values of a particular type, from which a contiguous vector of that type of element can be produced (not unlike, say, an `Iterator`). A condition table is a collection of queries arranged in some data-flow (e.g., one query might be mapping a function over another query, or zipping the results of two other queries together, or filtering another query based on some predicate). Query tables arise from the need for operational integrations to be defined across diverse logics and structural syntheses, and our representation structures that data sharing as data (the graph of query dataflow). A similar motivation underlies blackboard architectures and other data sharing approaches, and ours is inspired by existence-based processing in the data-oriented style (**?**).

Asterism's core trait, and a key contribution of this work towards operationalizing the theory of OLs, is `Logic`, representing an individual OL. Implementors of a `Logic` must define several types (notably, since a `Logic` may itself have type parameters, this also gives engine and even game developers the option of parameterizing the `Logic`):

- `Ident`: The type of objects governed by this OL. Must be cheaply copyable.
- `IdentData`: Data associated with each `Ident` by the OL. Must be cloneable (i.e., copyable).
- `Event`: The events this OL can trigger, or the predicates it defines; implementors of `Event` must define an `EventType` and a way to get the type of a particular event.
- `Reaction`: The abstract actions this OL can be made to perform. Must implement `Reaction`.

Implementors of `Logic` must also define trait methods for handling this logic's predicates and for getting and setting the `IdentData` for a particular `Ident`. Moreover, `Logic` instances must provide a way to iterate through their objects and through `Events` which have just occurred. The traits `EventType` and `Reaction` are *marker traits* that have no behavior—when defining a particular OL, a programmer provides types implementing these marker traits.

`Ident` and `IdentData` allow for the definition of concept co-occurrence structural syntheses and shared communication channels wherever multiple OLs share the same identifier type. More complex structural syntheses may involve the definition of intermediate concepts and more complex `Ident` types that combine several identifiers. `Event` and `Reaction` provide for operational integrations (along with each OL's required implementation of `OutputTable<(Ident,IdentData)>` and `OutputTable<Event>`, which produce the streams of object states and events that feed into operational integrations).

By way of example, a collision logic might be instantiated with numerical identifiers for `Ident` (e.g., "Body 1", "Body 2", and so on), objects' shapes and collision flags as their `IdentData`, produce for its `Event` new contacts between objects, and support triggering a `Reaction` such as changing an object's position or which set of objects it should collide with. To retrieve the current contacts to realize e.g. a teleporter, a game or engine would pose a query joining both the identifiers (and their collision shape data) and the contacts, look up the teleporter's destination using its corresponding `Ident`, and trigger a collision `Reaction` that moves the other contact to its new location. To synthesize this collision logic with a physics logic for continuous movement in space, we might instantiate a physics logic with the same `Ident` type. This move is not unique to Asterism—note that data-oriented entity-component systems also use identifiers as opaque handles to index into specialized regions of contiguous memory storage.

One useful consequence of defining `Logic` as a trait with so many parameters (of which several are *marker traits*) is that a single OL (e.g., a linking logic) can be instantiated multiple ways in a single game engine or program, serving in multiple roles (for example, a network of linked rooms as well as a network of dialogue options).

In Asterism, we implement two collision logics (one in a continuous space and one with a simpler tile-based grid space), a control logic, an entity-state logic, a linking logic, a physics logic, and a resource logic.

Composing a Game Engine

Asterism provides both OLs and a means of composing them. We have built two small game engines using different combinations of OLs, each of which defines specific structural syntheses as types, with fields providing a mapping between the data of their constitutive OLs. One such synthesis is the notion of a `Paddle` in our Atari 2600-inspired `paddles` engine, integrating collision, control, physics, and resource logics:

```
pub struct Paddle {
    pub pos: Vec2,
    pub size: Vec2,
    pub controls: Vec<(ActionID,
                       KeyCode,
                       bool)>,
}
```

A paddle's position and size feed into the collision logic, while the control scheme is given to the control logic. In `paddles` games each player's control is localized in a `Paddle`: part of defining a game means creating some `ActionID` values and keybindings (e.g., move up, move down, serve the ball) and assigning them to particular `Paddles`. The control map in each `Paddle` describes for each action which key triggers that action and whether the action is presently available.

Meanwhile, a `Ball` in `paddles` has a position, size, and velocity, but no controls, participating in collision and physics logics (with its position synchronized between the two). Other syntheses we define in `paddles` include `Wall` and `Score`. This supports not only games like Pong and Breakout, but also our cat-herding game Clowder and *Kaboom!*-like Apple Catching (as shown earlier in Figure 1). While Clowder and Apple Catching are not currently implemented in `paddles` (they are animation prototypes rather than structural synthesis examples), they use the same compositions of OLs.

For convenience, `paddles` defines a set of helper functions (e.g. `add_paddle` and `remove_paddle`) for each structural synthesis, so that whenever new objects of a particular type are created or destroyed their representations in the corresponding OLs are updated. In future work, this type of code could be generated by a macro or from a specification language, or indeed represented as data rather than code. For example, when a paddle is added to the game, the collision and control logics are updated to address the new entity (adding a collision body and defining a controller or keyboard mapping to its movement); when a ball is removed, its collision body and physics state are also removed.

Unlike `paddles`, `boxsy` (the fourth example in Figure 1) is a Bitsy-like engine composing control, linking, and resource logics with a custom collision logic based on discrete, tile-based collision. Moreover, whereas `paddles` provides the full query table interface to its games, `boxsy` only gives the user access to a closed set of operational integrations. Like Bitsy, the engine strictly limits the possible events a user can define reactions for, only allowing them to create links between physical positions on the map or make events trigger when two things touch.

At the moment, shared communication channels are defined mainly in the engine. If Asterism could be parameterized with a rendering system, perhaps using the query tables mechanism, we might be able to explicitly describe communication channels at the Asterism level and offer them to engines globally. This is a key next step in the development of Asterism: to bring mechanisms for audiovisual presentation into the types and constraints governing the connections between logics.

From Logics to Rules

Asterism's OLs, game engines, and games communicate via *query tables*, which hold lists of data generated by OLs that are filtered, zipped with other lists, and otherwise processed in a dataflow style (**?**). As such, every implementation of `Logic` must also implement the `OutputTable` trait both for its internal identifiers (and their data) and for its generated events, allowing for either to be used as a data source for query processing. Engine or game programmers can compose conditions for the query table, which ultimately result in reactions that can be applied back to OLs.

In `paddles`, we have defined a Rust macro (in effect, a tiny DSL) to generate game-relevant queries at compile-time. This macro defines game-specific `Event` types in a predicate/action style, which is one common way of defining operational integrations. A *Pong* game might be defined as in Listings 1 and 2.

Like the query filtering the appropriate types of colliding objects, `bounce_ball` itself is defined as a closure. In these closures, the first parameter is the event (after being processed in the query table), the second is the game state, and the third exposes the OLs provided by the engine. Through these data, a game programmer can produce any operational integration of the `paddles` OLs.

By way of contrast, `boxsy` rules are defined in a more data-driven style (see Listing 3). Resources, characters, tiles, and rooms are the key concepts, and events take place when resource quantities change, collisions occur, or the player moves between rooms; reactions to events can include moving between rooms or changing resource quantities.

Visual Presentation

Game rules are incomprehensible if they are not communicated to players: unless a player sees a reaction when two objects collide on the screen, it is difficult to form a mental model of the simulation. In Asterism, we provide a declarative sprite animation module that ties basic flipbook sprite animations to specific game objects. This module handles loading, storing, and accessing correct image data for a sprite's current animation state, and supports both animated objects and static backgrounds. At the engine level, structural syntheses like characters or terrain can be rendered conveniently by storing animation state with analogous `Ident` types and values, maintaining a mapping between game objects and their appearance (constituting a simple version of shared communication channels).

While ultimately rendering is highly engine-specific, we believe that just as Asterism provides a default collision and resource logic, a default windowing-system-independent 2D rendering system seems valuable. Visual animation state can be tied via query tables to information from the OLs (e.g., when a left-arrow control input is given the sprite can be faced left), or specific animation sequences can be triggered when events occur.

Other Game Engines

We have shown two distinct engines—`paddles` and `boxsy`—implemented in Asterism. Each composes a par-

```
paddles_engine::rules!(game =>
    control: [ /* ... */ ]
    physics: [ /* ... */ ]
    collision: [
        {
          // Define the "bounce" query as a filter...
          filter bounce,
          // over collision events...
          QueryType::ColEvent => ColEvent,
          // between a ball and either a wall or paddle.
          // This closure checks the types of the colliders, but it could
          // instead be represented as data like collision masks.
          |(i, j), _, logics| {
            // From the collision logic, obtain collider types
            let i_id = logics.collision.metadata[*i].id;
            let j_id = logics.collision.metadata[*j].id;
            i_id == CollisionEnt::Ball &&
            (j_id == CollisionEnt::Wall || j_id == CollisionEnt::Paddle)
          },
          // The side effect is to call the bounce_ball function for each bounce,
          // which triggers physics reactions
          foreach |col, state, logics| {
            bounce_ball(col, state, logics);
          }
        },
        // The "score" filter is defined similarly, except that if the ball
        // touches the left or right walls, a resource transaction increasing
        // the opposing player's score is executed.
        { /* ... */ }
    ]
    resources: [ /* ... */ ]
);
```

Listing 1: The `rules!` macro in `paddles` defining a *Pong*-like game.

```
// A closure; i and j are the colliding objects in the collision logic's terms.
let bounce_ball = |(i, j): &ColEvent, state: &mut State, logics: &mut Logics| {
    // Use the composition identifier of the first collider
    let id = state.get_id(*i);
    // Apply this reaction to the ball in particular
    if let EntID::Ball(ball_id) = id {
        // Determine from the collision logic which sides were touched
        let sides_touched = logics.collision.sides_touched(*i, *j);
        // Obtain a copy of the ball's corresponding physics representation
        let mut vals = logics.physics.get_ident_data(ball_id.idx());
        // If this is a touch against the top or bottom...
        if sides_touched.y != 0.0 {
            vals.vel.y *= -1.0;
        }
        // If this is a touch against the left or right...
        if sides_touched.x != 0.0 {
            vals.vel.x *= -1.0;
        }
        // Finally, update the physics data for this ball.
        logics.physics.update_ident_data(ball_id.idx(), vals);
    }
};
```

Listing 2: The `bounce_ball` reaction for a *Pong*-like game, defined in terms of `paddles` concepts.

```
// Define rooms as a grid of tile types...
let rooms = [r#"
00000000
0      0
0   2  0
0      0
0      0
0  3   0
0      0
00000000"#, r#"..."#];
// Then add the rooms to the world
game.add_rooms_from_strs(rooms).unwrap();

// Rocks are a type of resource
let rocks = game.log_rsrc();
// There is one pool of rocks...
let num_rocks = Resource::new();
// registered in the player's inventory.
player.add_inventory_item(rocks, num_rocks);

// There is a link from (3,5) in room 0...
// to (1,1) in room 1
let from = (0, IVec2::new(3, 5));
let to = (1, IVec2::new(1, 1));
game.add_link(from, to);
```

Listing 3: The layout of a map in `boxsy`.

ticular set of operational OLs and defines its own engine-specific syntheses and terms, and each exposes a different level of control to game programmers; the former is more open-ended and allows for new rules in a precondition/postcondition style, while the latter essentially asks game makers to define a data structure giving the game map and dialogue.

Other engines can readily be made using other OLs and renderers: for example, a Twine-like engine would compose linking, selection, and resource logics, while a platformer game engine might build on the `paddles` example with entity-state machines and a linking logic of game levels. As another example, a match-3 puzzle game engine would require defining a spatial matching logic, but could reuse the resource and control logics from Asterism.

5 Future Work

Besides expanding Asterism to incorporate more OLs and develop a declarative account of structural syntheses and communication channels (including not only spatialized channels like character locations but also UI elements, menus, and heads-up-displays), we hope to show that defining engines in shared base terms allows for more generic game design support tools, including but not limited to level editors, game data editors, AI playtesters, and so on. Since the hooks for these tools are defined by the OLs and their compositions, it seems plausible that shared hooks—or at least the trait mechanism underlying them—gives us hope that a small number of tools can be used in a large number of engines. This seems like a good foundation for the engineering of games amenable to automated game design tools (**?**).

We also need to expand Asterism's communication channels beyond visual communication and accommodate audio, haptic, and other forms of feedback. There is also no fundamental reason why generalizing to three dimensional games wouldn't work, since Asterism itself doesn't really commit to any encoding of space.

The choice to use a mechanism like query tables was made for flexibility as well as performance reasons. Measuring whether this table-based approach allows for good performance is important future work, as is expanding the range of table operations beyond zipping and filtering and into proper joins and other iterator operations.

All of the example engines so far require that games themselves are written as Rust programs and compiled with the engine. This is not a fundamental limitation, and engines could instead be implemented in a data-driven style as in e.g. Bitsy.

The most important next step for this project is to find collaborators interested in making game engines and games with Asterism. In particular, we believe Asterism would be well-suited for game design courses where students are asked to make many different kinds of games; traditionally, such courses introduce students to several different tools (e.g., Twine, Bitsy, and GameMaker) but it is plausible that staying within one universe of related tools would be better for students where one super-flexible game engine might have too many degrees of freedom to adequately support non-specialists. This could even be done at the high school level.

One of the authors taught such a course based on the theory of operational logics to early undergraduates. Another author has taught similar game-making courses at the early high school level. Videogames are known to be a motivating domain for early CS students, and the table-oriented style of Asterism could also make it appropriate for teaching stream processing, relational algebra, and related advanced computer science topics.

References

Bogost, I. 2007. *Persuasive Games: The Expressive Power of Videogames*. MIT Press. ISBN 978-0-262-02614-7.

Cardona-Rivera, R. 2020. Foundations of a computational science of game design: Abstractions and tradeoffs. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 167–174.

Cook, M. 2020. Software Engineering For Automated Game Design. In *Proceedings of the IEEE Conference on Games*, 487–494. IEEE.

Doirado, E.; and Martinho, C. 2010. I mean it!: detecting user intentions to create believable behaviour for virtual agents in games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 83–90. International Foundation for Autonomous Agents and Multiagent Systems.

Duplantis, T.; Karth, I.; Kreminski, M.; Smith, A. M.; and Mateas, M. 2021. A Genre-Specific Game Description Lan-

guage for Game Boy RPGs. In *Proceedings of the IEEE Conference on Games*.

Fabian, R. 2018. *Data-oriented design*. R. Fabian.

Gregory, J. 2018. *Game engine architecture*. CRC Press.

Llopis, N. 2010. Data oriented design: Now and in the future. *Game Developers Magazine* 17(8): 31–33.

Martens, C. 2015. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Martens, C.; Summerville, A.; Mateas, M.; Osborn, J.; Harmon, S.; Wardrip-Fruin, N.; and Jhala, A. 2016. Proceduralist readings, procedurally. In *Experimental AI in Games Workshop*, volume 3.

Mateas, M.; and Wardrip-Fruin, N. 2009. Defining operational logics. *Digital Games Research Association (DiGRA)* 4.

McCoy, J.; Treanor, M.; Samuel, B.; Reed, A. A.; Mateas, M.; and Wardrip-Fruin, N. 2013. Prom Week: Designing past the game/story dilemma. In *FDG*, 94–101.

Montfort, N.; and Bogost, I. 2009. *Racing the beam: The Atari video computer system*. Mit Press.

Osborn, J. C.; Lederle-Ensign, D.; Wardrip-Fruin, N.; and Mateas, M. 2015. Combat in Games. In *Proceedings of the Tenth International Conference on the Foundations of Digital Games*.

Osborn, J. C.; Wardrip-Fruin, N.; and Mateas, M. 2017. Refining operational logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–10.

Salter, A.; and Murray, J. 2014. *Flash: Building the interactive web*. MIT Press.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. doi:10.1109/CIG.2013.6633610.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2009. Computational Support for Play Testing Game Sketches. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Summerville, A.; Martens, C.; Harmon, S.; Mateas, M.; Osborn, J. C.; Wardrip-Fruin, N.; and Jhala, A. 2017a. From Mechanics to Meaning. *IEEE Transactions on Computational Intelligence and AI in Games* .

Summerville, A.; Osborn, J. C.; Holmgård, C.; Zhang, D.; and Mateas, M. 2017b. Mechanics Automatically Recognized via Interactive Observation: Jumping. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Summerville, A.; Osborn, J. C.; and Mateas, M. 2017. CHARDA: Causal Hybrid Automata Recovery via Dynamic Analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-O-Matic: Generating Videogames That Represent Ideas. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, 11:1–11:8. ACM. ISBN 978-1-4503-1447-3. doi:10.1145/2538528.2538537. URL http://doi.acm.org/10.1145/2538528.2538537.

Treanor, M.; Schweizer, B.; Bogost, I.; and Mateas, M. 2011. Proceduralist Readings: How to find meaning in games with graphical logics. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 115–122. ACM.

Wardrip-Fruin, N. 2005. Playable media and textual instruments. *Dichtung Digital* 34: 211–253.

Wardrip-Fruin, N. 2006. Expressive Processing: On Process-Intensive Literature and Digital Media.