Building a Customizable Virtual Environment for Studying Human Compositional Generalization

Ege Ersü

Master of Science

Cognitive Science

School of Informatics

University of Edinburgh

2021

Abstract

We developed a customizable virtual environment that allows researchers to collect behavioral data on how humans acquire rules and compositionally generalize from one rule system to another. The environment is built as a survival game where the player has to gather resources, combine those resources to craft more advanced items and kill enemies to stay alive until the timer runs out. We use our application to launch an example online experiment and gather more than 2000 crafting attempts from 179 participants. We finish the project by performing exploratory data analysis on the collected dataset and proceed to test two cognitive hypotheses on compositional generalization.

This work contributes to the understanding of human concept acquisition and compositional generalization by proposing a new experimental framework inspired by recent literature. We believe the customizable nature of the application will allow researchers to design and launch their own experiments through a single configuration file without having to touch the source code and collect large-scale behavioral data in a virtual environment.

Table of Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Outline 3

2 Background **4**

2.1 Compositional Generalization 4

2.2 Compositional Generalization in Machine Learning 5

2.2.1 Sequence-to-Sequence Translation 5

2.2.2 Learning to Apply Functions 6

2.3 Compositional Generalization in Psychology 7

2.3.1 Compositional Cognitive Models 7

2.3.2 Human Few-shot learning of compositional instructions . . . 8

2.4 Our Framework 9

2.4.1 Primitives 9

2.4.2 Rules 9

2.4.3 Valid & Invalid Attempts 10

2.4.4 Primitives as Resources 11

3 Game Design **12**

3.1 Crafting 12

3.1.1 Crafting System 13

3.1.2 Game Phases: Survive & Craft 14

3.1.3 Survival: Health & Hunger 15

3.1.4 Resources: Food & Weapons 15

3.2 Resource Gathering 17

3.2.1 Enemies 18

4 Implementation **20**
4.1 Using React as a Game State Manager 20
4.2 Map, Camera & Movement 21
4.3 Collecting & Using Resources 22
4.4 Combat & Enemies 23
4.5 Crafting Screen 25
4.6 Building online experiments: React, Netlify & Airtable 26

5 Experiment Design & Data Analysis **27**
5.1 Experiment Setup 27
5.1.1 Implementing the Experiment 28
5.1.2 Rulesets 28
5.2 Exploratory Data Analysis 31
5.2.1 Game Completion 31
5.2.2 Crafting Statistics 31
5.3 Hypotheses Testing 33
5.3.1 Generalization from Strict Rules to Flexible Rules 33
5.3.2 Generalization from Flexible Rules to Strict Rules 34

6 Conclusions **35**
6.1 Crafting 35
6.2 Future Work 35
6.2.1 Functions 35
6.2.2 Exploration & Exploitation 35
6.2.3 Input and Output Sizes 35

Bibliography **36**

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ege Ersü*)

Chapter 1

Introduction

1.1 Motivation

Humans have the remarkable capacity to understand and produce novel expressions that are composed of known concepts. For example once we learn the meaning of "meditate", we do not require any additional training to understand and produce expressions such as "meditate every morning" or "meditate while breathing". We also know that some other compositions like "meditate while dancing" or "I'll have meditate for lunch" are invalid, without any external supervision. Cognitive scientists have called this compositional ability of ours **systematic compositionality**, a property of our cognition that allows us to systematically generalize in a data-efficient and flexible way from only a handful of examples. [11, 24].

If we choose to conceptualize this ability as a primary characteristic of human learning, then we would also expect our machine learning models to adopt similar mechanisms. This would be especially helpful when our models are expected to generalize from only a small set of training examples to unseen compositions of what it has already seen. Being able to understand highly compositional inputs is not only a bonus feature for our models, but also a real practical problem within the industry. With the increasing popularity of Deep Learning and the availability of vast amounts of data, we now have models that can accurately represent complex visual scenes with many objects, or answer highly compositional questions such as *"What size is the cylinder that is left of the brown metal thing that is left of the big sphere?"* [16].

Lately there has been a new line of research trying to show that deep networks rarely have to show perfect systematicity to accurately process such test cases containing various compositions of known components. [19]. Their experiments have shown

that given enough training data containing many unique compositions, deep networks can learn to solve tasks such as Visual Question Answering and Grounded Instruction Following where the inputs are highly compositional [19, 16]. If that is the case, these highly-curated datasets would be doing most of the work as opposed to the deep models that are commonly argued to possess *human-level* generalization capacities.

Building on this, researchers started asking if deep networks can generalize equally well from a smaller training set, containing only a handful of unique compositions. This caused a new surge of synthetic benchmarks to be designed, with experiments confirming that they indeed do not generalize systematically [19, 14, 2, 9, 17]. This has caused further research aimed at discovering the cognitive mechanisms and inductive biases behind human generalization, through experiments that are similar to what neural networks are tested on [20, 21]. This project builds on top of that paradigm, where the goal was to build a customizable framework allowing users to modify the environment according to their own research hypotheses.

Instead of imitating more traditional web-based experiments, we have decided to design the framework as a virtual environment in the form of a browser game. Recently there has been many successful examples of object-oriented games being used to conduct rigorous experiments in Reinforcement Learning, Psychology and Neuroscience [5, 12, 6, 10]. Embedding the experiment into a well-designed game has the potential to increase the participant's engagement with the task [25, 27, 23], while also making it easier to find participants without having to spend any money on coupons or external crowdsourcing services. Other than these more practical benefits, it allows us to study how the participant's gameplay decisions evolve over time conditioned on the feedback they receive from an environment. This allows us to collect behavioral data that is more complex (but potentially harder to analyze) than the ones produced by more static concept learning experiments [24, 20, 14].

1.2 Outline

The dissertation starts with a Background chapter on compositional generalization, where we introduce the specific cognitive tasks that we think would benefit from a new tool for collecting large-scale behavioral data. In Chapter 2 we describe a game with the relevant crafting system and justify our game design choices. In Chapter 3 we dive into implementation details and explain how we built the virtual environment as a web application. In Chapter 4 we present our Exploratory Data Analysis on the behavioral dataset we have collected using the framework and use it to evaluate two cognitive hypotheses. In Chapter 5 we summarize our conclusions and propose specific experimental setups for future work, which could be easily launched online using our framework.

Chapter 2

Background

In this chapter we will introduce the experimental framework on which we built our environment. We will start by looking at influential compositional generalization tasks from the Machine Learning literature. We will then identify relevant experimental tasks from Psychology, which we believe are relevant for the type of generalization we are interested in. By the end of the chapter we will propose our own data collection framework, formalize the structure of the data we will be collecting and argue why we think it is novel and useful.

2.1 Compositional Generalization

We will start by admitting that the concept *compositional generalization* does not have a fixed formal definition that is constant across fields and research groups. Although arguing for a unified definition is beyond the scope of this paper, we will focus on two properties called *Systematicity* and *Productivity* which have received attention throughout time and still do to this day [7, 15, 1]. Our goal will be to design a framework which could at least be used to test for these two features.

Systematicity is commonly defined as the ability to understand novel compositions of already known concepts [7, 11]. For example if an agent knows the meaning of these three words: {jump, twice, and}; then it should be able to understand this bizarre sentence: "jump twice and jump and jump twice'. If we use a behavioral criterion, an agent that fails to jump five times after that instruction would not have systematicity. Ideally they would have to understand all possible compositions of these three *primitives* and produce the appropriate amounts of *jumps* accordingly. One might even argue that they should also realize which compositions are invalid, such as: "and jump and'.

Productivity is the second property we are interested in, which is more concerned with the length or depth of compositions. A generative view of language would imply that infinitely many sentences can be generated from a finite set of rules and components [7]. To build on our previous example, an agent that knows the meaning of "jump and jump' should also be able to understand "jump and jump and jump and jump and jump' if it can *generalize productively*. The focus here is on the ability to generalize from shorter compositions to longer compositions, given that these longer sequences were not seen before. Although this computational definition surely is too simplistic from a Linguistics perspective, it helps us compare the performance of humans and machine learning models in terms of rule acquisition.

2.2 Compositional Generalization in Machine Learning

We will now introduce two popular compositional generalization benchmarks from Machine Learning which have motivated further research in Psychology. We should note that there are many other useful benchmarks designed with similar goals [1, 17, 1, 28, 4, 3], but we believe introducing these two will be sufficient to motivate the type of sequence-to-sequence tasks we are interested in.

2.2.1 Sequence-to-Sequence Translation

jump	⇒	JUMP
jump left	⇒	LTURN JUMP
jump around right	⇒	RTURN JUMP RTURN JUMP RTURN JUMP RTURN JUMP
turn left twice	⇒	LTURN LTURN
jump thrice	⇒	JUMP JUMP JUMP
jump opposite left and walk thrice	⇒	LTURN LTURN JUMP WALK WALK WALK
jump opposite left after walk around left	⇒	LTURN WALK LTURN WALK LTURN WALK LTURN WALK LTURN LTURN JUMP

Figure 2.1: SCAN Dataset: sequence-to-sequence translation [19]

The first benchmark is an instruction-following benchmark called SCAN [19], where the task is to read an instruction like "turn left and jump twice' and output the correct action sequence: [TURN-LEFT, JUMP, JUMP]. We provide some samples from the dataset in Figure 2.1. Given enough data, this has been shown to be a trivial task for sequence-to-sequence recurrent neural networks. But once the dataset was split to test for systematicity and productivity, neural networks failed to generalize compositionally.

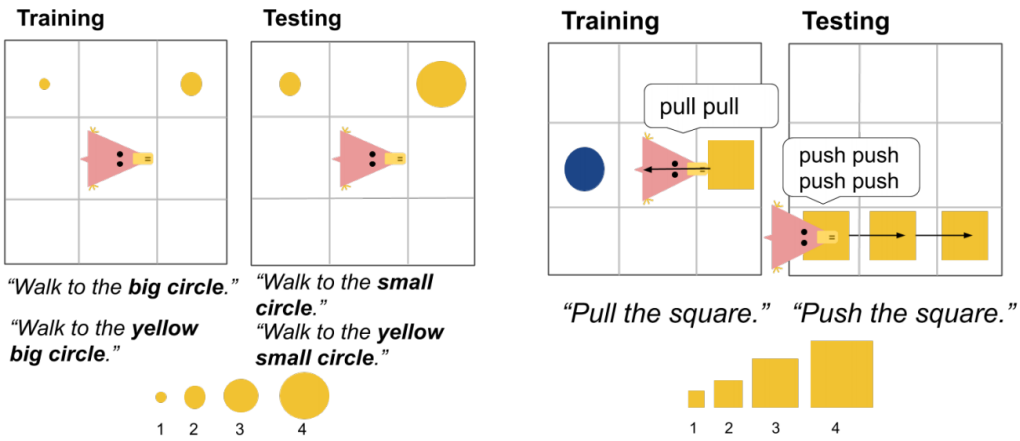

Figure 2.2: Grounded SCAN: Generalizing systematically [26]

To test for productivity the training set was made to include shorter instructions, while the test set consisted of longer instructions. Even though they were chaining together identical primitives like "jump and jump and ...', the network still failed to generalize productively. To test for systematicity, they did another data split which required the network to generalize by recombining pieces of training commands. We present some examples from the grounded version of SCAN in Figure 2.2.

2.2.2 Learning to Apply Functions

```
repeat A B C                    →  A B C A B C
echo remove_first D K , E F     →  E F F
append swap F G H , repeat I J  →  H G F I J I J
```

Figure 2.3: PCFG: Learning to apply functions [15]

The second benchmark that was explicitly designed to test the systematicity and productivity of neural networks is the PCFG dataset [15]. Although the dataset preserves the sequence-to-sequence nature of SCAN, it makes a distinction between *primitives* and *functions*, which is a customizable property of items on our web application. To give an example, applying the function REPEAT to the list of primitives [A,B,C] produces the primitives [A,B,C,A,B,C]. The functions can also be composed together, just like composite functions in algebra. This makes arbitrarily long compositions a possibility, allowing varying difficulties of tests for both systematicity and productivity. Some other examples are given in Figure 2.3.

The authors proceed to split the dataset in a similar way to SCAN, experimenting with neural architectures like recurrent neural networks, convolutional neural networks and transformers. They report that the task is once again trivial given a large enough dataset, but the performance drops drastically once they test for systematicity and productivity. They conclude that for a model to generalize systematically and productively, it should be able to learn to apply arbitrary functions compositionally without having to see a large number of examples.

2.3 Compositional Generalization in Psychology

Now that we have covered two benchmarks on compositional generalization from the Machine Learning literature, we will now introduce two psychological tasks which we think are relevant.

2.3.1 Compositional Cognitive Models

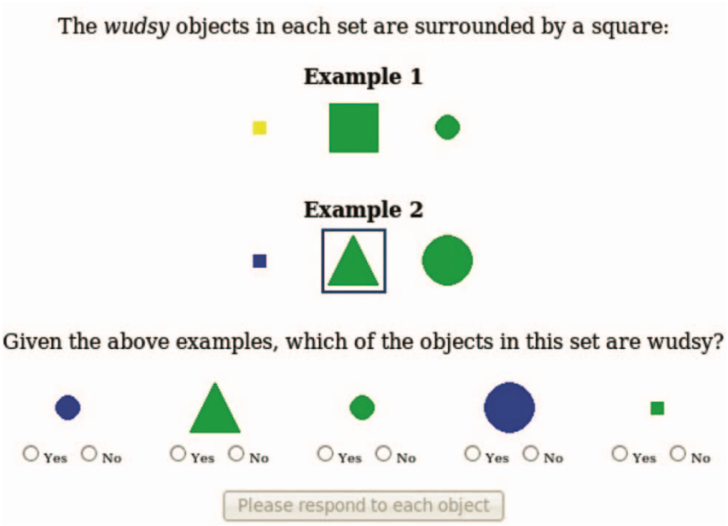

Figure 2.4: Concept Learning Experiments: Generalizing to a new set [24]

Compositional language of thought (LOT) is a particular view of cognition that assumes we start with a set of primitives, and then compose those primitives to build up more complex psychological concepts [11, 24]. While some research starts with assuming a particular set of primitives that are readily available to us, the concept learning experiments we are interested in do not make such an assumption. They give participants an arbitrarily named concept like *Wudsy* but do not give an intrinsic

definition of what it means. They are instead given a set of *Wudsy* objects to serve as a training set, which the participant studies to later make a prediction on whether some new unseen objects are *Wudsy* or not. An example is shown in Figure 2.4. The collected behavioral data is then modeled using Bayesian methods to predict distinct learning curves, where they iterate over each possible set of *LOT primitives* to find which one is more likely to have been the true compositional primitives.

2.3.2 Human Few-shot learning of compositional instructions

We will now look at a cognitive task that has been used to directly compare humans with end-to-end neural architectures [20]. Although this framework does not aim to find the underlying compositional primitives, it tries to identify some of the inductive biases human might be utilizing when they learn a new rule. The experimental setup is almost identical to the computational tasks we looked at, where the goal is to translate a sequence to another sequence, according some underlying rule.

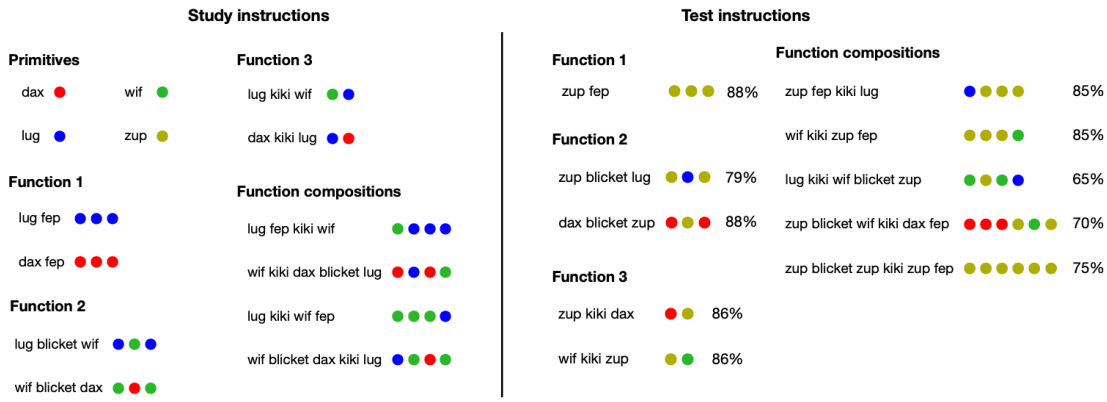

Figure 2.5: Few-shot learning of compositional instructions [20]

Just like the PCFG dataset, they have made a distinction between primitives and functions that take primitives as their arguments. We present some of their functions and primitives in Figure 2.5. For example the function FEP simply makes three copies of the primitive to the left of it, similar to the function REPEAT from the PCFG dataset that made two copies of the primitives to its right.

Their experiments show that people can indeed learn to use these functional concepts correctly after studying a tiny training set, and start applying them to unseen primitives. They also show that humans naturally start composing these functions in complex ways without any external supervision. Although the specific inductive

biases they propose are beyond the scope of this paper, their experiments involve giving participants the freedom to construct sequences of arbitrary length and composing functions however they want. To accommodate such experiments, we have also made input length and style customizable features.

2.4 Our Framework

2.4.1 Primitives

Our framework builds on top of these rule-learning tasks, but gives participants the flexibility to experiment with different sequences over time. So instead of studying a predefined training set and then being tested on unseen examples, they learn by playing and testing out different compositions. This allows us to collect behavioral data on how they acquire these rules *over time* through trial-and-error, rather than just studying a fixed dataset given to them. To make things more concrete, let us define the following primitives:

$$P = \{p_1, p_2, p_3, p_4\}$$

Now that we have our primitives, we can create sequences of arbitrary length. Here are some example sequences we can form:

$$p_1$$

$$p_1 p_1$$

$$p_2 p_2 p_2$$

$$p_1 p_2 p_3$$

$$p_1 p_1 p_1 p_1 p_1 p_1$$

2.4.2 Rules

In the tasks we looked at previously, each sequence was mapped to another sequence by some *rule*. For this example, let us define a simple **summation** rule. We stipulate that the resulting output will be a single primitive, but will have a subscript equal to the sum of the input subscripts. Here are some examples:

$$p_1 p_1 \rightarrow p_2$$

$$p_1 p_1 p_1 \rightarrow p_3$$

$$p_2 p_2 \rightarrow p_4$$

A set of primitives and a rule is all we need to replicate the experiments we saw before. For example to add the function FEP (which made three copies of its arguments) as the symbol f , we can just add the following rules to our system:

$$p_1 f \rightarrow p_1 p_1 p_1$$

$$p_2 f \rightarrow p_2 p_2 p_2$$

$$p_3 f \rightarrow p_3 p_3 p_3$$

$$p_4 f \rightarrow p_4 p_4 p_4$$

2.4.3 Valid & Invalid Attempts

Although this sequence-to-sequence framing is highly useful for making direct comparisons with machine learning models, we also want to study if humans can distinguish valid sequences from invalid sequences. Therefore we define **invalid attempts** to be those sequences that validate the rules and they produce no output. To give an example, let us modify the summation rule to be more strict. We can further stipulate that for a summation to be valid, the subscripts must be equal. If they are equal, we sum them up. If they are not equal, we do not return an output.

$$\begin{cases} p_{i+j} & i = j \\ \emptyset & i \neq j \end{cases}$$

Now once the participant starts experimenting with sequences, some of the sequences will be producing outputs and some will not be producing any outputs. This means they will be receiving a feedback signal from the system, indicating whether their composition was valid or not. For example here is an example progression of attempts where the participant *figures out* that the subscripts have to match:

$$r_1 r_1 \rightarrow r_2$$

$$r_1 r_2 \rightarrow \emptyset$$

$$r_2 r_1 \rightarrow \emptyset$$

$$r_2 r_2 \rightarrow r_4$$

$$r_1 r_1 r_2 \rightarrow \emptyset$$

$$r_1 r_1 r_1 \rightarrow r_3$$

$$r_1 r_1 r_1 r_1 \rightarrow r_4$$

2.4.4 Primitives as Resources

To finally turn this experimental framework into a game mechanism, we decide to treat primitives as resources. For example we could give the participant 10 many r_1 primitives to start with and tell them to form sequences as they please. If they make an invalid attempt, their resources go to waste since there will be no output. But if they make a valid attempt such as $r_1 r_1 \rightarrow r_2$, they will earn this new r_2 primitive. This can now be used in another attempt, which will produce new primitives. Although our primitive domain is small in this example, this could be used to study how humans explore highly complex compositional domains through time by trial and error as opposed to studying a fixed training set in isolation.

At this point our framework consists of a set of **primitives** and a set of **rules** governing what the output will be. The participants can form arbitrarily complex sequences by using the primitives given to them, and earn new primitives if their **attempts** are **valid**. If their attempt is **invalid** under the rules, they receive indirect negative feedback by not receiving any outputs.

Currently this setup is highly artificial and is unlikely to motivate participants to produce more primitives or try out new compositions. We will solve the former problem by making sure primitives actually do something within the virtual environment, so that they are desirable. We will also solve the latter problem by making sure deeper compositions produce primitives that are exponentially more desirable, by introducing a sense of progression such as $r_4 > r_3 > r_2 > r_1$, so that participants will be incentivized to dig deeper.

Chapter 3

Game Design

Our primary goal was to design game mechanisms that were strict enough to accommodate our experimental framework, so that the produced behavioral data had actual scientific value for studying rule acquisition and compositional generalization. We only added entertaining game mechanisms if they incentivized gathering and crafting more primitives, or if they pushed the player to explore deeper compositional possibilities.

We decided to build the game as a survival game, where the goal is to stay alive by consuming food resources and fighting enemies using weapon resources. The virtual environment is kept relatively simple, containing only resources that can be gathered and enemies that start chasing you if you get too close. We believe the survival genre was the perfect fit to our task, since collecting resources is necessary for beating the game, and it feels natural to embed a crafting system where you can combine those resources. [1]

3.1 Crafting

Crafting is a game mechanism that allows the player to combine their gathered resources to produce new resources that are generally more useful or stronger. Although combining arbitrary primitives is not an intrinsically pleasurable activity, when it is placed within the context of a game where you bring new virtual items into existence, it turns into a creative activity where players are inclined to explore new possibilities [13, 8].

Crafting has been popularized by influential survival games such as Minecraft [22]

[1] The game is live at `https://romantic-sinoussi-657706.netlify.app/`

and Don't Starve [18], which provide highly immersive environments with thousands of unique resources and crafting rules. But ufortunately the environmental complexity and the lack of customization options makes it difficult to conduct rigorous experiments on these platforms. Although this genre inspired us greatly in our game design, we had to simplify most game features to allow for more rigorous experiment design.

3.1.1 Crafting System

Most crafting systems lie somewhere in between *strongly defined recipes* and *undefined recipes* [13]. The game Don't Starve specifies strongly defined recipes through an in-game *cook-book*, where the player can look up what they need to combine to produce a desired output. In comparison, undefined recipes do not give any information to the player, but instead give them the flexibility to combine any n number of resources together, in whatever combination they want. The player has to figure out what works and what does not work on their own, which is exactly what we need to implement for our experimental framework.

We base our design on the crafting menu from Minecraft, which can bee seen in Figure 3.1. The player can see their collected resources on the **inventory** section down below. They can also drag resources from their inventory, and place them onto the **inputs** section of the menu to be used in crafting. When they click the CRAFT button, the game checks the resource configuration against all possible rules and places the appropriate output resource on the **outputs** section to the right. The player can now click on this newly crafted resource and it will appear on their inventory. For the Minecraft example, the player used 3 Wooden Sticks, 1 Coal and 3 pieces of Wood to craft a Campfire, which can now be used in the virtual world to light a fire.

Figure 3.1: Minecraft's Crafting Screen [22] (Left) and our Crafting Screen (Right)

In Minecraft you not only have to place the appropriate resources, but also have to put them in the correct boxes on a $3x3$ grid. Remember that our experimental framework only allows a sequence of items to be converted to another sequence of items, so

we only need input boxes to be placed on a horizontal axis. In Minecraft all crafting attempts produce a single item, but we want to be able to return a sequence of items when necessary (Remember the FEP function denoted by f: $p_1 f \rightarrow p_1 p_1 p_1$). It is also crucial that these input and output boxes are implemented to be customizable, so that the number of boxes can be controlled by the experiment designer.

3.1.2 Game Phases: Survive & Craft

Now that we have a crafting system that implements our experiment, we have to design a game around it. While our crafting system is heavily inspired by Minecraft, our gameplay cycle is inspired by the popular survival game: Don't Starve. We barrowed from them the concept of a **Day**. If the player survives Day 1, they get to move on to Day 2 and then to Day 3, and so forth. We could have a single day or 12 days depending on the experiment we want to design.

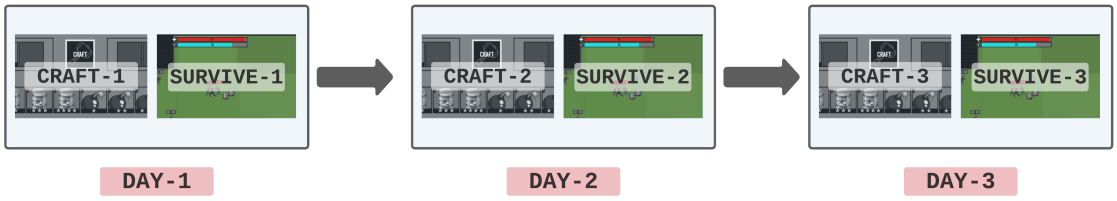

Figure 3.2: Game Cycle: Survive & Craft Phases

We believe dividing the game experience into these distinct Days allows us to design experiments with some resemblance of traditional experimental trials. By making the Days customizable, we give experiment designers the option to add new primitives, adjust the item distribution or change the rules between trials. For example if they want to study generalization from one rule system to another; they could use some rule R_1 for the first 3 days, and some other rule R_2 for the next 2 days. They could also chain 20 days one after another, each one with unique set of primitives and rules. We present an example experiment with 3 Days in 3.2.

Now that we have the concept of a Day, we divide each day into two distinct phases: The Crafting Phase and the Survival Phase. During the **Crafting Phase** the player is presented with the crafting screen (shown in Figure 3.1) and they use their gathered resources to craft new items. Once they are done with crafting, they transition to the **The Survival Phase** of the same day. During the survival phase the player moves

around on a large map, filled with resources and enemy creatures. The goal is to stay alive until the time runs out and the survival phase is over. Once the survival phase is also over, we transition to the next Day. Once the last day is over, the player wins. If they die before completing the last survival phase, they lose.

3.1.3 Survival: Health & Hunger

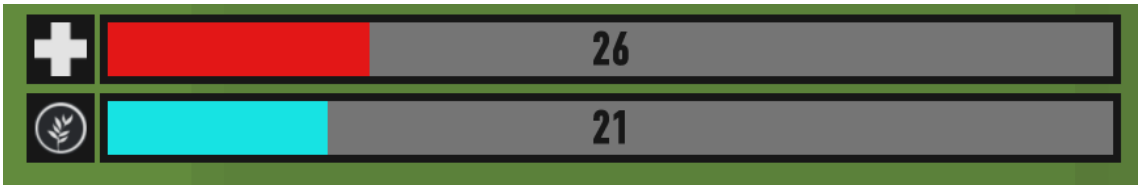

Figure 3.3: Player States: Health & Hunger

Like most other survival games, the player in our game also has two fundamental game states: **Health** and **Hunger**. These are both integers between 0 and 100. If the player's health drops to 0, they die and lose the game. If the player's hunger drops to 0, they start losing health points every second. A player can lose health points only if they get attacked by an enemy, but they automatically lose hunger points every second.

3.1.4 Resources: Food & Weapons

Figure 3.4: Food & Weapon Items

Now that we have the crafting system and player states in place, we have to turn these primitives into actual resources that can be used within the game. The first type of resource is **Food**. When consumed, food resources restore a certain amount of hunger and health points. So even though the player automatically loses hunger points every second, they can consume a couple of food items to set it back to 100. Similarly if they lost health points due to starvation or combat with enemies, they can also restore those lost health points.

Going back to our primitives and rules in chapter 2, we also want to introduce a mathematical ordering between similar primitives like $\{p_1, p_2, p_3, p_4\}$ so that we can additionally study mathematical rules like summation. To achieve this, we also assign a **level** to each type of resource, which is an indicator of how strong or useful it is. As displayed in Figure 3.4, the level of a resource is indicated by the number of stars. For example in this example we have 4 types of food resources: $\{food_1, food_2, food_3, food_4\}$. We also have to make sure the players are incentivized to combine their lower level items together to produce higher higher level items. We achieve this by making sure items get exponentially better as they are leveled up.

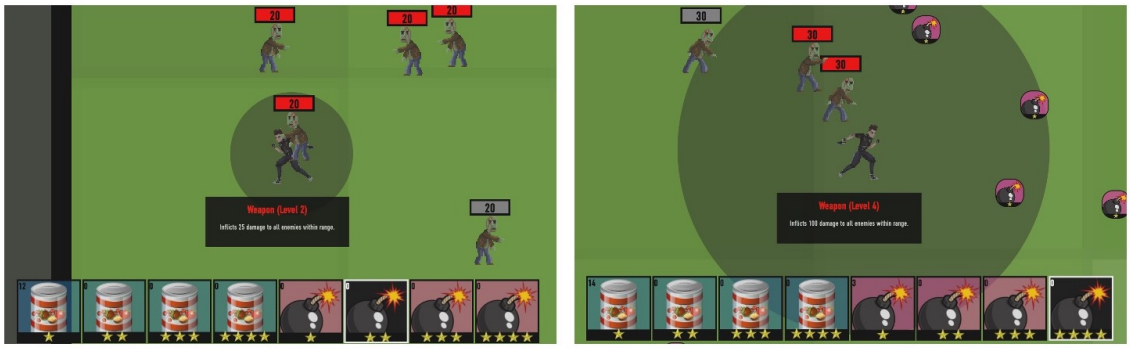

Figure 3.5: Weapon (Level 2) versus Weapon (Level 4) in terms of impact radius

The second resource we have in the game are **Weapons**. Consuming a weapon damages all enemies within a certain radius, and is useful for clearing out enemies that are chasing you. Although the player can usually outrun a couple of enemies, it becomes a risky tactic when there are too many of them. Similar to food resources, weapons also have a level attached to them. As the level increases, we make sure weapon's damage and impact radius increases exponentially; meaning they are more useful for staying alive. Although all of these resource properties are fully customizable, we present our custom parameters below:

Food Level	Health Restored	Hunger Restored
1	10	10
2	30	30
3	50	50
4	100	100
Weapon Level	Damage	Radius
1	10	200
2	25	300
3	40	400
4	100	700

We also add an optional item type called **Functions**. These are not resources that can be used within the survival phase like food & weapons, but can only be used as additional inputs in crafting. For example the user might find a mysterious function item that makes three copies of the resources next to it like FEP or REPEAT [20], one that swaps the level of two items like SWAP [15], or any other function that the experiment designer chooses to add to that Day's crafting rules.

3.2 Resource Gathering

In traditional experiments we can limit the number attempts that can be made by a participant by hard coding an artificial limit. If we do not limit the number of attempts they can make within a crafting system, we face the risk that our data might contain too many low quality or purely random attempts. Thankfully game designers across the years came up with more creative ways to solve this problem. In most survival games, crafting materials are designed to be a scarce resource required for survival. This way the player has to actually spend time and energy collecting resources to earn the right to make a crafting attempt, which forces them to be more careful and think harder before making a decision [13].

In our game there are two ways of acquiring resources: picking them up from the floor or killing enemies. At the start of the game we use a customizable distribution to place food & weapon resources on the map. As the player explores the environment during a survival phase, they can pick up these resources that are scattered around them which can be seen in Figure 3.6. They can also use their weapon resources to kill enemies, which in turn drops additional resources that can be picked up by the player.

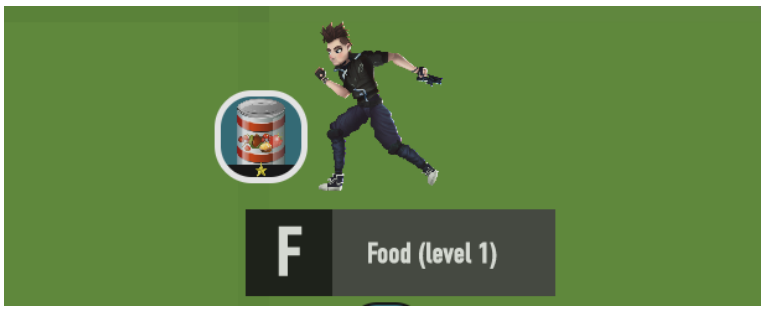

Figure 3.6: The Player can pick up nearby resources by hitting the F key.

3.2.1 Enemies

As the final component of our game, we introduce the enemies. Although this might seem like an optional game mechanism considering the experimental purpose of this environment, we believe having a real failure condition that is constantly chasing you would demand more attention from the player. It also gives the player a reason to collect & craft weapon items, allowing us to gather data on two different types of resources with possibly different crafting rules.

An enemy is always in one of two states: **Idle** or **Following**. They always start in the idle state, meaning they randomly walk around the map without any purpose. But if the player comes near an enemy, they transition into following state. Once the transition happens, they stay in the following state until the end of the survival phase, following the player where they go. If an enemy manages to catch the player, it starts damaging the player's health points every second. Just like the player, enemies also have health points which can be damaged using weapons. Once their health point hits 0, they die and drop a resource to serve as a reward to the player.

Additionally we also give experiment designers the capacity to set the difficulty of enemies by adjusting their speed, damage and aggression distance (the minimum distance between an enemy and the player which would not trigger a state transition). They can also control the number of enemies present in each Day, meaning they can turn them off entirely if they desire so.

Figure 3.7: Don't Starve [18] (Left) and our game's Survival Phase (Right)

Chapter 4

Implementation

We will now dive into the specifics of how the virtual environment was implemented as a web application. Due to the sheer number of game objects and interactions proposed in our game design, most of the effort was put into systematically building up a codebase that would allow the game to run on different types of devices, without causing overheating or unpleasant glitches that would distract the participant's engagement with the experimental task. [1]

Another big component of the proposed framework was to code the application in a way such that a single configuration file would control everything. That is why most of the effort was put into software engineering, designing every single class and React hook in a way that supports full customization of our proposed features. We believe diving into the implementation will produce a useful report that could help potential researchers who are interested in studying cognition, but are unsure about how to start building up an interactive environment that could produce relevant behavioral data for their research.

4.1 Using React as a Game State Manager

React is one of the most popular front-end frameworks used in the industry, mostly due to its emphasis on building interactive user interfaces. The framework achieves this through **Stateful Components**, portions of the interface that stores and manages its own internal data. Whenever the internal data of some component changes, React automatically re-renders that component to reflect what has changed. As opposed to using vanilla JavaScript where the entire UI has to be rendered after each change

[1] The codebase is available at `https://github.com/egeersu/compose-io`

(although this can be avoided using other frameworks or programming tricks), React allows us to optimally render any changes that might happen to the Player, Enemies or Items.

Using React also allows us to combine our game logic with UI. For example traditionally, we would have to maintain a Player class to store our Player's health and location; and then write a separate program for putting the player on the canvas and handling the animations. This would require us to first implement complex operations to represent the game state properly and then also write functions to convert the representation into UI components. In React they are one and the same. We implement the Player class as a stateful component of our UI, which also holds all the required game states. For example the player's x-coordinate as a *game state* is one and the same with the UI component's position on the canvas, relative to the Map.

4.2 Map, Camera & Movement

We wanted to create a large map, which could potentially have hundreds of enemies and resources if the experiment designer desires so. But displaying the entirety of such a map within a browser screen requires all other objects to be extremely small. To get around this problem, we create two separate React components: Map & Camera; and implement a common visual trick used in browser games, which we try to visualize in Figure 4.1.

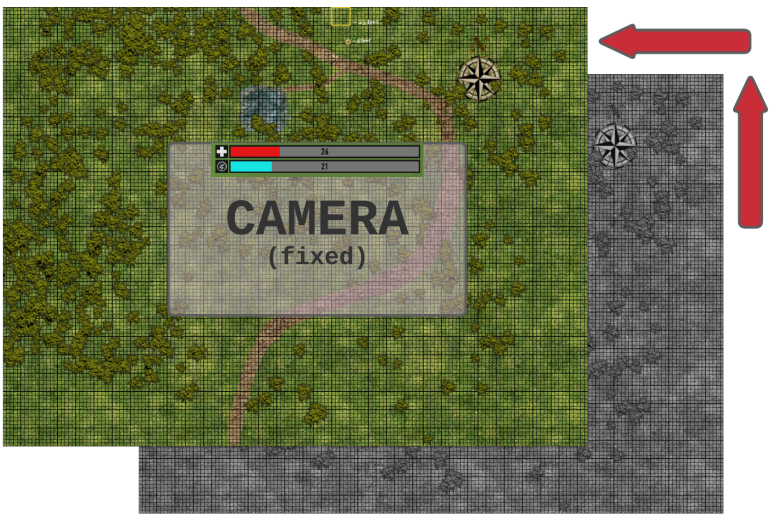

Figure 4.1: The Visual Trick for Simulating Movement & Exploration

The Camera is the component we design to stand for the player's viewpoint. It covers the entire browser screen, has the action bar, health bar and other relevant UI components placed on it. The Camera's position never changes, which makes sure the UI elements always stay where they are relative to the browser. Independent of the Camera component, we have the Map component. The map is typically too large to fit within a browser screen, and it contains all game objects such as the player's avatar, enemies and resources that can be picked up. In most games it is the camera that moves, but in browser games the camera stays constant.

Let us say the player pushes the DOWN key, with the intention of travelling South. Our movement function does two things. It first moves the player South relative to the Map. So if the player was at position (x,y) it is now at $(x,y+dy)$ relative to the Map. Second, we also move the Map in the opposite direction, which in this case is North. Since the Camera stays fixed, pushing the Map North and the player South with respect to the Map, creates the illusion of moving downwards. If these words do not make much sense on their own, we advise playing around with the application while using the React developer tools on your browser to check positional values. In addition to this trick, we write our own React movement hooks to handle when there are multiple key presses at the same time (Ex: Holding UP & RIGHT at the same time and then releasing the RIGHT key to keep going Up.).

4.3 Collecting & Using Resources

Figure 4.2: Information Panel for Picking Up Resources

When the player approaches a resource, we highlight its borders and present an information screen indicating that they can now pick up the resource, as in Figure 4.2. Resources that are further away which cannot be picked up do not have such a highlight. We achieve this by making each resource have a variable that holds the distance between that resource and the player. Whenever the player moves, these distances are

recalculated. Although the current naive implementation does not raise any issues, for gigantic maps it would be more reasonable to implement a grid system. In a grid system each resource and the player would belong to an individual grid. When the player moves, only the resources within the player's grid would trigger a computation.

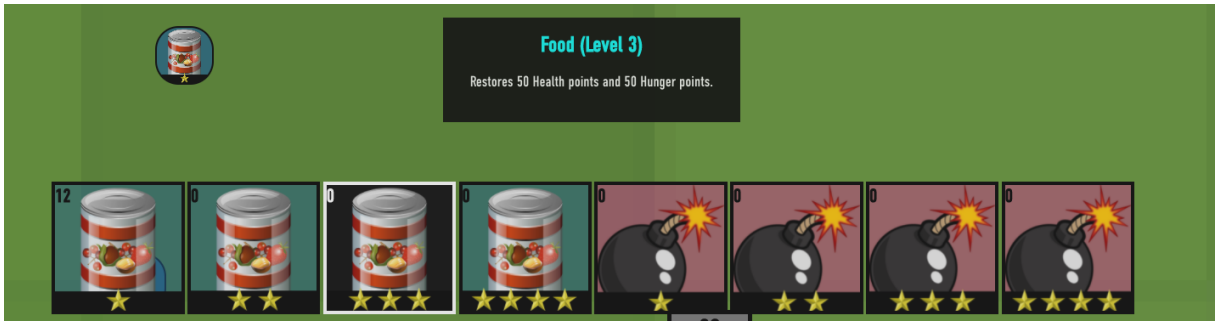

Figure 4.3: Information Panel for Consuming Resources

We position the usable resources on an *Action Bar* which is fixed on the Camera. The player can use an item by clicking on the item. If they hover over the item, we display an information box that gives the name, level and description of the item. We display all information panels right below the player's position with respect to the camera, so that it stays fixed relative to the browser even if the player is moving.

4.4 Combat & Enemies

We allocate most of our computational resource to handle combat and enemies. Just like resources, they also have to store the distance between them and the player, which is used to trigger state transitions from IDLE to FOLLOWING. Every frame, all enemies in the FOLLOWING state also have to compute a movement vector that points towards the player's location. We then use these movement vectors to update the location of all enemies, which results in the following behavior. To make sure IDLE enemies also pose a threat, we make them move around the map randomly. This is achieved by picking a random location on the map for each IDLE enemy, and then making them compute movement vectors between them and this random location. When they reach that random location, we pick another random location on the map to continue the behavior. Once enemies reach the player (i.e. their distance to each other falls below a certain range), we start inflicting damage to the player's health point.

To make the game more engaging, we also animate the enemies using a royalty-free online sprite sheet. The walking and chasing movements are achieved by looping

Figure 4.4: Animating enemies using sprite sheets.

through a list of 30 images every frame, and changing the enemy component's background image. Each enemy has to keep track of their location within this sequence of images, which is incremented every m miliseconds. By playing with m, we can adjust the speed of the running animation. We also add a Health bar on top of each enemy's avatar, which shows their remaining health points. Whenever an enemy is damaged, the red within the bars are updated to reflect its current state. We use the same animation strategy to animate the player avatar, but by making sure the pointer is incremented only when there is movement; which enables us to stop the avatar when the player is not moving.

Figure 4.5: Weapons damage all enemies that fall within the impact radius.

To make sure players focus more on crafting rather than game mechanics like shooting weapons, we keep the combat system simple. When the player hovers over a weapon they have, we display the impact radius using a dark circle, which can be seen in Figure 4.5. The circles are adjusted based on the weapon strengths and radii specified in the customization file. Once a weapon is consumed, we retrieve all enemies that fall within that radius and reduce their health points accordingly. Since enemies tend to gather up in large groups when chasing the player, the player can just use two level

4 weapons to clear them all at once. This further incentivizes making higher order compositions to craft stronger items. If an enemy's health drops to 0 we change their avatar with a skull sign indicating they are dead, while also triggering a new item to spawn at that location based on the resource distribution specified in the configuration file.

4.5 Crafting Screen

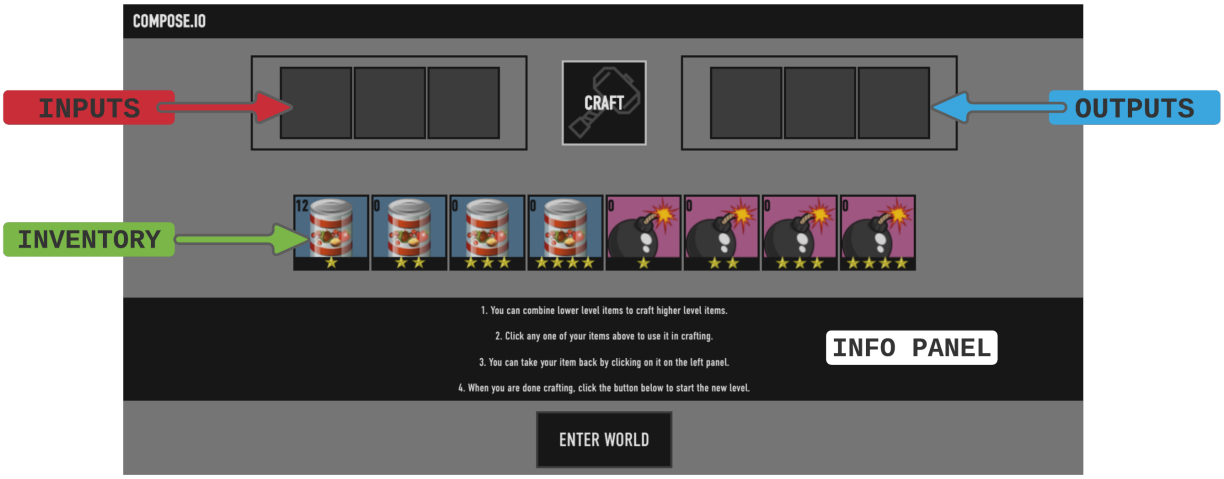

Figure 4.6: Crafting Screen

We keep the crafting screen design as simple as possible, and make sure it reflects the sequence-to-sequence nature of the crafting task as seen in Figure 4.6. Like most games we put the outputs panel to the right of the inputs panel & craft button, aiming to indicate that the crafted resources will be appearing there. We present an information panel that has instructions for people who are not familiar with such systems. Once the player hovers over a resource, the information panel is updated to display information about that specific item. To use a resource in crafting, the player has to click on that resource from the inventory panel and it is placed on the inputs panel. They can take their resource back by clicking it on the inputs panel, and it is then restored to the inventory panel.

Once the player is ready with their sequence, they can click the CRAFT button to initiate crafting. The button calls a function which receives that Day's rules from the configuration file (specified by the experiment designer) and produces appropriate outputs by searching the player's input in that Day's ruleset. Once they outputs are

computed, they are placed on the Outputs panel. The player can click on individual resources to collect them, which are then added to their inventory. If they forget to collect the outputs before making another crafting attempt, we automatically add them to their inventory. We also color code successful and failed attempts. If a crafting attempt is successful, the output panel is highlighted in green for 5 seconds. For Failures we do the same, but in red. Once the player is done with crafting, they can press the ENTER WORLD button at the bottom of the screen to end the crafting phase and start the next survival phase.

4.6 Building online experiments: React, Netlify & Airtable

While building this project, we identified a specific development stack which can be used to develop sophisticated online experiments without having to meddle too much with backend code. Almost all of our development was done using JavaScript & React, which was sufficient on its own to implement all of our game logic, schedule events and update stateful UI components. But at the end of the day we had to host the project as a web application and save the crafting data to a database for further analysis. To solve the hosting problem we used **Netlify**, a hosting platform that enables users to build and host their React applications for free, while also offering serverless backend services. For storage we used **Airtable**, an online spreadsheet-database hybrid with its own REST API. This allowed us to make serverless calls to our spreadsheet-database from within React, creating a new record whenever the player made a crafting attempt or completed a phase successfully.

Since reading and writing to the database is made very simple with this serverless development stack, we also utilize it to balance our experimental groups. We create a separate spreadsheet-database that keeps track of experimental statistics like how many participants completed the game for each experimental group. We read that data as the webpage is loaded, and initialize the game according the configuration of the experimental group with fewer recorded participants. This auto-balancing feature allows us to distribute a single link to the game, rather than building two versions of the game and balancing the groups ourselves. This setup also allows other researchers to design and launch their own experiments without writing any additional code. They just have to create a free Airtable account and enter their API key on Netlify to direct the crafting data to their own spreadsheets. [2]

[2]The spreadsheet-database is available at `https://airtable.com/shrfZouBdVsH8QfQZ`

Chapter 5

Experiment Design & Data Analysis

In this chapter we will be diving into the behavioral dataset we collected by sharing the application link online. Our primary aim is to argue that the environment was engaging and the game was designed well enough to produce useful behavioral data for our cognitive task. We will start by setting an example experiment to test for two specific research hypotheses. We will then do Exploratory Data Analysis on the dataset. We will end the chapter by testing our hypotheses on the data.

5.1 Experiment Setup

The goal of this custom experiment is to study compositional generalization across two rule-systems. The player will first start crafting with some underlying ruleset R_1 and after some time (Days in our game) they will start crafting under another ruleset R_2. This will produce behavioral data for two phenomena: (1) The learning of ruleset R_1 and (2) the learning of ruleset R_2 after learning R_1. For the second experimental group, we will swap the order and introduce ruleset R_2 first and R_1 later. This will produce behavioral data for two additional phenomena: (3) The learning of ruleset R_2 and (4) the learning of ruleset R_1 after learning R_2. Let us name these datasets to make things more concrete:

- **Group-1 Train:** Learning R_1.
- **Group-1 Test:** Learning R_2 conditioned on R_1
- **Group-2 Train:** Learning R_2
- **Group-2 Test:** Learning R_1 conditioned on R_2

5.1.1 Implementing the Experiment

Although the setup is currently too abstract, we can implement it in our framework by assigning these rulesets R_1 & R_2 to distinct resource types: Food & Weapons. So when the player is crafting **Food** resources, they will be crafting under ruleset R_1; and when they are crafting **Weapons**, they will be crafting under ruleset R_2. We can then use the configuration file to simulate the experience of having a **Training phase** on ruleset R_1 and a **Testing phase** on ruleset R_2.

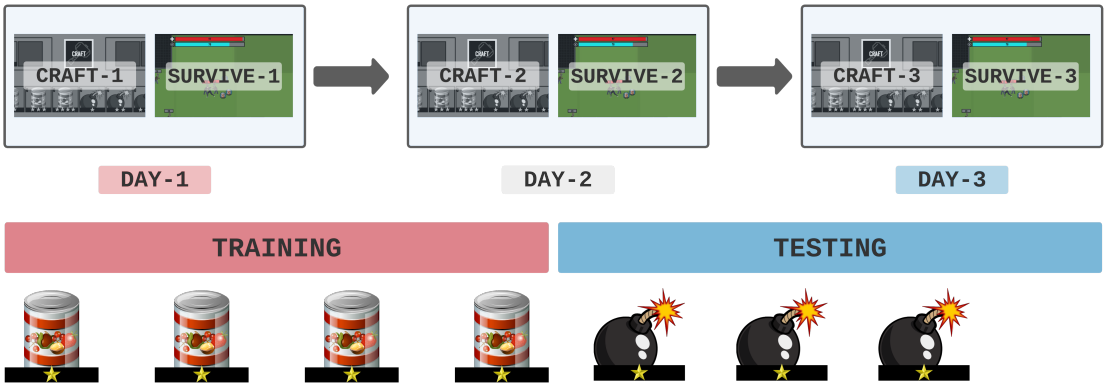

Figure 5.1: Customizing Days to simulate Training & Testing phases

The configuration we used is displayed in Figure 5.1. Our goal is to collect R_1 crafting data during CRAFT-1 & CRAFT-2, and R_2 crafting data on CRAFT-3. We start the player with 10 food resources in their inventory, so they can only craft under ruleset R_1 during CRAFT-1. During SURVIVE-1, we only put food resources on the map for them to gather. In CRAFT-2, since they only gathered food resources, they keep crafting under ruleset R_1. During SURVIVE-2, we place 100 weapon resources on the map so that in CRAFT-3 we will be gathering a lot of weapon crafting data under ruleset R_2 for the first time.

5.1.2 Rulesets

We will now specify the rulesets R_1 and R_2. We already motivated the specific rule-systems we will be working with in 2 when we talked about summation, but here we will name and formalize them. For the first rule R_1 we will use **flexible summation**. Given any number of resources, this rule returns a single resource with a level equal to the sum of the level of the inputs. For the second rule R_2 we will use **strict summation**. This rule does the same summation, but only returns a resource if all inputs have the

same level. Even any one of the items have a different level, it is a failed attempt. For a more visual depiction of these rulesets, we present both rulesets applied to the same resource type 5.2. In case things got complicated, we also provide a summary of the experiment setup in Figure 5.3.

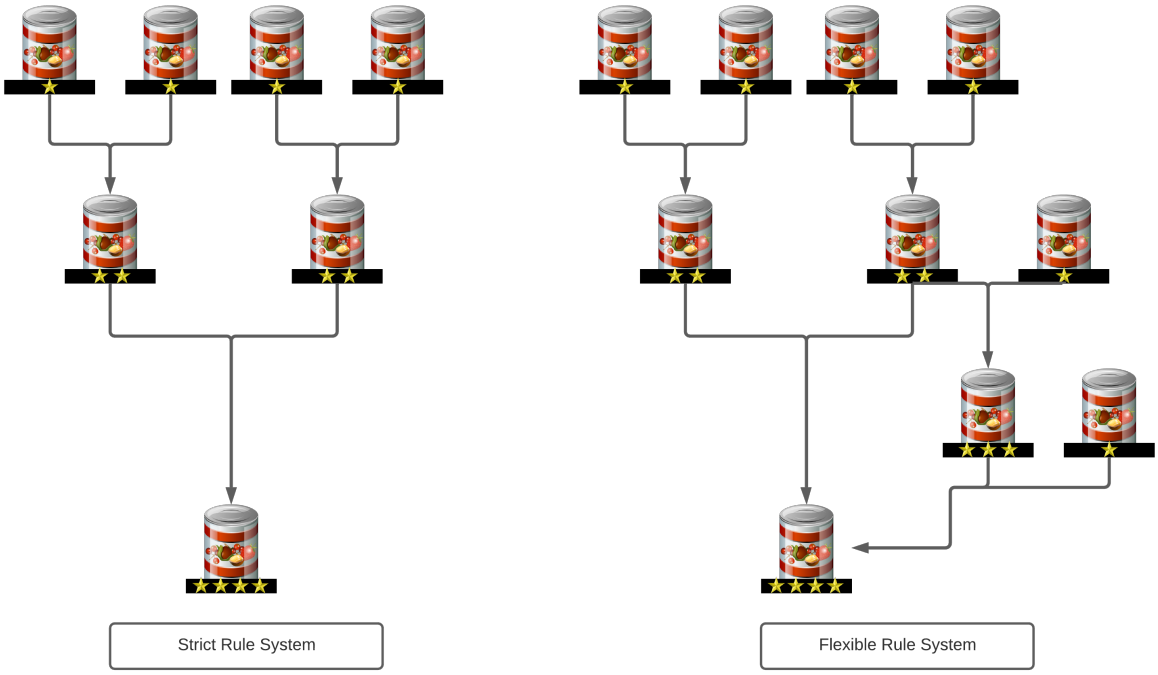

Figure 5.2: Strict Summation (Left) and Flexible Summation (Right)

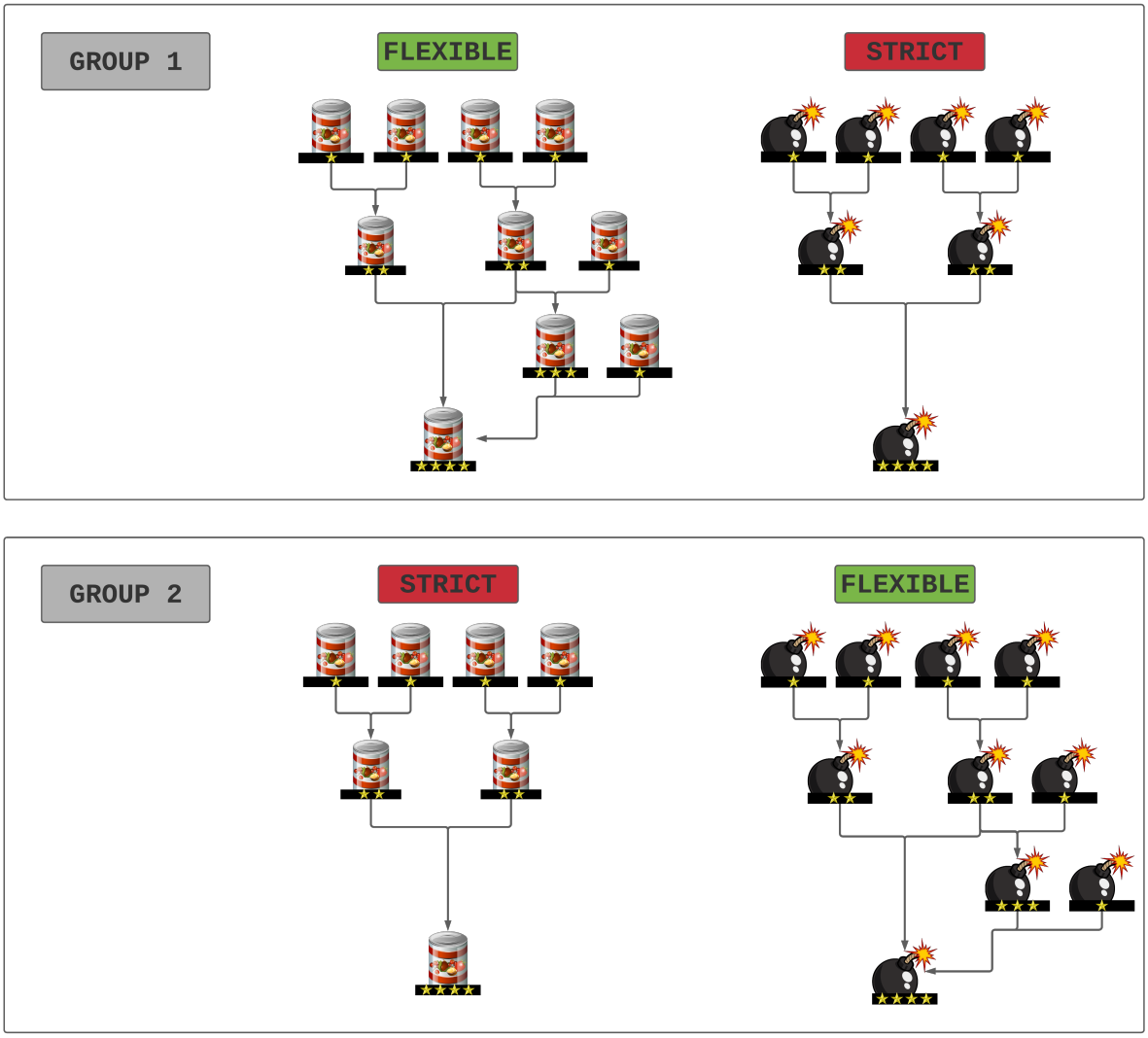

Figure 5.3: Summary: Experiment Setup

5.2 Exploratory Data Analysis

5.2.1 Game Completion

One of the advantages of designing an experiment as a browser game is that it becomes easier to find participants from non-academic communities, by simply advertising the experiment as a *survival game*. The participants were completely anonymous and were mostly recruited through sharing the application's link on various Game and Web Development subreddits on Reddit. We had a total of 482 visitors loading the webpage and 259 of them made at least 1 crafting attempt. 80 of those quitted the game without starting the first survival phase and 179 of them entered the environment. We present a more detailed Sankey Diagram in Figure 5.6 showing the number of participants that made it to each phase. We were also curious about what kind of gameplay decisions were correlated with winning the game. We present a pair-wise Spearman correlation heatmap in Figure 5.5.

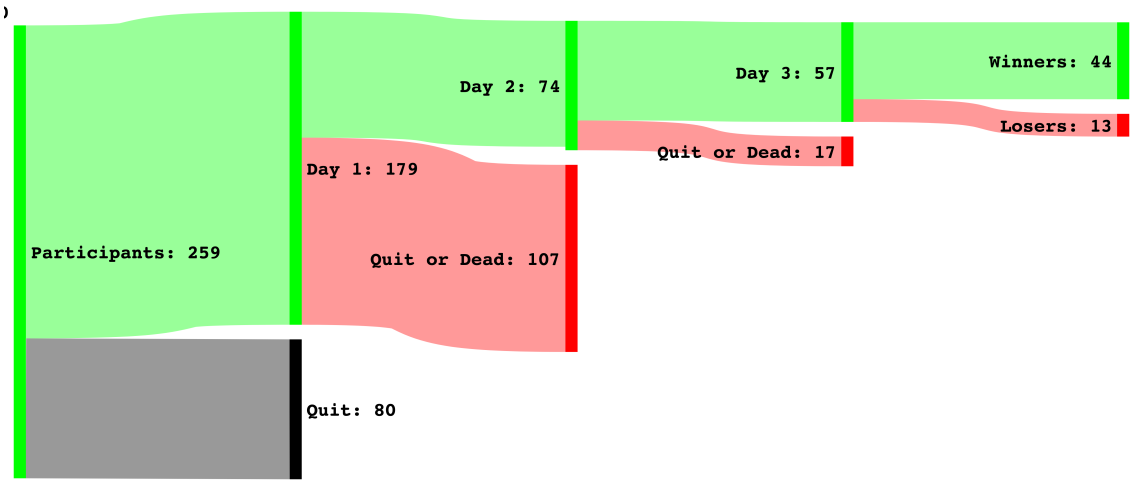

Figure 5.4: Visualizing In-App Participant Journey

5.2.2 Crafting Statistics

Without excluding any participants, we were able to collect 2047 crafting attempts of 59 unique sequences. Only 114 attempts were single-primitive sequences like *food1* or *weapon2*. There were only 6 crafting attempts that tried to combine food items with weapon items, raising further questions about how humans think about heterogeneous compositions. It is also interesting to see that 63% of participants started by composing

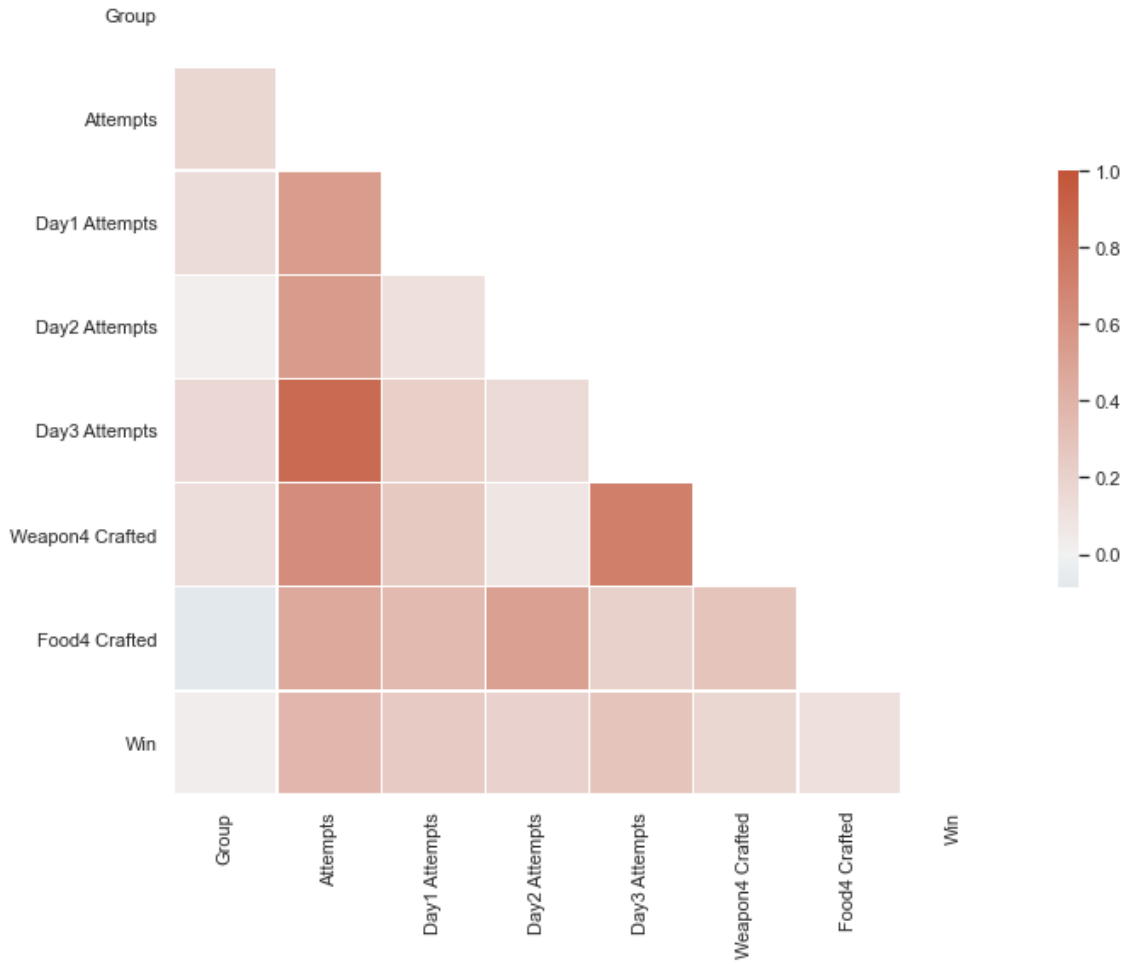

Figure 5.5: Pair-wise Spearman Correlation of Game Decisions

three (Level-1) Food resources, which is the number of input boxes we have made available. In Figure 5.7

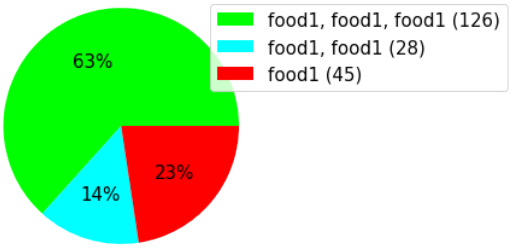

Figure 5.6: First sequence attempted by participants

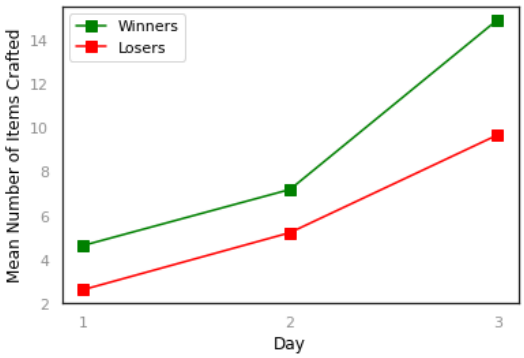

Figure 5.7: Mean number of items crafted across days

5.3 Hypotheses Testing

5.3.1 Generalization from Strict Rules to Flexible Rules

Our first hypothesis is that exposure to a strict system prior to a flexible system causes participants to be more *conservative* when learning the new system. This is because during training their flexible crafting attempts will result in failures, so they will have to stick to strict attempts such that levels match. To test this hypothesis, we use the following datasets:

- **Group-1 Train:** Learning Flexible. (27 Participants, 301 attempts)
- **Group-2 Test:** Learning Flexible, conditioned on Strict. (27 participants, 410 attempts)

To measure *conservativeness*, we simply take the ratio of strict attempts over all attempts. Higher this number, higher the participants stick to strict attempts with equal levels, rather than experimenting with flexible attempts where levels could differ. We compute each participant's conservativeness score, giving us a list of scores for each group. We compute $p = 0.052$.

5.3.2 Generalization from Flexible Rules to Strict Rules

Our second hypotheses is that exposure to a flexible system prior to a strict system will cause participants to have a higher failure rate during testing. We theorize that experimenting with flexible rules first might make it harder for participants to learn a strict ruleset, since they might have had successful attempts where the levels did not match. To test this hypothesis, we use the following datasets:

- **Group-2 Train:** Learning Strict. (27 Participants, 273 attempts)
- **Group-1 Test:** Learning Strict, conditioned on Flexible. (27 participants, 325 attempts)

To measure failure rate, we simply take the ratio of failed attempts over all attempts. We compute each participant's failure rate, giving us a list of failure rates for each group. We compute $p = 0.58$, which indicates strong evidence for the null hypothesis.

Chapter 6

Conclusions

6.1 Crafting

6.2 Future Work

6.2.1 Functions

6.2.2 Exploration & Exploitation

6.2.3 Input and Output Sizes

Testing out different input output sizes / flexible box sizes

Bibliography

[1] Jacob Andreas. Measuring compositionality in representation learning, 2019.

[2] Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: what is required and can it be learned? *arXiv preprint arXiv:1811.12889*, 2018.

[3] Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron C. Courville. Systematic generalization: What is required and can it be learned? *CoRR*, abs/1811.12889, 2018.

[4] Jasmijn Bastings, Marco Baroni, Jason Weston, Kyunghyun Cho, and Douwe Kiela. Jump to better conclusions: Scan both left and right, 2020.

[5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[6] Christopher F. Chabris. Six suggestions for research on games in cognitive science. *Topics in Cognitive Science*, 9(2):497–509, 2017.

[7] Noam Chomsky. Syntactic structures (the hague: Mouton, 1957). *Review of Verbal Behavior by BF Skinner, Language*, 35:26–58, 1957.

[8] Kate Compton and M. Mateas. Casual creators. In *ICCC*, 2015.

[9] Verna Dankers, Elia Bruni, and Dieuwke Hupkes. The paradox of the compositionality of natural language: a neural machine translation case study, 2021.

[10] Jason Fischer, John G. Mikhael, Joshua B. Tenenbaum, and Nancy Kanwisher. Functional neuroanatomy of intuitive physical inference. 113(34):E5072–E5081, 2016.

[11] J. Fodor and Z. Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28:3–71, 1988.

[12] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018.

[13] April Grow, Melanie Dickinson, Johnathan Pagnutti, Noah Wardrip-Fruin, and Michael Mateas. Crafting in games. *Digital Humanities*, 11(4), 2017.

[14] Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: how do neural networks generalise? *Journal of Artificial Intelligence Research*, 67:757–795, 2020.

[15] Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: how do neural networks generalise?, 2020.

[16] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C. Lawrence Zitnick, and Ross B. Girshick. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. *CoRR*, abs/1612.06890, 2016.

[17] Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. *CoRR*, abs/1912.09713, 2019.

[18] Blitworks Klei Entertainment. Don't starve. [Android, PlayStation 4, Xbox One, Nintendo Switch], 2013.

[19] Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks, 2018.

[20] Brenden M. Lake, Tal Linzen, and Marco Baroni. Human few-shot learning of compositional instructions, 2019.

[21] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people, 2016.

[22] 4J Studios Telltale Games Double Eleven Other Ocean Interactive Mojang Studios, Xbox Game Studios. Minecraft, 2011.

[23] Wei Peng, Jih-Hsuan Lin, Karin A. Pfeiffer, and Brian Winn. Need satisfaction supportive game features as motivational determinants: An experimental study of a self-determination theory guided exergame. *Media Psychology*, 15(2):175–196, 2012.

[24] Steven Piantadosi, Joshua Tenenbaum, and Noah Goodman. The logical primitives of thought: Empirical foundations for compositional cognitive models. *Psychological review*, 123, 04 2016.

[25] Scott Rigby and Richard M Ryan. *Glued to games: How video games draw us in and hold us spellbound: How video games draw us in and hold us spellbound*. AbC-CLIo, 2011.

[26] Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M. Lake. A benchmark for systematic generalization in grounded language understanding, 2020.

[27] Michael Sailer, Jan Ulrich Hense, Sarah Katharina Mayr, and Heinz Mandl. How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction. *Computers in Human Behavior*, 69:371–380, 2017.

[28] Kexin Yi*, Chuang Gan*, Yunzhu Li, Pushmeet Kohli, Jiajun Wu, Antonio Torralba, and Joshua B. Tenenbaum. Clevrer: Collision events for video representation and reasoning. In *International Conference on Learning Representations*, 2020.