# Cognitive Modelling - Complex Behavior

## *The 'Finger Spoof' app*


Tom Veldhuis
s3451771

David ten Kate
s5713323

Thijs Lukkien
s3978389


April 2024

# Contents

The code for this iPhone app can be found at here. If there are any problems with accessing the code, please contact Tom by email.

# 1 Introduction

## 1.1 Our goal

The goal of this project is to create an app that is based on the party game "Finger Spoof". It must allow human players to play against AI opponents. These AI opponents should be designed to approximate human cognition as such that their game behavior is similar to that of a human player.

## 1.2 Game mechanics

The game of "Finger Spoof' is relatively simple; players lay their fingers on the edge of a cup and take turns in a clockwise or counter-clockwise fashion. When it is your turn, you count down from 3 to 0. At the count of zero, two things happen: the first thing is that every player decides to stay on the cup or pull away their finger. The second thing is that the "turn-player" shouts a number, predicting how many fingers remain on the cup. If the prediction is correct, the "turn-player" gains a point. Now it is the next player's turn. Figure 1 shows an example of one round.


Figure 1: The first two turns of an example round. Here, nobody scores a point. These four frames are an extract from the game's tutorial gif.

# 2 Interface design

In this section, we aim to give a general overview of our design principles, choices, and resulting views.

## 2.1 Style

For the interface, we needed a style that matched best with our type of game. For this, it needed to fit a number of requirements. Firstly, the interface must not become too cluttered as there could be as many as 10 people playing the game simultaneously. It must be simple, as gameplay can be hectic enough by itself. It must draw the players' attention to the functional game components, for the same reason. Lastly, we want it to be both visually appealing and joyful as Finger Spoof is originally a party game. The style best matching the first requirement is the minimalist style, though it may need some adjustments to fit the last requirements. Minimalism comes with the added benefit of being easily adaptable from early development stages as it requires little extra artwork.

## 2.2 Design principles

It can be difficult to articulate what principles constitute a specific style. For this reason, we consulted numerous blog posts that analyzed hundreds of minimalist web designs. We then distilled their design principles into the three main rules of our design. A concatenation of design principles given by some much-read blog posts [1, 2, 3] can be found in Table 1.

| N | Principle | Count |
|---|---|---|
| 1 | Flat patterns & features | 2 |
| 2 | Grid blocks and layouts | 2 |
| 3 | Simple (monochromatic) palette | 3 |
| 4 | Negative space | 3 |
| 5 | Attention attractors | 3 |
| 6 | Core functionality | 2 |
| 7 | Large and few graphics | 1 |
| 8 | Simple navigation | 2 |

Table 1: Collection of minimalist design principles

From these 8 principles, we focus mainly on 3, 4, and 5. Principles 2 and 7 are irrelevant to our project, and principles 1, 6, and 8 are almost automatically adhered to due to the limitations of SwiftUI design. Principle 3, usage of a simple (monochromatic) palette, implies that one should limit themselves to shades of black and white, and to one shade of colors. In this project, we mainly use shades of orange, black, white and a dark shade of blue as an alternative to black. Principle 4, negative space, refers to padding every object with as much space as functionally possible. This works best when the objects are designed in accordance with principle 5. Principle 5, attention attractors, says that functionally-important elements should be 'loud' and 'bold'. Later, we will show how these principles were integrated into our app design.

## 2.3 Views

### 2.3.1 Landing view

When launching the app, the user arrives on the landing page. Here, the first implementation of our design principles can be seen. The design is limited to what is important to us: the users need to recognize the game, adjust its main (2) settings, and play the game. We wanted to draw attention to the title first, then to the settings, and lastly to the play button. If a player is confused about how the game works, then we assume that the player will look further and find the less attraction-attracting tutorial link. However, we did not want to emphasize this as it is not a core functionality.

It can be seen that the number of human players can be increased, but not decreased. This is for the simple reason that we do not allow a bot-only game. A minus button appears when the number of human players is set to a value larger than 1. For similar reasons, the number of AI players cannot be set to 0. The total number of allowed players is set to 10, at which point all plus buttons will disappear. Clicking the link 'How to use' redirects players to the tutorial view.

### 2.3.2 Tutorial view

This is a scrollable view, the main purpose of which is to give players a rough idea of how the main game mechanics work. The core functionalities are; an explanation gif, the goal of the game, and a set of explicit instructions. For navigation purposes, we set an obvious title at the top in the same style as the landing page. Pressing the cross on the top left will return the player to the landing view.

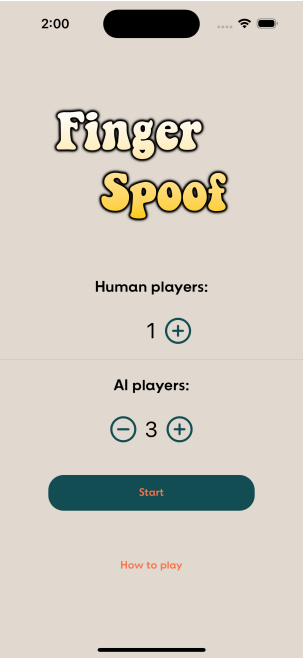


Figure 2: The landing view.

### 2.3.3 Game views

When the number of human and AI players has been decided, and the player presses 'play' on the landing view, the game starts. That is the moment this view is shown. The core of this page is the so-called 'spoof circle'. This is a minimalist representation of the cup in the original game. On this circle, the player buttons are laid. There is one button per person, indicated by their ID. 'H1' stands for 'Human player one' and similarly 'B2' stands for 'Bot player two'. Each player is bound to their button the whole game to allow scorekeeping. In Figures 4 and 5 it can be seen how player buttons are automatically generated to lay spread evenly on the spoof circle.

The white circle indicates whose predicting turn it is. The fill color of the buttons can be black, empty, or dark blue. Before the round starts, players are asked to place their fingers on the cup, accompanied by a black fill color of the buttons. This can be seen in Figure 4 Only when all players have placed their fingers does the round start.

When the round begins, the 'turn player' is shown a number of buttons, allowing them to enter a prediction. Once this has been entered, a countdown timer will appear in the center of the spoof circle.

After the countdown, the result view is shown, as can be seen in Figure 5. Here, the decision of each player can be seen by the design of their buttons. If a player has 'pulled', then their circle will be empty. If a player has 'stayed', then their circle will be full. The results of this prediction can be seen in the middle of the screen and the overall game results can be seen in the place of each player ID.

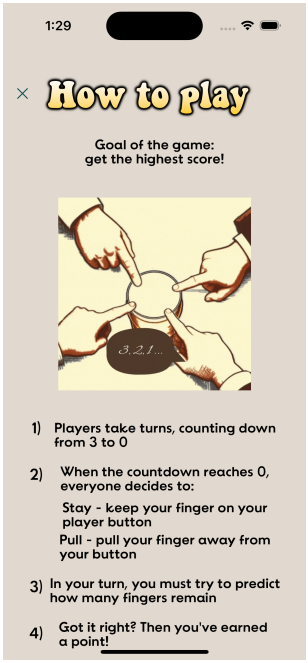


Figure 3: Tutorial view. A user can scroll down slightly to better be able to read the instructions. The picture in the center is one of the eight frames from a GIF. More frames can be seen in Figure 1.

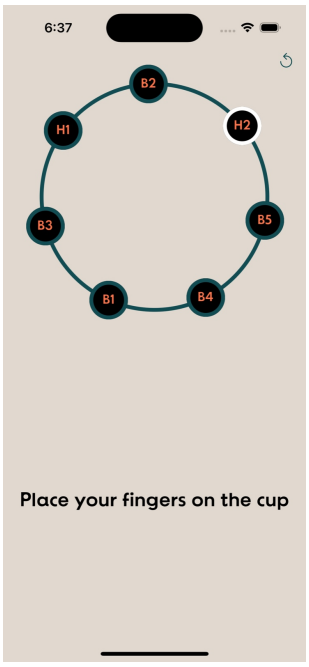


Figure 4: Waiting for the next turn to start.

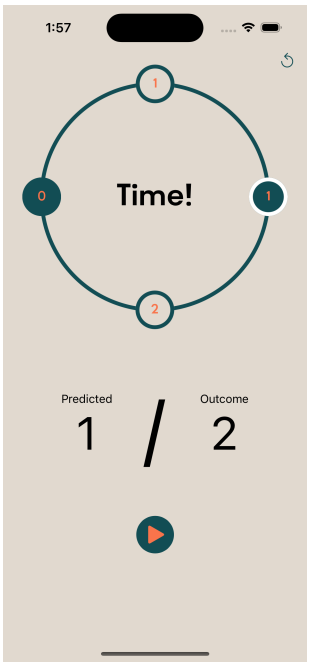


Figure 5: Result of a turn. The 'turn-player' did not score a point.


Figure 6: Option to quit the game after pressing the button in the top-right.

We found this way of showing the scores more intuitive as a text representation would be difficult to associate with the actual players. On the other hand, the button positions are fixed from the moment the game starts. This allows players to easily associate scores with player actions, as everything is graphically represented in the spoof circle.

Once again, let us discuss how the three main design principles are implemented in these views. Firstly, we want the spoof circle and the play button to be attention attractors, as they constitute the core features. Secondly, we aim to give more attention to each individual component by surrounding them with negative (or empty) space. Lastly, we rely on the same calm color palette to draw attention or allow attention to be drawn away.

The game can be quit at any time via the arrow icon at the top-right of the screen, as can be seen in Figure 6.

# 3 Programming

We used Swift to implement our application for iOS devices. The full source code can be found on GitHub[1].

## 3.1 Structural overview

- `FingersApp`: The `FingersApp` class is responsible for defining the scene. It handles which view to render by controlling the flow and is responsible for instantiating the `FingersViewModel` class, which is passed on as an argument to the `FingersView` class.
- `FingersModel`: The model class is part of the MVVM paradigm, and completely acts independently from the user interface. It is subscribed to the `FingersViewModel` class to notify when changes from the user interface are registered, like user input. Furthermore, the class represents simple data and handles the game logic.
- `FingersViewModel`: The ViewModel class is also part of the MVVM paradigm, where it binds data from the View to the Model. Another task of the `FingersViewModel` class is to prepare data for visual presentation.
- `FingersView`: The final class in the MVVM paradigm is the View class. It is stateless and dynamic so that it can execute its main task of visualizing the user interface and recording user input, which it sends to the `FingersViewModel` class.
- `LandingView`: The `LandingView` is the View that the application opens on, and contains the logic to configure and start a round of Finger Spoof, the tutorial, and the credits.
- `PredictView`: This view is responsible for rendering the grid of available numbers that you can choose to make your prediction when it is a human player's turn, or to visualize that a bot is making a decision when it is a bot's turn.
- `Player`: The class includes the protocol `Player`, which `Human` and `Bot` inherit from and in which they define their own prediction and decision-making logic. Additionally, the `Bot` class defines which `BotModel` they use.
- `BotModelProtocol`: A template to create a `BotModel` class with. The protocol has an extension to define an `update()` function as every `BotModel` requires this function to trigger an update in `FingersView`.

[1] at `https://github.com/tomveldhuis/CogMod_Fingers/`

- `BotModel`: The model class that human players play against which incorporates functions from the Swift ACT-R library written by Prof. Dr. N.A. Taatgen[2]. Multiple implementations of the `BotModel` class are possible as it inherits from the `BotModelProtocol` class.

### 3.2 Design choices

Throughout the project, we have created two protocol classes, `Player` and `BotProtocol`. We chose a protocol-oriented approach rather than an object-oriented approach, as we wanted a blueprint for the objects that would then be created from these protocols to ensure an abstract definition of the class. This gave a clear purpose to the classes that inherited a protocol and allowed for the reusability of properties and methods that were shared between various implementations of said protocols.

## 4 Cognitive model

Each AI opponent within the game uses a cognitive model, based on the ACT-R cognitive architecture [4], to make decisions as they play the game. The cognitive model has two goals:

1. During each turn of the game, a **decision** has to be made about whether an AI opponent will stay or pull their finger from the cup.
2. When an AI opponent is the current player, they also have to make a **prediction** about the number of fingers that will stay on the cup that turn.

In the next subsections, both the inspiration and design of the cognitive model that accomplishes these goals are discussed.

### 4.1 Model inspiration

The main question that needs answering before starting to design the cognitive model, is how the aforementioned decisions and predictions are made in the real world as that will inform us how the cognitive model should be designed.

#### 4.1.1 Idiot spoof

In the original game, there is a concept called 'idiot spoof'. This happens when someone predicts an impossible outcome, made impossible by that person self. For example, person A predicts 0 but is also still on the cup. This tells us that one may keep the processes of deciding and predicting separate.

#### 4.1.2 Decisions

There are several ways to look at how decisions (deciding to stay/pull your finger on the cup) are being made. Given that the game of Fingerspoof is a fast-paced game, it would make sense that specific simple tactics based on decisions of players are to be generally employed by people. Examples of these tactics can be: choosing the most or least occurring decision during the game or mimicking a specific player. An ACT-R model could implement this by saving the decisions of players in declarative memory during each turn. These players could be the set of decisions of an individual player (in the case of mimicking a player) or the set of decisions of all players.

Note that not only decisions of players can influence a player's decision, but also the predictions of players and the outcome of turns. A player wants to win by having the most points and concurrently

[2]can be found on GitHub at `https://github.com/ntaatgen/ACT-R-SU`

trying to let other players not get any points. For this reason, it makes sense that a player's decision should aim to influence the outcome of a turn such that it is not the same as another player's prediction (as that would give that other player a point). Similarly, a player's decision should also aim to influence the outcome of a turn such that it is in line with their own prediction (such that they would win a point). An ACT-R model could implement this line of thinking by looking for specific patterns in previous decisions and predictions and adjusting a player's decision accordingly, since knowing which decisions and predictions are likely to occur could inform the AI player about which decision would benefit them.

#### 4.1.3 Predictions

When an AI opponent makes a prediction, their aim is to get have it be the same as the outcome of the turn (the exact numbers of fingers on the cup after the countdown). If their prediction is not exactly the same as the outcome, they will not gain a point. Thus, it would makes sense for players to look at the results of the turns so far to inform their current prediction. An ACT-R model could implement this by saving the outcome of previous turns in declarative memory.

### 4.2 Model design

Two types of cognitive models are currently used by the AI opponents in the game: the main model and the extended model. A few general things should be noted about the usage and workings of these models:

- At initialization, one of these two models is randomly picked. The idea behind this is to introduce variety in the way AI players can approach their decision- and prediction making.
- After a turn has ended, the model time is increased by 5 seconds to account for the time between turns.
- When a decision or prediction has been made, it is automatically added to the imaginal buffer.

#### 4.2.1 Chunk design

| | |
|---|---|
| isa | `goal` |
| state | "deciding" / "predicting" / "waiting" / "deciding_extreme" / "deciding_personal" |
| numPlayers | $i \in \mathbb{N}$ |
| breakTendency | $j \in [0, 1]$ |
| isActive | "yes" / "no" |

Table 2: Goal chunk used in the main and extended models

Table 2 shows the contents of the goal chunk. The information that is contained within the goal chunk, is assumed to be (implicitly) known throughout the game. The following should be noted regarding the properties within the goal chunk:

- 'numPlayers' is a natural number representing the total number of players within the current game
- 'breakTendency' is a floating number (between 0 and 1) representing an AI player's willingness to break from observed patterns throughout the game (will be further elaborated upon in section 4.2.3).

- 'isActive' is set to "yes" if the AI player is expected to make a prediction during the current turn (otherwise it is set to "no").

| isa | `lastDecision` | isa | `lastPrediction` | isa | `lastResult` | isa | `myDecision` |
|---|---|---|---|---|---|---|---|
| decision | "stay" / "pull" | prediction | $i \in \mathbb{N}$ | result | $i \in \mathbb{N}$ | decision | $i \in [0, 1]$ |
| | | result | $j \in \mathbb{N}$ | | | | |
| | | win | "yes" / "no" | | | | |

Table 3: Chunks used in the main and extended models

Table 3 shows the 4 different chunk types that are used within the model:

- `lastDecision`: Used to track the overall decisions of all players within the game (due to it being able to merge with identical chunks). Only used in the main model.
- `lastPrediction`: Used to track the outcome and prediction of a turn. 'prediction' and 'result' are both a natural number between 0 and the total number of players. When 'prediction' and 'result' have the same value, the value of 'win' is set to 'yes', otherwise 'no'.
- `lastResult`: Used to track patterns of specific outcomes throughout the game (due to it being able to merge with identical chunks). Only used in the extended model.
- `myDecision`: Used to track likelihood of the AI player staying ('decision' is equal to 1) or pulling ('decision' is equal to 0), based on previous experiences. Only used in the extended model.

### 4.2.2 Main model

Figure 7 show the process of the main model. During each turn of the game, the model goes through 2 or 3 states (depending on whether the AI player needs to make a prediction), starting at the 'Deciding' state:

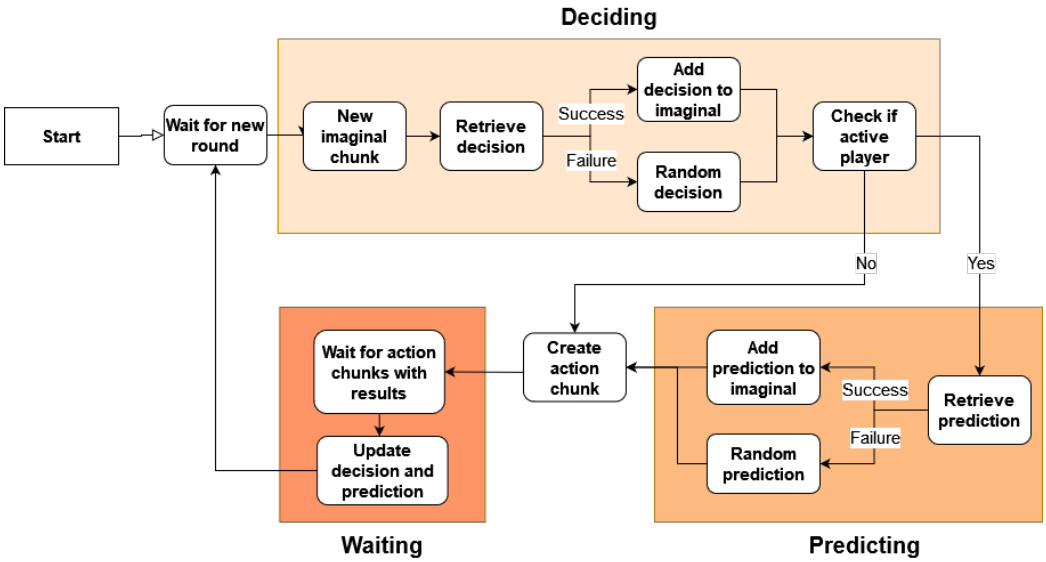


Figure 7: Flowchart of the main ACT-R model

1. **Deciding**
   (a) **New imaginal chunk**: A new chunk is made and set to the imaginal buffer, such that decisions (and predictions) can be stored and retrieved later during the turn if needed.
   (b) **Retrieve decision**: Regular retrieval is being used to retrieve a `lastDecision` chunk from declarative memory.
   (c) **Add decision to imaginal**: Given that a `lastDecision` chunk was retrieved, the value for 'decision' is added to the imaginal buffer.
   (d) **Random decision**: Given that no `lastDecision` chunk could be retrieved, a random value for 'decision' is added to the imaginal buffer using the `actrNoise` function (from the Swift ACT-R library).
   (e) **Check if active player**: If the value for 'isActive' in the goal chunk is set to "yes", the state changes to the 'Predicting' state. If it is set to "no", an action chunk is being made instead.
2. **Predicting**
   (a) **Retrieve prediction**: Blended partial retrieval is being used to retrieve a `lastPrediction` chunk from declarative memory. A mismatch penalty of -1 is added to encountered `lastPrediction` chunks, if they have their slot "win" set to "no".
   (b) **Add prediction to imaginal**: Given that a `lastPrediction` chunk was retrieved, the blended value for 'prediction' is added to the imaginal buffer.
   (c) **Random decision**: Given that no `lastPrediction` chunk could be retrieved, a random value for 'prediction' in the range between 0 and the 'numPlayers' value (from the goal chunk) is added to the imaginal buffer using the `Int.random()` function (from the Swift standard library).
3. **Waiting**
   (a) **Create action chunk**: An action chunk containing a decision (and a prediction if the AI player is the current player) is made and sent to the game code outside of the ACT-R model. After the action chunk has been sent, the model will stop running until the action chunk is updated by the game code outside of the ACT-R model (while the state also changes to the "Waiting" state).
   (b) **Wait for action chunks with results**: After the countdown within a turn has ended, information about the result of the turn is sent to the model: specifically the "result" (the total number of fingers on the cup at the end of the countdown) and the "currentPrediction" (the prediction made by the current player). This triggers the model to start running again.
   (c) **Update decision and prediction**: Using the information from the action chunk, the declarative memory is updated as follows:
      - Assuming $i$ is the "result" value and $j$ is the "numPlayers" value, $j$ new `lastDecision` chunks are added to declarative memory with $i$ of those chunks having the value of "decision" set to "stay" (and the others set to "pull").
      - A new `lastPrediction` chunk is added to declarative memory using the information from the action chunk. If "result" and "currentPrediction" have the same value, the value for "win" is set to "yes" (otherwise it is set to "no").
      - The contents of the imaginal buffer is also added to declarative memory.

   After the declarative memory has been updated, the model stops running again until the next turn is started and the model is asked for giving a decision (and a prediction) again.

#### 4.2.3 Extended model

The extended model introduces a few additional concepts to the main model, with the goal of specifically enhancing the process for making decisions in terms of cognitive plausibility:

- When an AI player is initialized, a property called 'breakTendency' is set as well. This property describes the likelihood of an AI player to 'break' through patterns if it encounters one during its decision process. The value for 'breakTendency' is sampled from a Gaussian distribution in the range $[0, 0.5]$ with 0.5 being randomly added, such that a sampled 'breakTendency' value can center aturn either 0.25 or 0.75.

- The extended model tries to observe whether 'extreme' results (found through `lastResult` chunks) have been taking over the outcomes of recent turns. An 'extreme' results indicates that either 0 or the total number of players has been the outcome of the recent turns. The idea here is that knowing that 'extreme' results are occurring would make it easier for all players to make correct predictions. And since one of the goals of the game is to ensure your rival players are not making correct predictions, countering such an 'extreme' result by doing the opposite could work in your favor.

- The extended model also tries to observe whether personal patterns (found through `myDecision` chunks) are occurring in the AI player's own decision making. A personal pattern can occurs when the AI player has been deciding to either "stay" or "pull" more often in recent turns. The idea here is players should be able to introspect whether their most recent decisions have become predictable (which can lead to easier predictions and consequently rival players gaining points more easily) and diverge from their predictable pattern by doing the opposite instead.

Figure 8 show the process of the extended model. During each turn of the game, the model goes through either 2, 3 or 4 states, starting at the 'Deciding (Extremes)' state:

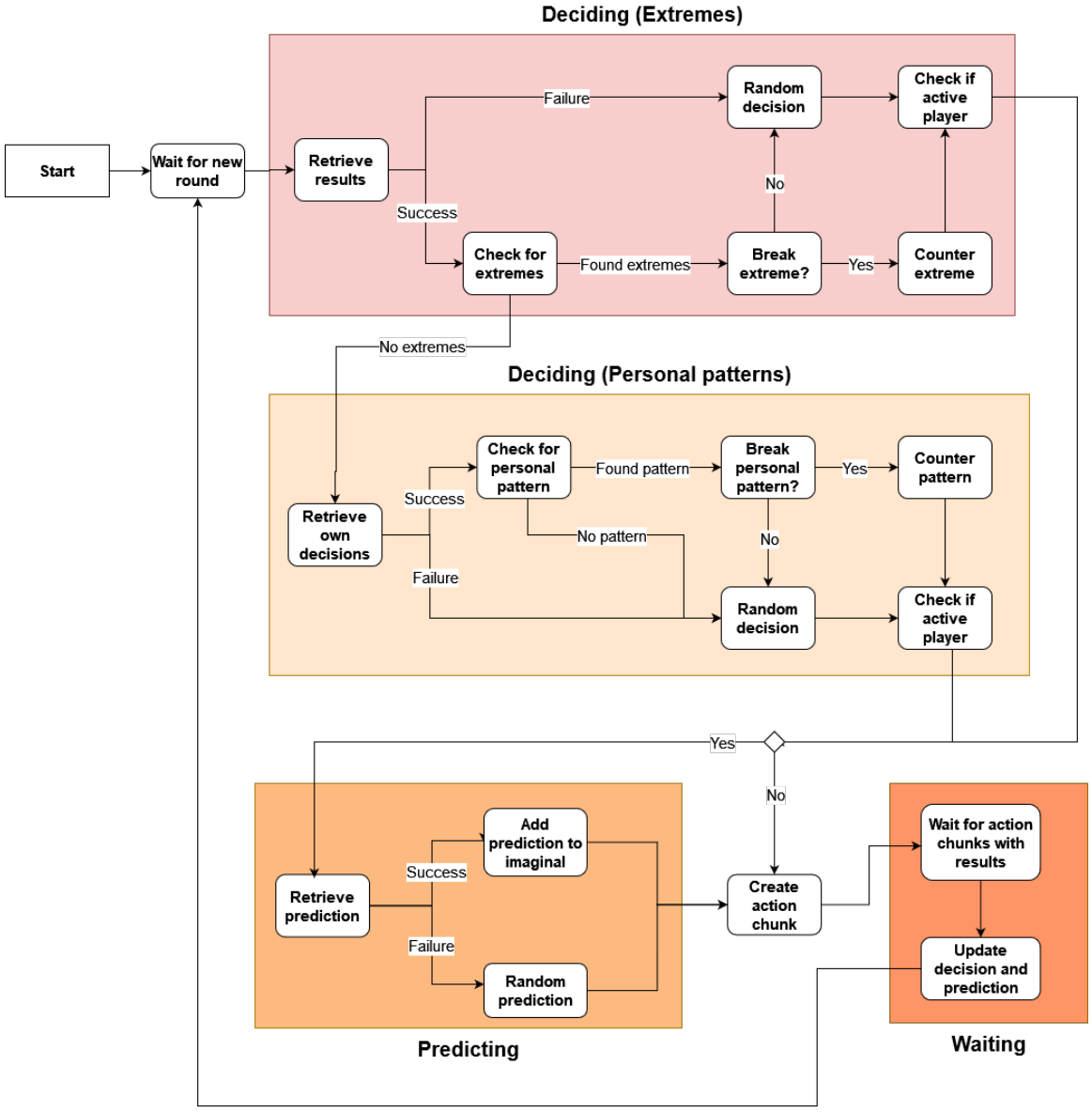


Figure 8: Flowchart of the extended ACT-R model

1. **Deciding (Extremes)**

   (a) **Retrieve results**: Firstly, a new chunk is made and set to the imaginal buffer, such that decisions (and predictions) can be stored and retrieved later during the turn if needed. Then, regular retrieval is used to retrieve a `lastResult` chunk from declarative memory.

   (b) **Check for extremes**: Given that a `lastResult` chunk was retrieved, it is checked whether the retrieved value of 'result' is either equal to 0 or the value of 'numPlayers' such to indicate whether an 'extreme' result has been recently observed.

   (c) **Break extreme?**: Given that an 'extreme' result has been observed, the value of 'breakTendency' is used as the probability of deciding to 'break' through this 'extreme' result.

The function `Double.random()` (from the Swift standard library) is used to generate randomness.

(d) **Counter extreme**: Given that the AI player has decided to 'break' through the 'extreme' result, a counter value of the 'extreme' result is added as the value for 'decision' to the imaginal buffer: if the 'extreme' result is equal to 0, the value for 'decision' will be "stay", whereas if it's equal to the value of "numPlayers", the value for decision will be "pull".

(e) **Random decision**: Given that either no `lastResult` was retrieved or the AI player has decided not to 'break' through the 'extreme' result, a random value for 'decision' is added to the imaginal buffer using the `actrNoise` function (from the Swift ACT-R library).

(f) **Check if active player**: (identical to the same routine in 4.2.2)

2. **Deciding (Personal patterns)**

(a) **Retrieve own decisions**: Given that no 'extreme' result has been observed, the state changes to the 'Deciding (Personal patterns)' state. Then, blended retrieval is being used to to retrieve a `myDecision` chunk from declarative memory.

(b) **Check for personal pattern**: Given that a `myDecision` chunk was retrieved, it is checked whether the retrieved blended value of 'decision' is either lower than 0.25 (i.e. a "pull" pattern) or higher than 0.75 (i.e. a "stay" pattern) such to indicate whether an personal pattern has been recently observed.

(c) **Break personal pattern?**: Given that a personal pattern has been observed, the value of 'breakTendency' is used as the probability of deciding to 'break' through this personal pattern. The function `Double.random()` (from the Swift standard library) is used to generate randomness.

(d) **Counter pattern**: Given that the AI player has decided to 'break' through the personal pattern, a counter value of the personal pattern is added as the value for 'decision' to the imaginal buffer: if the retrieved blended value of 'decision' is lower than 0.25, the value for 'decision' in the imaginal buffer will be "stay", whereas if it's higher than 0.75, the value for 'decision' in the imaginal buffer will be "pull".

(e) **Random decision**: Given that either no `myDecision` was retrieved or the AI player has decided not to 'break' through the personal pattern, a random value for 'decision' is added to the imaginal buffer using the `actrNoise` function (from the Swift ACT-R library).

(f) **Check if active player**: (identical to the same routine in 4.2.2)

3. **Predicting**: (identical to the same routine in 4.2.2)

4. **Waiting**

(a) **Create action chunk**: (identical to the same routine in 4.2.2)

(b) **Wait for action chunks with results**: (identical to the same routine in 4.2.2)

(c) **Update decision and prediction**: Using the information from the action chunk, the declarative memory is updated as follows:

- A new `lastResult` chunk is added to declarative memory using the 'result' value from the action chunk.
- A new `myDecision` chunk is added to declarative memory using the 'decision' value from the imaginal buffer. If the value of 'decision' in the imaginal buffer is equal to "stay", the value of 'decision' in the `myDecision` chunk becomes 1, whereas if it's equal to "pull", the value of 'decision' in the `myDecision` chunk becomes 0.

- A new `lastPrediction` chunk is added to declarative memory using the information from the action chunk. If "result" and "currentPrediction" have the same value, the value for "win" is set to "yes" (otherwise it is set to "no").
- The contents of the imaginal buffer is also added to declarative memory.

After the declarative memory has been updated, the model stops running again until the next turn is started and the model is asked to give a decision (and a prediction) again.

# 5 Evaluation

In this section, we reflect on the project, our goals, and the overall model performance.

## 5.1 Model performance

In this subsection, we want to have a look at how the final model performs. We will try to give a qualitative evaluation of how the model performs in a 1-human-3-bots game. To do this, we recorded 16 rounds (R) equaling 64 turns against the extended model. The results can be found in Figure 9

We can make a multitude of inferences from Figure 9. Before we start with those inferences, we would like to place a big asterisk on all of the conclusions drawn from these results as our sample size (64 turns) is still very small and the results could differ in other games. Moreover, to get an accurate view of Bot behavior, it would be desirable to have an equal number of bot and human players in the game. However, due to simulation constraints, this was not possible in an environment with more than two players. We specifically wanted to evaluate an environment with more than two players, as there is little analyzable group dynamic in a duo.

Firstly, it is not easy to tell which player is human (P2), and which is a cognitive model. Mainly in the first 8 rounds can one tell which player is the human one (it is P2). This gives an indication that the model behaves human-like, which is one of the goals we aimed to achieve. A future improvement would be incorporating more human behavior for comparison in the analysis. Secondly, we can see that idiot spoof still occurs for bot players. While resulting in poor performance, it is important that this is still possible (and happening) as the phenomenon occurs in human games as well. Thirdly, we can see that idiot spoof occurs less as the rounds proceed. This seems to indicate that the bots are learning new behavior, although this needs to be further looked into with more data. Fourthly, we can see rare (extreme) occurrences triggering extreme prediction behavior. Which can be predicted correctly (see R7, turn 2) or incorrectly (see R7, turn 3). This occurs often in human-only games, and this behavior seems to be validated by the model.

It must be noted that the bot models have no game knowledge besides their tactics, and so we would argue that the bot models will start behaving more realistically as they gain memories. We can see this happening from the decreasing number of idiot spoofs. However, we only have a limited sample size to evaluate.

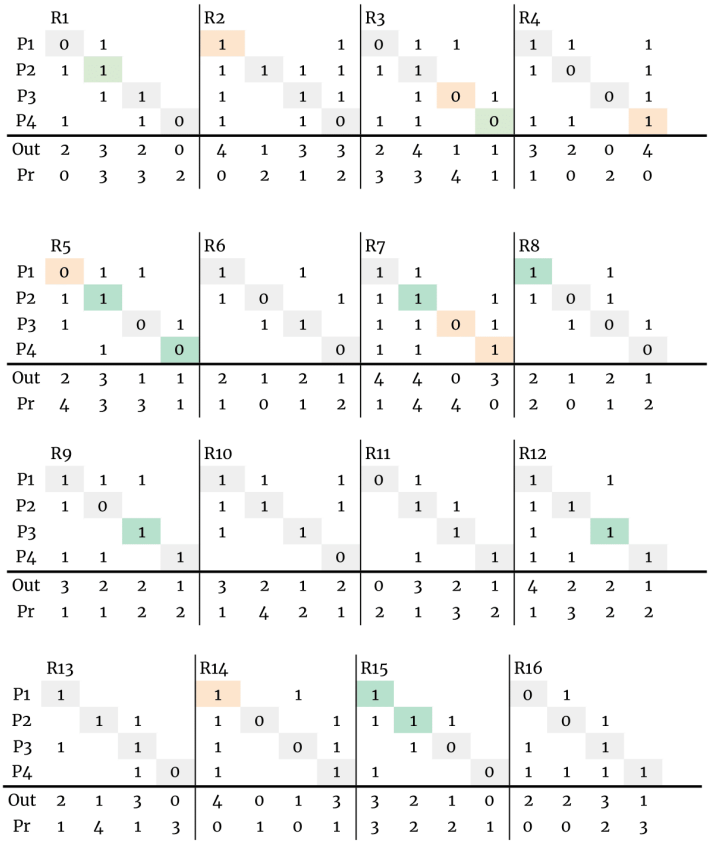


Figure 9: The results of 16 rounds (= 64 turns). 'Px' refers to 'Player x'; can you guess who the human player is? A one indicates a player staying that turn, and a zero (or empty cell, for readability) indicates a player pulling that turn. The grey diagonal indicates which player's turn it is. The color can be overlaid with green or orange. Green indicates a correct prediction, orange indicates an idiot spoof. 'Out' refers to the outcome (sum) of all decisions. 'Pr' refers to the prediction made by the player that turn.

## 5.2 Goal reflection

For this project, we had 4 main objectives:

- Our app must allow human players to play against AI opponents.
- These AI opponents should be designed to approximate human cognition
- as such that their game behavior is similar to that of a human player.
- The app must be intuitive to use and visually appealing.

The first objective was a baseline; by the production of this app, this objective has been achieved. An important part of this is that the app (game) runs without bugs or crashes throughout extensive testing.

The second objective has been achieved by implementing decision-making methods based on ACT-R declarative memory in our AI players. All of this is done using substantiated methods that

simulate memory functioning in humans. Still, we acknowledge that our models are based on several assumptions (see Section 4.1) about the decision-making methods of a human, and on our intuitions.

The third objective is difficult to evaluate. In the previous section, we discussed the problems with defining a 'human-like' behavior. For this reason, we can only say that the human player behavior is difficult to discern from the behavior of the bots. This seems to indicate human-like behavior from the bots, hence we would say that this objective has been achieved (albeit with some asterisks).

The fourth objective is very subjective. We feel as though our design is substantiated and thought-through. Whether this objective has been achieved is up to the personal opinion of a player.

### 5.3 App interaction

During bug testing, we encountered some minor glitches, all of which have been solved now.

There are two remaining issues, both of which are caused by the limitations of simulating a virtual device. We have found workarounds for these problems, as well as theoretical solutions for a real-life version of the game. Still, the workarounds do make the game feel of lower quality than the real-life version would have been.

Firstly, in the original, physical version of Finger Spoof, the predicting player also decides simultaneously. In our version of the game, we wanted to recreate this. However, this implementation came with a practical difficulty. As we are playing the game on a mobile phone, players would physically have to move around one another to access the predicting view. In a simulation, this is solved as multiple player cursors can move around one another. Still, we created this app with physical users in mind and for that reason, we looked into this potential problem anyway and found a theoretical solution.

The main idea is to use the magnetic compass of the device being played on to rotate the spoof circle. This way, the phone can be rotated towards the next player, while this rotation is being reversely applied to the spoof circle for effective stabilization with regard to the physical players. We mainly wanted to address this potential problem and show that we thought of a solution. As there are no physical players and no available magnetometer in the simulated app, this feature remained theoretical.

Related to being bound by a simulation; we have not been able to use or test a multi-human game, as there is no support for multiple simulation users.

## References


[1] Unknown, "Minimalist web design: Principles and challenges," 2021, accessed on 12/04/2024. [Online]. Available: https://www.webnode.com/blog/principles-for-minimalism-in-web-design/

[2] C. Chapman, "Simplicity is key: Exploring minimal web design," Unknown, accessed on 12/04/2024. [Online]. Available: https://www.toptal.com/designers/ui/minimal-web-design

[3] K. Moran, "The characteristics of minimalism in web design," 2015, accessed on 12/04/2024. [Online]. Available: https://www.nngroup.com/articles/characteristics-minimalism/

[4] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind." *Psychological review*, vol. 111, no. 4, p. 1036, 2004.