UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# PUPPITOR: BUILDING AN ACTING INTERFACE FOR VIDEOGAMES

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTATIONAL MEDIA

by

**Nick Junius**

September 2019

The Thesis of Nick Junius
is approved:

Professor Noah Wardrip-Fruin, Chair

Professor Michael Mateas

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

# Table of Contents

# List of Figures

## Abstract

Puppitor: Building an Acting Interface for Videogames

by

Nick Junius

Videogames have historically relied on players picking what their characters say or do from a list or entering their desired action into a text parser and then having those intentions carried out in some form by the game characters. There is an understanding that a videogame player exists somewhere between an audience member and a stage actor—but rarely are players allowed to express themselves in a manner similar to an actor. This thesis argues that the acting and directing knowledge of theater is a potentially bountiful resource for designing player and NPC interactions and proposes the reversal of the player picking an action and the characters acting out the response: have the player gesture and move as their character and interpret those actions to alter the lines of dialogue characters are speaking (or in this case, displaying on screen). To illustrate this, this thesis presents a literature review of theatrical methodology, its existing relationship to games, and a survey of projects in the interactive narrative and character interaction spaces. The chosen theater practices provide a useful basis for a new type of interaction between players and non-player characters. Additionally, particularly when looking at acting practices, their major concerns with the relationship between character and actor provide useful language to describe and further explore the relationship between the player and their avatar. As part of this exploration, we created

Puppitor, a rules-based input detection system that translates mouse and keyboard inputs into emotional affect values for use in changing the tone and direction of dialogue heavy scenes. We discuss the design principles behind Puppitor's architecture, how its inspiration from theater and fighting games influenced the implementation of each system module, authoring of rulesets and animations, and propose a direction for further work in the realm of interactive drama and storytelling more broadly.

# Chapter 1

# Introduction

Realtime interactions with non-player characters (NPCs) in video-games, if there is any incorporation of movement through space, commonly put a large degree of emphasis on these same NPCs' reactions and behavior to create engaging experiences. Most of these reactions and behaviors have revolved around physical combat or violence of some sort and, commonly, when games featuring this kind of gameplay want to have character focused, narrative moments, they take away much of the spatial and realtime elements in favor of dialogue trees. Popularized in 1997 by *Fallout* [46], much of the work advancing character interaction has been focused on graphical fidelity and voice acting rather than on changing how the player interacts, as works like *The Witcher 3* [48], *Dragon Age: Inquisition* [13], and Telltale Games' projects in the past decade have shown. With this dichotomy of interaction, even when the player has built their character themselves, they are moving between acting as their character (while walking around and in combat) and poking at their character's personality (while talking to

NPCs).

This thesis focuses on realtime gameplay due to its similarities to stage acting's live nature and our goal of building a system inspired by physical acting practice. Additionally, while evaluating *Façade*'s user interface, Sali et al found that players enjoyed the realtime, natural language understanding version of the game, in spite of the difficulties and systemic breakdowns they encountered, when compared to the more industry-standard menu options displaying full lines of dialogue or short abstract responses [56]. With this in mind, we know there is something to the live performance aspect and want to further explore that space, particularly because the industry still heavily relies on dialogue trees and other more turn-based approaches to character interaction.

When viewing the distinction between the spatial gameplay and dialogue trees, it was useful to categorize them as operating at the *acting level* and *editing level* respectively. Interacting at the acting level involves interacting in realtime with a playable model of some kind of character behavior, including combat and communication. Interacting at the editing level does not commonly involve a playable model or realtime interaction, though it may involve one of these elements (as *Prom Week* [39] does). As an example, "The Last Wish," a quest in *The Witcher 3,* involves the player fighting a djinn (a wish granting monster) and later, during a conversation, choosing whether or not Geralt and Yennefer should reaffirm their relationship. Fighting the djinn relies on the game's playable model of combat (the acting level) while the conversation between Geralt and Yennefer is a siloed, pre-authored dialogue tree (the editing level).

Even when games feature friendly NPCs, as *Bioshock: Infinite* [20] does, the player cannot interact with them at a similar level to their foes. While the player's primary companion, Elizabeth, might intelligently move and act based on where the player is looking at any given time [65], that is largely the extent of the free form, non-combat interactions. There are no verbs beyond *walk* and *look* the player has to interact with their companion, only the overloaded "interact with" button presses to begin little scripted vignettes. Still, having a character that understands their spatial relationship to the player for reasons other than combat is a very real step towards new kinds of gameplay.

In 2016, both *Oxenfree* [60] and *Titanfall 2*[1] [15] allowed players to talk to characters while moving around their environments, in contrast to traditional interactive dialogue scenes. With that said, neither game placed much weight on what the player was doing while talking with other characters. They still rely on the player picking from a set number of choices and playing certain animations based on what the player picked. What they do provide is a step towards more real time interaction between players and NPCs as they naturally incorporate character silence rather than needing to add a timer or extra dialogue option. These two games still ask the player to operate as an actor and editor even as they ask that the player does both at the same time. Additionally, the interaction the player has with the game's plot is still very much at the editing level and whatever physical acting the player may do as part of roleplaying means nothing to the game's systems.

[1]These are not the first games to allow action to continue during dialogue. *Starflight* [63] allowed both the player and NPCs to use all their ship functions while in conversations, including firing weapons.

Theater provides a framework for translating spatial elements into narrative meaning. There have been repeated calls to look to theater for design inspiration [28] [34] [64], but specifically how actors and directors use human physicality and location in space to create emotional responses has largely been ignored. We pose gesture and movement through space as two possible starting points for incorporating this theatrical knowledge into game and system design. This thesis is primarily concerned with gesture, though there is still an eye towards the creation of a *narrative nav-mesh*, spatial markup and tagging to allow game systems and NPCs to interpret player actions narratively. In this thesis we explore the theory and design of character interactions at an acting level, focusing on theater's knowledge of physicality and practices of *physical acting* as a model. For our purposes, physical acting, with regard to player avatars, refers to the affordances and actions available to the player, not that a player must perform specific physical actions themselves as input.

We take this view of physical acting, in part, because of the lower technical burden of reading inputs using more traditional interfaces, such as the mouse and keyboard or Xbox 360 gamepads, compared to real world physical gestures. It also allows us to further explore the relationship between player and avatar as we attempt to create new types of interaction using these common interfaces. Furthermore, we seek to create a constrained environment that leverages players' existing experience with realtime, combat focused games to hopefully make communicative realtime gestures more accessible. Asking players to gesture using their own body removes some of those constraints and does not necessarily increase player engagement with characters—it may even make

acting as another character more difficult [11]. For this reason we view work like Project IMMERSE [58] as falling beyond the scope of this work.

Our goal with this thesis is threefold. First, to provide an overview of some of the existing arguments that have been made about the relationship between theater and interactive narrative. Second, to identify relevant insights from these theories, as well as theories and practices of physical acting, to inform the creation of a computational system (briefly described in the following section) that enables game players to engage in physical acting, using the bodies of their avatars, as the primary interactive component. Finally, to discuss the technical details and design specifics of the system, named Puppitor.

## 1.1 System Design Goals

When discussing plot and story, we use Aristotle's definitions [1]. The *story* is made up of events the audience may or may not see. The *plot* is made up only of the events the audience experiences. For our purposes, these definitions are useful in describing the relationship between the full extent of the content authored, largely dialogue fragments (the story) and what any given playthrough will expose, the completed lines actually displayed (the plot).

In Puppitor, the primary actions available to both the player and NPCs are a constrained set of gestures with changing the speed of a gesture being a secondary action. To avoid overwhelming the player with granularity, we decided to limit the


Figure 1.1: Aristotle's distinction between plot and story as diagrammed by James Bierman. For Aristotle (and ancient Greek theater) the playwright was responsible for choosing which events in a given story the audience would see, not for creating both the plot and the story from scratch. [1].

number of gestures (beyond the default resting gesture) to three and speeds (beyond the default neutral speed) to two. Additionally, the length of time a gesture takes is directly under the player's control. They simply need to press and hold the button corresponding to a particular gesture and then release it when they want to switch to a different gesture.

Our overarching goal is to create a set of systems to allow a player, through their character's gestural and spatial physicality, to alter lines of dialogue at both the generation stage and the display stage (similar to line reads in acting). The rules governing what and how any given character can express themselves are designer defined and allow the range of emotions characters in a particular scene or plot should be able to express. These rules define how, for Puppitor, gestures change a character's expressed emotional affects and in the future, how another system would use rules attached to a narrative nav-mesh would use a character's spatial movement as part of the update cycle.

## 1.2 Research Questions

The original conceptualization of the system that would eventually become Puppitor was forged out of a simple premise: What if we reversed the interaction of a dialogue tree? Instead of having the player pick a line of dialogue and having the characters animate and move accordingly, what if the dialogue was tailored to how a player and NPC acted physically towards each other? While we propose the idea of a *narrative nav-mesh*, it turns out making a brand new interaction paradigm is a lot of work. The time it took to research and synthesize a variety of theater practices into a somewhat simple system meant that we could only focus on one part of the idea we originally proposed in *Towards an Expressive Input for Character Dialogue in Digital Games* published at the 2019 Foundations of Digital Games conference [25]. Thus this thesis answers the following research questions:

- Where has the focus of character interaction in interactive drama been in the past?
- What are some theatrical practices that can be drawn from to create a simplified model of physical acting?
- What does an implementation of part of such a model look like?
- What are authoring considerations for such a system?
- Where do we go from here?

# Chapter 2

# Existing Game Case Studies

In this chapter we will briefly look at various games to come out of the interactive narrative space and how their gameplay relates to our discussion of theater's usage of physicality and space as well as plot and story structure. Not all of the games in this chapter are interactive dramas or even explicitly interactive narratives but each game's focus is the interaction between characters, even if the player does not have an avatar.

## 2.1 Façade

Mateas and Stern's *Façade* [36] is explicitly an interactive drama and focused extensively on implementing the neo-Aristotelian model of drama Mateas outlined [34]. Integral to the interaction model was the concept of *discourse acts*, actions the player expressed through dialogue with the characters using a text parser, rather than the physical acts of firing a gun or obstacle avoidance [37]. Additionally, the two NPCs present, Grace and Trip, interpret the player's lines through abstract social games. These include

the *affinity game*, responsible for determining whose side the player is taking; the *hot-button game*, responsible for surfacing incendiary topics, exposing character backstory, and updating the affinity game; the *therapy game*, responsible for updating Grace and Trip's self-realization about their problems [37].

Much of the structure of *Façade* is rooted in the concept of beats, primarily to afford a level of autonomy to the characters within the plot [35]. Due to beats having an underlying canonical structure [37], we can characterize the drama manager in *Façade* as reconstructing the plot to fit reasonably well with whatever the player does. Additionally, *Façade* selects new beats to add to the plot based, in part, on an Aristotelian tension arc [37].

When outlining *Façade*, Mateas pointed to the importance of embodied interactions, including moving through the environment, interacting with objects, and physical contact with characters [34]. In the final experience, these are implemented but not foregrounded to the same degree as the player typing dialogue. Compared to our discussion of physicality in theater, *Façade* is light on those elements though it still exists at the acting level because the game treats the player first as a character in the world and expects them to infer the effect of their behavior through a playable model [37]. The way *Façade* uses its real-time playable model of character dialogue to redirect the plot is a major inspiration for our proposed relationship between player movement and character dialogue.

For our proposed interface, we want to provide the player with something more constrained than the free form text input of *Façade*, in part to avoid unproductive

misinterpretations of input. With that in mind, we still want to build an expressive space of player driven gesture—and use the more constrained space to enable players to more easily understand the rules governing their interactions than some of the abstracted games in *Façade* would allow [37].

## 2.2 Prom Week

Unlike the other games in this chapter, *Prom Week* is a social interaction game where the player is not a character and exists more as the narrative causality driving certain characters. Integral to *Prom Week*'s design was the concept of a "social physics engine," invoking the emergent properties of physics simulation in many modern games [41]. The primary point of interaction a player has in the game is choosing what social action a character should take, chosen from a list organized by characters' desires [38]. Characters' desires are determined by the social considerations created by the underlying social network [38]. Once an action is chosen for a character, it is up to the other character to decide whether to accept or reject the intent behind the action (like being asked out on a date) [40]. Though *Prom Week* bases its social exchanges between characters on Goffman's dramaturgical analysis and Berne's psychological games, these actions' focus on characters changing their relationships [38] in fact creates the same type of stage action Stanislavsky describes.

The player's relationship to *Prom Week* and all of its characters exists at the editing level even though there is an underlying playable model of social interaction.

This is because of the game's turn-based nature. That said, the game does allow for creative, combinatorial approaches to character action through the creation of a story world without explicit connections between actions and world states [38]. Additionally, while the characters do play stylized animations[1] in response to events, they are less character-specific than our goals for player controlled gestures and movement. Our main interest in *Prom Week* is its distillation of theories from the social sciences and observations from the operations of media into working, playable models in a digital game. Also important to us are the game's affordances for plot generation stemming from the variety of approaches to problem solving it allows [42]. While *Prom Week* is not expressly a story generator, that its primary concern was with characters' goals, not with a well structured plot, allows for a significant amount of transformational variety and player expression in the creation of a plot through a given scenario.

This variety in the plots through a given scenario was facilitated by *Comme il Faut* (*CiF*), the social system of *Prom Week*, and its decoupling of character, role, and action within the system [41]. Another important goal of *CiF* was to reduce the burden of authoring social exchanges compared to the behaviors of *Façade* [41]. This modularity lends lends itself to our goal of building plots based on players' actions, incorporating the Viewpoints' emergent view of movement, and our observation of the potential transformational variety in Stanislavsky's concepts of leading characters and objects bearing psychological load.

[1] Similar to the expressive gestures described in the Viewpoints.

## 2.3 Versu

*Versu* is an interactive drama built using autonomous characters constrained by social practices (recurring social situations) [16]. Characters' actions arise from their own beliefs and desires—rather than characters being forced into acting based on a drama manager [16]. Additionally, *Versu* decoupled social practices from characters and characters from roles, similar to what was done in *Prom Week*, to allow for significantly more possible variations within a story [16]. By making everything modular and simulation driven, one of *Versu*'s goals, along with its button clicking interface, was to make the process of understanding the rules of the world easier, and thus allowing the player to act in a more informed way than in *Façade* [16]. With the type of interface we've been describing, we want to split the difference between *Versu*'s explicit, always valid, set of choices and *Façade*'s purposefully obfuscated underlying systems. As described in the section on the Viewpoints, we believe that everything the player can do should be valid to the simulation (as *Versu* does) though we also see the value in the hiding some of the details about the underlying system to further ground the player in the experience (as *Façade* does).

What the player is physically doing in *Versu* is no different from our earlier characterization of dialogue trees and *Prom Week*'s interaction, meaning the interface itself exists at the editing level. To return to Zeami's comparison of actors to puppets, *Versu* makes the existence of the strings controlling the puppet central to the experience. Though we want to create something more continuous and graphically focused

than *Versu*, its autonomous approach to characters, along with its similarities to *Prom Week*, connect it to our view of plot construction, the emergent properties of movement according to the Viewpoints, and Stanislavsky's concept of the leading character. With this in mind, our proposed system falls somewhere between *Prom Week* and *Versu* at one end and *Façade* at the other. We want to keep the real time interactions and compositional elements of *Façade* but constrain the base actions available to the player to allow them to get a clearer understanding of the rules governing the interactions as described in *Prom Week* and *Versu*.

## 2.4 La Dama Boba

*La Dama Boba* is an adventure game adaptation of the Lope de Vega play of the same name designed to help introduce high school students to theater and the specifics of the play [33]. The adaptation process was heavily influenced by the Americanized version of Stanislavsky's work [33] we discussed earlier. Once Manero et al had picked the player character, Laurencio (the male lead of the original play), they used Stanislavsky's concept of the superobjective for a single character to choose what needed to be cut from the script without breaking their leading character's plot line [33]. They then separated Laurencio's arc into separate milestones to then turn that section of the play into one of five adventure game challenges that made up the game's plot [33]. Each character was given an agenda to facilitate conflicts and engage the player through Stanislavsky's definition of stage action [33]. Additionally, the game was made non-

linear as certain milestones were considered to be semi-independent and described as creating a performative space [33].

We find the usage of Stanislavsky's objectives interesting from an adaptation perspective but when looking at a playthrough of the finished game [12], the connection to Stanislavsky's methods are somewhat difficult to see and could be characterized as the *Tale-Spin* Effect[2] [66] in relation to the design of a game rather than its processes. Some of this is due to the game explicitly being designed as an adventure game and within the constraints of adapting an existing play. In our view, the player's interactions with the game exist at the editing level because of its adherence to adventure game conventions, and thus lack of an underlying playable model, even modeling one of its challenges directly on the battle of wits in *The Secret of Monkey Island* [33]. Additionally, the description of the adaptation process is mostly concerned with the design-time writing of characters instead of how characters express themselves in the final artifact.

There are two instances where the game does allow the player to get closer to acting as their character rather than editing them. First, the player must complete a Redonilla[3] [33] and is actually finishing sentences on the page. Second, the player must find and correct spelling errors in a poem [33], again on a page rather than simply telling the character what to do. In our view, the *La Dama Boba* game provides an example of the limitations of adapting novel design approaches within the constraints of existing game design paradigms.

[2]When the output of a process hides the underlying complexity of that process.
[3]A form of Spanish poetic composition [33].

# Chapter 3

# The Crossover of Games and Theater

Aristotle has been one of the more influential guides for discussing the relationship between games and theater since Brenda Laurel's dissertation and subsequent publication of *Computers as Theater* [28] [34]. More recently there has been more interest in Konstantin Stanislavsky's acting methods as a lens for understanding interactive narrative [64] and as a game design methodology [33]. In this chapter we focus on how theater has influenced the current understanding of games and interactive drama as well as how our proposal for building systems inspired by theatrical acting and directing fits into this prior work.

We have chosen to avoid discussing the work done applying Boal [17] and Brecht [57] to videogames in significant detail in this chapter primarily because of our focus on the transformational properties of more traditional acting techniques [51] [29] [2]. For Boal's Forum Theater, the *spect-actors*[1] would take turns playing the protagonist

[1]The audience members are participants, not simply passive observers in Theater of the Oppressed.

to explore how the protagonist's oppression could be broken [17]. In a sense, Boal's practice was more akin to the rehearsal described by more traditional theater practitioners [51] [29] [2] while we are more interested in what it means to be asking the player to be a more traditional type of actor during a performance.

Brecht's view of the character and actor as remaining distinct [57] is applicable to our interest in physical acting. However, at this point in our exploration of the subject, our primary concern is in better understanding the effects of attempting to capture the transformational properties of physical acting as a game interface. Additionally, with our definitions of the acting and editing levels of interaction, we view the switches between the acting and editing levels as implicitly Brechtian [57] in how they change the distance between player and character in a game like *The Witcher 3* when transitioning from combat to dialogue scenes.

## 3.1 Acting and Constraints

One of the early observations Brenda Laurel makes in *Computers as Theater* is that simply placing an audience on stage is not a useful metaphor or practice for applying Aristotelean thinking about the theater to human computer interaction. She specifically states that by inviting the audience to interact with the the work, either theatrical or digital, they necessarily can no longer be considered audience members or observers: they must, by definition, be actors [28]. Following this assertion, Laurel says "Optimizing frequency, range, and significance in human choice-making will remain inadequate

as long as we conceive of the human as sitting on the other side of some barrier, poking at the representation with a joystick or a mouse or a virtual hand" [28]. In other words, when we previously characterized the current standard of character interaction in digital games (the dialogue tree) as operating at the editing level, it is because the player can only prod the characters into acting; they are not in fact acting alongside the characters.

Another important observation Laurel makes is that the impreciseness of theater is a strength when looking to it for inspiration in interface design, particularly when designing interactive stories. Of particular note is her assertion that "when 'imprecision' works, it delivers a degree of success that is, in balance against the effort required to achieve it, an order of magnitude more rewarding than the precision of programming, at least for the non-programmer" [28]. Specifically with regards to games, designers don't always need to strive for perfect simulations or understanding of player intent. After all, misunderstandings and unintended consequences are a common cause of conflict and conflict is central to drama. Why shouldn't we embrace the conflict that emerges from the interaction of players and systems at a narrative level? By framing systems as characters, these breakdowns can actually make characters more believable and engaging [27].

Laurel also reminds us that how a failure is presented is largely responsible for whether there is a breakdown in the experience [28]. A text parser not recognizing a word and simply stating that fact will not create a conflict useful to a plot. A character misinterpreting what the player said as flirting and then forcefully rejecting them is

useful to a plot. While both of these are failures of a system to recognize player intent, in order to create a useful conflict, an action, or at least a strong suggestion of action, should follow. Otherwise the player is asked to do the bulk of the work repairing the breakdown on their own.

In building interactive spaces, game designers must also be creating extrinsic and intrinsic[2] constraints for interacting with the world. As Laurel observes: "the actor is constrained in the performance of [their] character primarily by the script and secondarily by the director, the accoutrements of the theater... and the performances of [their] fellow actors... In spite of these narrow limits, the actor still has ample latitude for individual creativity" [28]. Again this is why we do not usually consider a player's interactions with a dialogue tree as existing in the acting level. The constraints placed on the player rarely allow for creativity in expression or choice, rather they are simply expressing a preference of tone or character that already exists within the authored content. A combat encounter in *F.E.A.R.* [47] still has a massive number of explicit and implicit constraints placed on it, largely personified by nearly every button the player can press creating an attempt to cause physical harm. As Laurel describes "intrinsic constrains should not shrink people's perceived range of freedom of action, but rather enhance them" [28]. In this way the constraints in *F.E.A.R.* breed creativity in players in a similar way the constraints of a play script and production breed creativity in an actor.

[2]Extrinsic constraints refer to the context of a person as an interactor, the placement of "pause" and "quit" buttons on a controller layout for example. Intrinsic constraints refer to the context of a person's ability to interact within the fictional world, whether they can pick up and throw a rock for example [28].

## 3.2 Aristotle and the Causes

Aristotle has been the centerpiece of both Laurel and Mateas's work in interactive drama, with the Four Causes informing how player agency fits into traditional narrative structure. *Formal cause*, the structure created by the playwright (or designers), and *material cause*, what the audience experiences [34], have been given the most focus, with *efficient cause*, the material components that contributed to the production (including skills, tools and techniques), and *end cause*, the emotional experience of the audience [28], being less emphasized. Laurel does however observe that "the human interactor is also part of the efficient cause; that is, interactors are co-authors" [28] and this is one of our primary interests in building systems to facilitate digital acting. While actors may not have the agency of a character, they still have a large amount of agency in how a character is expressed and experienced.

In building the definition of interactive drama, Mateas incorporated Janet Murray's categories for the analysis of interactive story experiences: immersion (the feeling of being present and engaged in another place), agency (the feeling of empowerment from taking intentional action in the world), and transformation[3] [34]. Of these three categories, he considers agency to be the most fundamental to creating an interactive drama, in part because he views immersion and transformation as already encapsulated in the Aristotelian model. His characterization of transformation, as a change in the

[3]Mateas identifies three distinct meanings of transformation: transformation as masquerade, or allowing players to become someone for the duration of the experience; transformation as variety, or allowing players to explore facets of a theme from a variety of perspectives; and personal transformation, or facilitating players' self discovery.

protagonist, and additions to Aristotle's formal and material causes [34] explicitly position the player as a *character* in the world, not simply enacting a character, and to have a satisfying experience they must be able to affect the world as a character would.

Like Laurel's discussion of intrinsic constraints, Mateas's definition of agency in interactive drama relies heavily on how setting boundaries through material (constraints dictated by what the player can interact with) and formal (constraints dictated by plot and setting) affordances allows for a feeling of agency and creativity. He defines an experience creating a high amount of agency when the actions available to the player appear in concert with with what the plot, setting, and structure dictate and a lack of agency when there is an imbalance between the formal and material affordances [34]. In doing so he invokes Aristotle's idea of unity of action, or when actions in a plot all build towards some goal. It is for this reason we are specifically interested in acting and directing methodology since they provide fairly concrete ways of constraining the available actions for an actor in a natural way while still giving them agency over the character, scene, and plot.

In his discussion of immersion in relation to the neo-Aristotelian model of interactive drama, he re-frames it as usefully providing formal and material constraints for players. In particular, when talking about immersion, he mentions the usage of masks (as player avatars in games) as providing material and formal constraints for a player's actions [34], allowing them to better engage with the world. Theater has long used masks and other devices to facilitate action on the stage, both for the audience and actors' benefit [51] and it's here where we are primarily interested in leveraging

theatrical knowledge that remains largely untapped. Compared to masks, costumes, and puppets, a digital game character can provide many more intrinsic constraints and thus, by building interactive systems expressly using the same principles, we can more readily facilitate players acting as characters the way Mateas describes.

Mateas's reincorporation of transformation into his model of interactive drama provides a guide for the type of dialogue interface we are interested in. He states that there shouldn't be easily identifiable branch points and that possibilities should necessarily narrow as actions build up and force the plot in specific directions [34]. We are still interested in using the core of his explanation of this interactive dramatic structure as there are still currently very few experiences built with this in mind. However, rather than dialogue being the primary mode of interaction, the player's avatar's movements will inform the direction the conversation takes. In part, we want to further explore the player's relationship with their avatar and a new form of character interaction will be our starting point.

## 3.3 Performing a Role

In response to the focus on allowing players to shape plot in interactive narratives, Tanenbaum argues that participation in a story, rather than interaction, is a way forward for creating interactive digital stories. He likens this participation to the constraints placed on actor in a scripted play and a type of *bounded agency* [64], similar to how Mateas views transformation as masquerade. Throughout the discussion of bounded

agency, Tanenbaum cautions against seeing the player as a co-author, citing Laurel's ideas regarding intrinsic constraints [64]. While we find this warning useful, we feel it has the potential to be overly restrictive and minimizes the player's role in the efficient cause. We believe there is still space within the confines of bounded agency to allow the player some level of control over the plot, specifically by building narratives using Aristotle's distinction between plot and story, and his view of the playwright[4] [1]. Mateas points to this when he says "each run-through of the story has a clean, unitary plot structure, but multiple run-throughs have different, unitary plot structures" when discussing a dramatic world with agency [34]. With this in mind, we can view the player as one of the constructors of the plot, within the story space the designer has created. It is still useful to view the player primarily as an actor in this type of experience, specifically to define how they interface with the world.

When discussing method acting, Tanenbaum notes that "not everyone who engages in an interactive story wants to be an author" [64]. What interests him is the transformative potential of method acting and how that transformation can make scripted lines feel spontaneous and concludes that "acting is a process that uses external perceptions and actions to transform the internal state of the actor" [64]. In light of the inherent challenges that come with acting, our goal is to provide an interface to facilitate a similar experience to acting but is constrained enough to keep attention on the dramatic action, not in the difficulty of acting. We believe limiting the player to consistent physical actions and interpreting those actions to drive the dialogue (and

[4]Aristotle viewed it as the playwright's job to pick and order events rather than fabricate the stories those events are drawn from [1].

narrative progress) will avoid some of the problem of gating plot behind seemingly arbitrary challenges [49]. We do not want to force everyone interacting with this type of dialogue system to have to be an actor. Rather we are happy to give players a set of tools to get an idea of what the character they are playing is like. We want the mask their avatar represents to be as easy as possible to put on.

Tanenbaum and Mateas, as we discussed briefly in the previous section, find the theatrical practices of masks as a useful lens to view player avatars and, more broadly, to create a focal point for the intrinsic constraints placed on the player. Tanenbaum says "we can imagine a player's avatar as a form of Mask with a set of powerful character associations built into it" [64] and one current example to draw from is the *Street Fighter* series. *Street Fighter* might not be a narrative game in the traditional sense but every character's set of moves is both an expression of their character and a set of constraints both extrinsic and intrinsic. From the original release of *Street Fighter II* [4] onwards, Ryu and Guile have shared similar moves (they both can throw projectiles and have powerful, high risk uppercuts) but the differences in how the player performs these moves translate to noticeably different playstyles. Guile must hold a direction (back for projectiles and down for uppercuts) for a second or two (depending on the game) before pressing a button for a move to register. Ryu needs to perform a quick series of directional inputs before pressing a button for a move to register. When playing either character, the player will develop the sense that Guile is someone who needs to plan and focus for more powerful attacks while Ryu is more willing to rely on his reflexes. With this in mind, while we don't want to replicate the special moves system found in

*Street Fighter* (and numerous other games in the genre) we do find their connections to masks and physical acting affirming in our desire to use character movement as players' primary interface with a plot.[5]

If we want to follow the promise presented by Laurel's belief that bringing the audience into the experience makes them actors, then we need to be building systems around constraints of performance of character in the same way we build systems around combat and navigation of space. Noticeably absent from much of the work done applying theater to games is theatrical direction, where many of the decisions about the use of space and physicality reside. This is why we view the player as only one of the authors of the plot in our proposed approach. It is up to the game itself, through a system heavily informed by directing and acting methods discussed in the following chapter, to interpret the player's actions and add to the plot based on that interpretation and available story content.

[5] *Elemental Flow* [21] is a conversation based role-playing game currently in development also drawing inspiration from fighting games.

# Chapter 4

# Theatrical Theory and Practices

In this chapter we look to an interpretation of the Stanislavsky method, elements of Japanese *Nō* theater, and the Viewpoints practices as a basis for constructing a set of player actions and a model of interpretation. We picked these techniques specifically because they are practices used in training actors and directors in addition to being incorporated into the rehearsal process. We want to be clear, we are not saying these are the only ways to facilitate the type of physical acting we are trying to create. There is far too much written on physical acting to be comprehensively incorporated into this paper and the practices we discuss below were the ones we felt fit our goals the best. We should also note that these practices focus on training humans to act in physical space themselves. Because of this, given that we are building a digital system, our interest rests more in how these different techniques convey emotion—though the specifics of how they facilitate the internal transformation of actors are still useful to consider for exploring the relationship between player and avatar.

In attempting to create the outline of a digital model of physical performance, we are not attempting to capture and encode a single model and philosophy. Instead, and in concert with the intrinsic constraints we seek to create, we believe drawing from different traditions will allow us to construct a model reflective of our desire to mold dialogue around player actions.

## 4.1 Stanislavsky and Energy

Stanislavsky's method is one of the foundations of modern, naturalistic acting and though his work was indeed published, the changes he made towards the end of his life were instead primarily passed down through the Moscow Art Theater Studio Theater School and the State Institutes of Theater Art in Moscow and St. Petersburg. What Stanislavsky left behind would evolve into the Method of Physical Actions. This evolution would emphasize each actor's relationship to everyone else on stage as action was performed, further strengthening Stanislavsky's view of *stage action* as being born from disagreement, conflict, or struggle between people [29]. This definition of action allows us to further embrace Laurel's view of the impreciseness of drama [28] and more easily embrace a system's failure to properly interpret the player's actions as a source of conflict. In this way, we want to treat the interpretation system itself as part of our digital stage and as a partner for a player to play off of.

Another component of stage action for Stanislavsky is that the conflict generating the stage action must come from somewhere within the world of the play and

be directed towards a partner. Whichever character initiates a conflict is considered the *leading character* of that action. For Stanislavsky, there are only two possible relationships between characters engaged in a conflict. Either the leading character is attempting to impose their view onto another character or they are making an observation about their "opponent" [29]. While much of the mechanics of what Stanislavsky describes here in regards to conflict is more applicable to how dialogue is authored for our proposed model, the concept of the leading character fits well with our goal of building dynamic conflicts in a plot. By having the model consider a new leading character after every action, the player will not always be the one in control of the direction of an action. With this, we hope to achieve the unique, unitary plot structures Mateas describes [34] with the addition of failures of the model to always interpret player intent correctly being a source of action.

When actually, physically performing a role, Stanislavsky refers to the concrete, external performance of a stage action as an *actor's adjustment*. He describes the particular arrangement of adjustments, defined in a production's rehearsal to be repeated each show, as part of the *mise-en-scéne*, the arrangement of scenery and stage properties in a play. Crucially he notes that it is impossible to predict exactly what an actor will do within the constraints of the mise-en-scéne and that an actor's adjustments should never become fixed and repeated [29]. This is of particular importance to us as we want to build enough constraints into our model to allow players a range of physical expression without relying on something as discrete as the move inputs in *Street Fighter*. We are still constrained by making a digital system and, to reduce the

Figure 4.1: Performance of Anton Chekhov's *The Cherry Orchard* by the Moscow Art Theater in 2004 (Renata Litvinova as Ranevskaya and Andrey Smolyakov as Lopakhin pictured) [14]. The Moscow Art Theater has been a focal point of the continued development of Stanislavsky's methodology and was where he developed much of his philosophy.

complexity of the interface, we necessarily limit the number of gestures available to the player and rely on some number of authored animations rather than ask the player to construct complete gestures entirely on their own.

Important to Stanislavsky's view of physicality is the concept that each action is performed with some amount of energy and that actors' external movements should be preceded by an internal movement. He likened this internal movement to controlling a ball of mercury and that an actor's body should be reacting to where the actor wills the ball. Additionally, he makes sure to note that this direction of energy should not only be contained within an actor's body but flow out as an extension of their will. With energy, like with conflict, containing it within a single actor or character

is not enough. To create a stage action, energy must be directed at a partner and the direction of energy towards a less than willing partner is a *dispatch of energy with impediment.*[1] Of note is the two forms the impediment can take. First, an actor can restrain energy that wants to be directed at their partner. Second, an actor can direct energy at their partner to push them away [29]. We view this concept of energy as fundamental to building the library of gestures for players to use, particularly because it allows for much more readable expressions of player intent than purely relying on a model of movement through space. Additionally, by basing the meaning of gestures on this definition of energy, the interpretation of a gesture can be deterministic in our simplification of physical acting, allowing players to more easily understand how the system views their adjustments without being overly prescriptive (as gesture is not the only element of picking a line of dialogue in our model).

This is one of our reasons for limiting the player to three distinct gestures. By default, a character will project energy, though with little force, towards other characters and willingly receive energy. The three gestures then correspond to Stanislavsky's description of energy flow between characters. One will represent the unimpeded flow of energy to bring the other character closer to the gesturing character's view or position. One will represent the projection of energy at a character to repel that character and close the gesturing character off. One will represent the restraint of energy towards another character to hide the gesturing character's feelings towards the other character.

With regard to objects on stage, Stanislavsky puts them in two categories,

[1] This is in contrast to the unrestricted flow of energy characters can project.

objects that bear a psychological load and objects that do not. Objects that bear a psychological load are explicitly tied to a character and help an actor explore that character's internal life and their relationship to other characters. Note that these objects do not have to belong to the character to have a psychological load. Objects that do not have some psychological importance are simply tools for actors to add to their performance [29]. In his description of the material affordances, Mateas notes the importance of objects for player action [34] and our interest in objects is much the same as his. We consider objects that bear a psychological load as modifiers to gestures of a character rather than being additive. They afford us a way of altering the energy system underlying gestures and the gestures themselves while still maintaining our limit of three available gestures. Similar to the concept of the leading character, Stanislavsky's two categories of objects create the possibility of more dynamic interactions between a player and NPC through the use of both types of objects, and even changing which objects carry psychological load on repeated playthroughs to allow for more transformational variety as Mateas describes [34].

While we do see Stanislavsky's concept of a character's *superobjective*, a volitional objective (an answer to the question "what do I want?") that must encompass all of a character's desires [29], as useful to building constraints for players, we believe it exists too much at the editing level for our purposes. Additionally, the Method of Physical Actions was a step away from this purely internal focus, which characterized much of the American interpretation of Stanislavsky's work [29], and is more interested in how the external informs the internal (what actually interests us). This is also one

of reasons for taking inspiration from Zeami's essays on *Nō* theater.

## 4.2 Zeami and *Nō* Theater

Zeami Motokiyo was the progenitor of *Nō* theater in Japan during the 14th and 15th century where he built his father's theatrical practices into a total experience, incorporating mime, dance, poetry, and song—a practice that has continued to present day. Zeami's essays we are drawing from were never intended for wide circulation and are primarily concerned with defining his views on how *Nō* should be practiced, both in the mechanics of the art itself and the complete commitment needed by an actor from the age of seven onwards [51]. Of course we aren't in the position of asking for this level of commitment from our players, but for Zeami, his art was a lifetime commitment—and his likening of the aesthetic effect of theater to a flower, and its ever changing nature, reflected this [51].

His metaphor is still useful for our purposes as one of his primary concerns, like Stanislavsky, is that performance must be alive, grow, and change with both the practitioner and the audience. One final note before we discuss some of his philosophy and practice: unlike Aristotle, Zeami was a practitioner of theater and as a result, he often makes note of how his teachings should be used in relation to an audience [51]. As we are making a digital game, and explicitly interested in Laurel's view that a player/interactor is an actor [28], this reminder of having an audience is useful since we do not want to build an experience that requires theatrical knowledge to be engaging,

Figure 4.2: *Nō* performances feature dance, live music, and song to create a stylized performance where the actors are the center of attention [67].

much like how Zeami thought it necessary to create an engaging experience regardless of an audience's understanding of *Nō*.

In describing role play, Zeami states that, in general, role play involves imitation, though the degree to which the imitation reflects reality depends on the station of the character being portrayed. He notes that it is not just movements that must be imitated but dress; poorly considered costumes can undermine a performance. Additionally, when describing the role play associated with different roles, especially of women and old men, he emphasizes the importance of conveying grace, as well as character, through physical movements. Regarding his description of how an old man should be played, he likens the movements to capturing the same beauty as an old tree still blossoming [51]. For Zeami, physical action was about conveying both the physical and

internal truth of a role, though in a significantly more stylized way than Stanislavsky, and it is this approach to stylization that we want to use to inform the actual gestures players perform. Stylization can help make gestures more clearly read as a particular affect and more easily surface some of a character's internal feelings, making it easier for players to understand their character in the moment [2]. Additionally, Zeami's view of the importance of looking the part, not just acting the part, strongly parallels Laurel's view of intrinsic constraints creating an explorable space of action, a guiding principle for us when creating our characters' physical appearances.

Similar to Stanislavsky's view that all stage action must come from the text of a play, Zeami sees all movement as necessarily being informed by the words chanted on stage or, "communicate first by hearing, then by sight." [51] In his example of an actor portraying a character weeping on stage, Zeami notes how if the gesture of raising a sleeve to the face precedes the concept of weeping, the words will seem out of place and the totality of the moment is reduced. More broadly he states how a person's intentions give way to their behavior and that by informing movement on stage with the words of a play, the action will feel natural [51]. This is a much finer grain view than Stanislavsky's volitional objectives and informs us of another utility we must write into characters' lines: a suggestion of action. Though we are picking the next lines of dialogue based on what the player does, those lines should help guide a player in some way towards further stage action to keep them continuously engaged in how a scene plays out.

When describing the necessary novelty of performance, Zeami emphasizes the

need for an actor to continue to find new ways of performing the same gesture to, if nothing else, continue to refine their art. He also notes that even if there aren't visible external changes to the performance, the audience will pick up on the novelty the actor has found. It is from this novelty of performance that Zeami sees the transformative potential of acting and in his example, how an actor can move from imitating an old man to becoming one [51]. For our model, and since we are interested in having the player help construct the plot, Zeami's view of the power of transformation for an actor is very much what Tanenbaum describes as a way forward for interactive narrative. By giving the player control of their adjustments at all times, they will be able to find the novelty Zeami describes even if they find themselves in a plot they've already experienced.

Zeami likened what the audience sees of an actor on stage to a marionette and the actor's presence and intensity of mind to the strings controlling the puppet [51]. By turning the audience into an actor, in one sense we are expressly going against his belief that the audience should never see the puppet's strings. In another sense his metaphor is useful to us in how it ties back to the transformational pleasure of becoming another character. For us, it is the player's job to breathe life into their character and become that character through the interface we build. We may be asking the player to use the strings Zeami describes, but if we can make the strings disappear and facilitate the player feeling as though they are acting, we will have succeeded both in creating a new type of dialogue system and keeping the audience unaware of the puppet's strings.

## 4.3 The Viewpoints

The Viewpoints approach to movement and staging was born out of the 1960s and 70s and a desire to question everything about traditional performance techniques. It was partially a reaction to the internal focus of the American interpretation of Stanislavsky's work. Born from innovations in dance and choreography, the Viewpoints turned the source of movement inward and expressly professed that whatever movements came from this shift in perspective was the art itself or, "what made the final dance was the context of the dance" [2]. Viewpoints, much like Zeami and Stanislavsky, sees the process of performance as integral to the final product. This perspective is integral to our approach to plot as being a collaboration between the player, designer, and game system. Additionally, it provides another description of Mateas's view of the transformational variety of interactive drama and the value of such variety.

Anne Bogart an Tina Landau define the Viewpoints as:

- "A philosophy translated into a technique for (1) training performers; (2) building ensemble; and (3) creating movement for the stage" [2].
- "A set of names given to certain principles of movement through time and space; these names constitute a language for talking about what happens on stage" [2].
- "Points of awareness a performer or creator makes use of while working" [2].

They also use nine Physical Viewpoints, separated into Viewpoints of Time and Viewpoints of Space, rather than the Six Viewpoints developed by Mary Overlie [2].

Figure 4.3: Viewpoints training emphasizes being in tune with one's own body as well as other performers. Many of the techniques used incorporate at least some degree of activity to create group cohesion and allow for trust to be built during the rehearsal process [43].

These Physical Viewpoints are our primary basis for interpreting a character's movement through space to build a narrative nav-mesh as well as informing contextual actions related to the playable space. Bogart and Landau also developed Viewpoints for voice, but the incorporation of voice is beyond the scope of our model for now. What follows is a discussion of the nine Physical Viewpoints.

### 4.3.1 The Viewpoints of Time

These include *tempo*, the rate of speed at which movement occurs; *duration*, how long a sequence of movement continues, specifically how long a group of people stay inside a certain section of movement; *kinesthetic response*, the impulsive movement that occurs from a stimulation of the senses; *repetition*, repeating a movement within your own body or repeating the shape, tempo, gesture, etc. of something outside your own body [2].

For our model, these Viewpoints primarily inform the development of the interface between player and character. We seek to give players as much control as possible over their character's movements and gestures, within the constraints of purely digital input. We particularly want to allow players to explore tempo and duration, with the gestures we create for characters, by allowing them to choose how quickly to perform and how long to hold particular poses. We cannot capture the granularity of what these Viewpoints are capable of in physical space, in part to avoid overwhelming players with options, but we feel that by providing players a constrained version (the distinct default, run, and slow walks described in Section 1.1) we will allow them to experience a similar type of expressiveness in a much shorter amount of time.

### 4.3.2 The Viewpoints of Space

These include *shape*, the outline the body (or bodies) makes in space made up of lines, curves, or some combination. Additionally the shape can either be stationary or moving and can take one of the following forms: the body in space, the body in relation to architecture, the body in relation to other bodies; *gesture*, shape with a beginning, middle, and end and broken into *behavioral gestures*[2] and *expressive gestures*[3]; *architecture*, the physical environment and its effects on physical movement to create spatial metaphors[4]; *spatial relationship*, the distance between things on stage especially one body to another, one body (or bodies) to a group of bodies, and the body with architecture. Additionally

[2]These are observable behaviors in everyday reality. [2]
[3]These are abstract and symbolic gestures aimed at representing an internal truth. [2]
[4]The process of giving form to feelings like "I'm trapped" or "I'm up against the wall." [2]

there is an emphasis placed on extremes of distance and the expressiveness of changes in distance; *topography*, the landscape, floor pattern, and design of a space as characterized by movement through it [2].

These Viewpoints are the primary basis for the creation of a narrative navmesh, with particular attention paid to architecture and topography. Topography especially can allow us to create spaces that change the way a player's character is able to move, allowing them to directly feel their character's reactions to the environment and choose how to play with that reaction. By building a model grounded in the shapes created by the position of players in their environment and relative to other characters, we can begin to interpret player movement through space in a constructive way and use that interpretation to pick appropriate lines of dialogue.

### 4.3.3 Soft Focus

This is described as the physical state where the eyes relax and take in more than simply one or two things in sharp focus. A primary goal of soft focus is to allow a person to look at their surroundings and other people without desire. In the physical space, this is primarily to allow actors to have a more holistic view of the space they inhabit and connections to everything happening on stage [2]. For our purposes this concept is useful for two reasons. First, we want players to do more than stand in front of other characters to interact with them, encouraging them to move through the environment, either through dialogue or the design of the space itself. Second, as players are not in direct control of what their character says, their relationship to other characters will

not be purely transactional since the other characters are the core of the experience not simply a means to an end.

Bogart and Landau explicitly position the Viewpoints as a collaborative approach to performance with the goal of, to return to Laurel's intrinsic constraints, giving performers the freedom to explore all possibilities within a space [2]. Noteworthy for our purposes is that all of those possibilities represent a valid approach to the performance [2]. Therefore all the actions our interface enables must allow player expression to the game system to be used in the construction of the plot in some capacity. Beyond this, there are significantly more details about the Viewpoints than we can include, the bulk of which are about the specifics of their practice and incorporation into the rehearsal process.

We should note that we are not the first to attempt to create a computational system inspired by the Viewpoints. The Viewpoints AI project [23] sought to create a gesture recognition system using physical gesture. Unlike the system we are proposing, the Viewpoints AI utilized Microsoft's Kinect as its source of input and used Bogart and Landau's ideas as a way to translate the player's performance in physical space into something understandable by a virtual character. As we have already stated, we are not building a system to recognize physical gestures from players' bodies, rather, we are using the Viewpoints to guide our design of player actions, both by having these actions always be interpreted meaningfully by the system and through our decisions of which viewpoints to put directly under the player's control and which are concerns more for content authoring.

# Chapter 5

# System Description

## 5.1 System Introduction

Puppitor is specifically designed to translate keyboard actions into character gestures, the details of which are discussed in sections 2 and 3. This is in contrast with prior interactive drama systems, like those found in *Façade* [36], *Versu* [16], Soar AI [32], and the Viewpoints AI system [23]. *Façade*'s text parsing interface is the closest existing design to Puppitor, though it offers significantly fewer constraints on player interaction and focuses more on the player writing character dialogue than on physicality. *Versu*'s interface, for all its breadth of choice, still has players picking off of a list. The Soar AI system is significantly more concerned with accurately modeling human psychology and physiological drives than Puppitor. While our system is gesture based, and indeed taking inspiration from the Viewpoints as discussed in chapter 3, this is where we see the similarity to the Viewpoints AI ending.

To further contrast Puppitor with Soar AI and the Viewpoints AI, we are making a *computational caricature* [59] of both stage acting and the expression of emotion. Smith and Mateas describe computational caricatures as broadly describing systems that embody a theory about the important elements of a domain and providing avenues of direct inquiry into the subjective nature of a system's implementation, particularly the bias, exaggeration, and oversimplification that come with reifying aspects of a computational system.

Puppitor is a simplified combination of the approaches to acting taken by Stanislavsky, Zeami, and Bogart and Landau (described in chapter 3 and further discussed in sections 4 and 5). Puppitor's model of emotional expression is derived from a simplified interpretation of Descartes' six universal passions, wonder, love, hatred, desire, joy, and sadness, described by Joseph Roach [52] (further discussed in section 7). By focusing the system design on the elements most important to the problem domain, the computational caricature approach to system design allows us to make more informed design choices about which elements should be authorable.

Puppitor is primarily focused on leveraging more traditional forms of user interface to explore new modes of play rather than using gesture recognition as the input method. This is partially due to the technical difficulty of interpreting human movement compared to using a keyboard or gamepad. Additionally, as we previously pointed out in chapters 2 and 3, across both game design and acting practice, constraints foster creativity, and the limited number of inputs our system uses is a way of avoiding overwhelming players [7] and encouraging exploration of the interface. It is here where

inspiration from fighting games, like the *Street Fighter* and *Tekken* series, and, more broadly, the study of game feel [62] comes into play.

While we aren't replicating the technical complexity of inputs found in either 2D (e.g., *Street Fighter*) or 3D (e.g., *Tekken*) [22, 61] fighting games, their usage of animation to express game information [26] (further discussed in section 8) and connection of buttons to specific character actions has been a source of technical and design inspiration for Puppitor. After an experiment in physicality of its own,[1] *Street Fighter* [3] has used a six button layout, with the top row of buttons corresponding to punches and the bottom row corresponding to kicks, ascending in strength from left to right. Since its first release, *Tekken* [45] has given players individual buttons to control characters' limbs, often called left punch, right punch, left kick, and right kick, to allow for a more literal connection between player and character actions.

As we are using the Stanislavsky energy states (open flow, closed flow, projected) that we describe in chapter 3, our buttons correspond more figuratively to our characters' actual actions than either *Street Fighter* or *Tekken*'s input metaphors allow. Instead our input metaphor of button press as a class of character action more closely mirrors more stylized games like *Under Night In-Birth* [18], where the general actions associated with buttons are named using the letters A, B, and C. In *Under Night*, the specific action corresponding to a button is entirely determined by the character being played. This is in contrast with *Tekken* and *Street Fighter*'s approach to making an explicit connection between a button and a character action (e.g. left punch in *Tekken*

[1] The original version of *Street Fighter* determined the strength of a punch or kick based on how hard players hit the pressure sensitive pads on the arcade cabinet [61]

or medium kick in *Street Fighter*).

David Surman describes the execution of a special move (like throwing a fireball) in *Street Fighter* as having a:

> moment of correspondence between the special move and the player's gameplay performance, [where] there is a heightened sense of gratification on the part of the player as he or she 'becomes' their chosen martial arts superstar, in that peculiar subject position characteristic of videogames [61].

While Puppitor's interface does not involve any input like the special move motions in *Street Fighter*, the connection between player and character Surman describes is the same connection we hope to evoke with Puppitor.

## 5.2 Overview

The gesture system we outlined has four main steps in its interaction loop:

- The player pressing a button (or buttons).
- The player character performing a corresponding gesture.
- How the gesture was performed updating the values corresponding to emotional affects.
- The largest affect values being used to generate a line of dialogue.

The rules governing the translation of a button press into a character gesture are directly inspired by all three of the theater practices we discussed. Each gesture, beyond the default state, is labeled as one of Stanislavsky's energy states: open flow,

closed flow, and projected energy. The animations a character performs when in each of those energy states is that character's physical interpretation of embodying that particular energy state. For example, a character who is usually open and honest about their feelings would have a more open stance towards the other character when performing the gesture corresponding to open flow and when performing the closed flow gesture would position themselves to literally be trying to keep everything they can close to their core, similar to how Zeami describes actors' movements conveying grace and character. Additionally the control the player is given over the tempo, duration, and repetition of each of the gestures comes directly from the Viewpoints of Time and contributes directly to the concept for the basic control scheme for the interface.

Each button corresponding to a gesture can be held for as long as the player wants, holding that gesture for the same time. By repeatedly holding and releasing the button, the player can repeat the full gesture as many times as they want. Similar to the gesture buttons, the tempo buttons can be held as long as a player desires and will increase or decrease the tempo of the performed gesture.

Translating the character's gesture into changes in emotional affect values is again inspired by all three of the theater practices we covered. This step in the interaction loop is analogous to the practices' insistence that physical movement reflect something about the internal state of the character and/or performer. Since we are giving the player direct control over the character's physicality, the way the character moves must have some effect on the kinds of emotional affects they are expressing. Each gesture can increase or decrease the value of each emotional affect the character

can express. This update may result in possibly conflicting emotional affect values being raised by the same gesture. For example, if the closed flow gesture results in the character fidgeting in their seat, both the emotional affects of anxiety and joy could be raised at equivalent rates as both can be expressed by fidgeting. Slowing the fidgeting down might make joy's value rise faster than anxiety and speeding it up might have the opposite effect should the player want to have the character be more definite in the way they are expressing themselves.

Our desire to use Expressionist to generate the lines of dialogue corresponding to the emotional affects performed by a character is primarily due to the system's tagging interface. The tagging system allows us to directly use the set of emotional affects with the largest values as part of the rules to generate lines of dialogue and use the three theater practices' common belief that physicality needs to be grounded by a character's or performer's internal state. Additionally, this means that to close the loop of interaction, the dialogue generated is a reflection of the character's physical actions and will help the player decide if they want to continue performing the same gesture, reminiscent of Zeami's belief that words inform the way a gesture is read.

Puppitor is built around updating a character's *affect vector*, a structure attaching floating point values to the set of emotional affects defined in a rule file and equilibrium values for when the player is choosing to do nothing. These updates are driven by a mapping of individual keys and buttons to the metaphorical actions of the three Stanislavsky energy states and the two additional tempos we previously identified.

In actually implementing the proposed system outline, the primary system

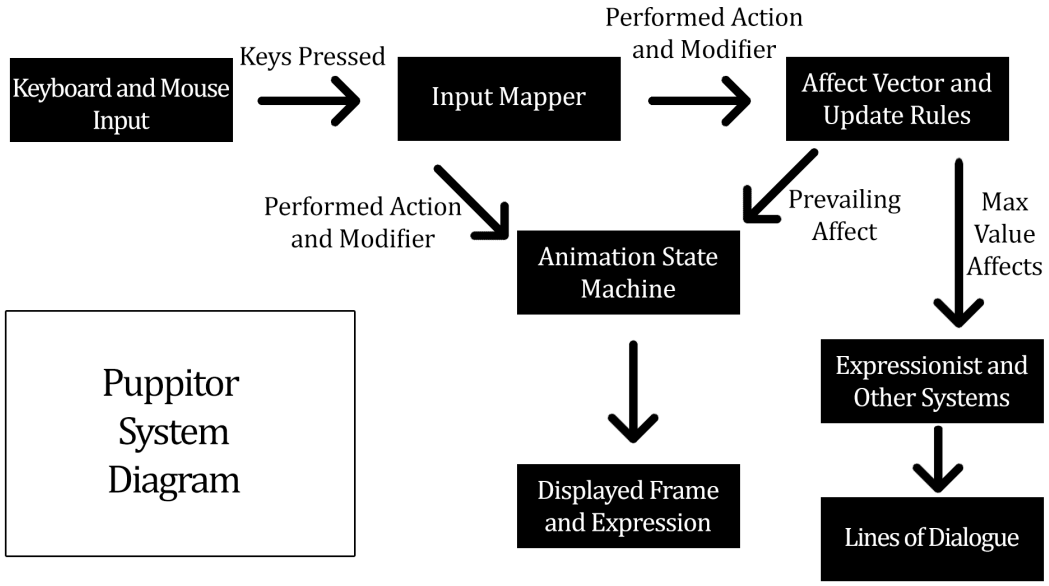


Figure 5.1: An overview of the flow of information between user input, Puppitor's modules, the display, and other systems. The primary information being passed between Puppitor's modules is the gesture state (interpreted from the keyboard and mouse input) and affect values stored in an affect vector.

interactions are:

- input mapping
- updating the affect vector
- animating the gesture
- choosing a prevailing affect

To help illustrate the flow of information through Puppitor's modules we use the following example of a player wanting to have their character express wonder.

## 5.3 Input Mapping

This module is responsible for detecting input and translating it into either an *action* (one of the three Stanislavsky energy states) or a *modifier* (one of the three Viewpoints tempos). To create a buffer between the raw input and the rest of the system, we have a two step process to actually send an action and modifier in a usable form to the other modules. The first step simply maps the keys being pressed to each action and modifier. The second enforces the rule that only one action and one modifier may be performed at any given time. This two step translation allows Puppitor to still read multiple key presses at once while producing a simplified representation of the state for use elsewhere.

These separate representations of the raw input and the simplified state representation also allow for a more structured approach to integrating the input mapping module into other code, in our case the Ren'Py visual novel engine [53]. All raw input can be handled in an event listener while everything that directly interacts with an affect vector can remain in a separate update loop and react accordingly. As part of our interest in authorability and clarity of function, our input mapping interface emphasizes the metaphor of actions and modifiers through the methods used to update the representation of the key state. This not only clarifies which elements of the input are being updated at any given moment, it also creates a consistent flow of information across the modules: the current energy and tempo state.

This module does not directly connect keys to affects, though through the actions and modifiers associated with each button we can understand how having a

character express wonder involves more than a simple button press. In our current ruleset, every action has an effect on all six of our emotional affects, meaning that even if the goal is to make the prevailing affect wonder, there will be other affect values increasing if the player takes the simplest approach to maximizing wonder: performing the projected energy gesture.

## 5.4 Updating the Affect Vector

The affect vector is where the translation from the gestures, mapped to the Stanislavsky energy states and the Viewpoints' tempos, to our set of emotional affects (discussed in section 7) is stored. The affect vector is updated every frame by looking up the rules associated with each individual affect based on which action and modifier are currently being performed. The default affect update value associated with the current action is multiplied with the value associated with the current modifier, then added to the affect vector's current value for the associated affect, then clamped to a specified floor and ceiling. The exception to this application of the rules, and the reason for the inclusion of equilibrium values, is to allow each affect value in the vector to trend towards a default point when the player is taking no action.

Our rationale for this choice is primarily to encourage points of rest while interacting through the interface by making the default action (of doing nothing) still have a noticeable effect on the state, encouraging some amount of interaction while also providing a baseline for the system. This equilibrium value also serves as a method of

characterization through Puppitor's design by allowing us to reify the baseline intensity of a character's emotional affects.

Our current ruleset allows each gesture, including our resting gesture, to update every value in the affect vector. While Puppitor does not require this approach, we wanted to build a system to allow players to explore the combinatorial space of expressivity created by giving them control over a character's physicality. For example, our current ruleset does not allow only a single affect to maximize its value when any single button is held down. Each affect has one action/gesture that will rapidly increase and rapidly decrease its value in the affect vector, with the other actions/gestures being less extreme. Our reason for having a single button press effect the entire affect vector was to avoid the dominance of single optimal strategies to maximizing particular affect values. This is our method of encouraging player expression through character acting in a similar way one-frame-links [2] can allow players to express their confidence in their own abilities [9]. In Puppitor's case, rather than dropping a combo, a player would express a set of emotional affects that might not line up as well with their intent.

In our example, holding the projected energy gesture alone would not guarantee the expression of the desired affect, as both love and hatred have higher default update values and desire is also positively influenced by the gesture. If the player chose to perform the projected energy gesture at a faster tempo, assuming enough of a difference between the three affects' values and the ceiling value, the value of wonder would eventually outpace the value of the other affects before all four of them reached the

[2]Continuations of a combo in a fighting game that must be executed on exactly one frame of the game's update (a window of 1/60th of a second) [10].

ceiling value.

Puppitor's general approach to its update cycle forces a degree of transition between prevailing affects due to its realtime update cycle and rarely completely stable affect values. Though the action and modifier interpreted from a player's input are applied instantly, the affect values, while also being responsive instantly, must take time to fully transition to reflecting the new gesture, and gestures must play their transitions from one gesture to another. This less than perfect responsiveness is reminiscent of the actions in fighting games having a certain degree of commitment and transition associated with them [31, 55]. It also shares the sentiment of George Henry Lewes's observations of Edmund Kean's acting performances in the late 19th century.

Lewes was enamoured with Kean's ability to express *subsiding emotions*, characterized by lingering hints of a prior emotional state after rapidly transitioning to a new one [30, 52]. While Puppitor is not designed to express the level of nuance a human actor is capable of, its affordances for transitioning from one gesture to another do capture, as a computational caricature, a degree of this style of acting through its incremental value updates. Additionally, as a function of both our approach to storing animations and our limited number of gestures compared to the number of emotional affects, Puppitor can create flashes of expressions during these transitional moments.

## 5.5 Animating the Gesture

Another difference between Puppitor and prior work is its usage of traditional 2D animation, compared to the 3D spaces of *Façade* and *Haunt 2* [32] or the primarily textual space of *Versu* [16]. Similar to our decision to use keyboard and mouse input rather than a text parser or physical gesture system, our usage of hand drawn animation was both born from the simpler technical challenges to overcome as well as the lengthy history of 2D animation and its techniques in the character-centric genre of fighting games [5, 26].

As with our input mapping and affect modules, the animation state machine we use stores lists of animation frames using our metaphor of actions and modifiers with an additional layer of emotional affects to store character expressions along with the full body animations. Individual frames are accessed by referencing the modifier and action they are associated with and providing the specific list index of that frame. Puppitor's set of indices for whichever frame it is currently displaying are stored and accessed using a similar modifier-action look-up as the animation frames. The primary reason for this unusually heavyweight solution for storing list indices is to allow our system to quickly switch between the three tempos of a given action, as all three tempos of an action are updated simultaneously, regardless of which one is currently being performed.

Our decision to use hand drawn animation and desire for a flexible system also necessitated Puppitor's animation state machine to allow for changes in animation frame-rate, semi-independent of the rest of the game's update rate. While our current usage of the animation state machine has used a fixed frame-rate for all animations,

the system features completely dynamic frame-rates to allow animators and designers to further express characters through their usage of Puppitor.

Another feature of our animation state machine is its delineator indices, once again stored and accessed through the modifier-action look-up paradigm. Each frame list has a corresponding set of delineators marking the ends of each phase of the animation. Our three segmented approach of giving each animation a "startup", "loop", and "recovery" phase was directly inspired by the "startup", "active", and "recovery" states in fighting games [3] [31, 55]. Where Puppitor's version differs is both in its usage of animation loops as part of a character's gesture and how it "cancels" the animation.

In a traditional fighting game, the part of the active or recovery state is what can be canceled into another move to create a combo. Puppitor doesn't have combos and has an animation loop in the middle of its list, so instead of canceling the recovery when the player switches the gesture they are having the character perform, if the character is in the looping part of their animation, they skip to the recovery part of the animation to allow for a middle ground between responsiveness and smooth animation.

[3] As an aside, the three states of animation both in Puppitor and numerous fighting games have a connection to Zeami's description of *jo* ("introduction" or "prelude"), *ha* ("development" or "exposition", literally "breaking"), and *kyū* ("climax" or "finale", literally "rapid") which he applies to the macro-level ordering of plays in a days' performance to actors' individual movements [51]. While it is unlikely that fighting game animations were directly inspired by Zeami's treatises, his work provides another lens for understanding the importance of each stage of a character's animation beyond purely mechanical concerns.

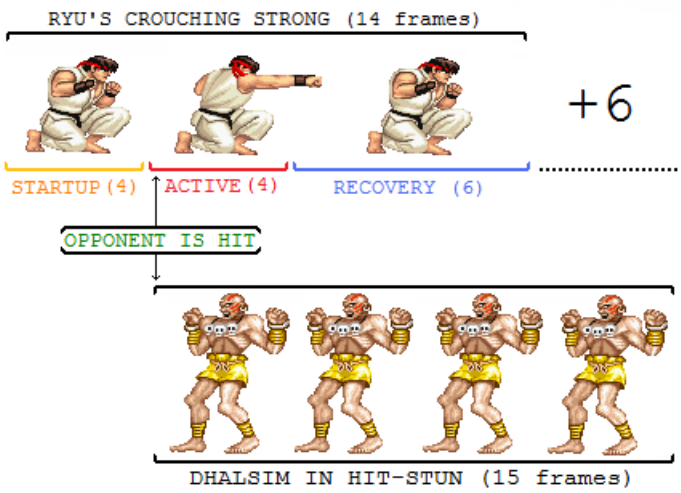


Figure 5.2: A breakdown of Ryu's crouching medium punch into the three main phases of a move in a fighting game: startup, active, and recovery and how these calculations factor into having an advantage when a move successfully connects with an opponent [24].

## 5.6 Choosing a Prevailing Affect

The process Puppitor uses to choose a single prevailing affect is a two part, transparent process. First, Puppitor collects all of the affects with the largest value from a given affect vector. There is always at least one affect in this set, but often there are more due to the ceiling value allowing multiple affects to reach the same value at different rates, which is why the second step exists. Puppitor will then pick the current (or only affect) in the affect vector. If there is more than one affect collected by the first step, and the current affect is not in that set, Puppitor will try to randomly pick an affect that is defined as connected to the currently prevailing affect by the ruleset in use. If no connected affects are in the gathered set, Puppitor will randomly pick from the set of disconnected affects.

The reason we describe this process as transparent is that the results from the first step of the process are accessible before the second step begins and can even be reasoned over or have additional steps added to the pipeline before passing the set of affects to the second step. As with the other main pieces of Puppitor's functionality, this decision came from our desire to keep the system flexible and allow for its processes to have additional steps inserted should a designer want to do something more complicated than what our default behavior allows. Additionally we built a single step wrapper around both steps should a designer not want to worry about maintaining the multistep process of choosing a single prevailing affect. For our eventual goal of using the affects chosen by Puppitor as part of a media experience, the open pipeline allows us to use the (possibly) more nuanced results from the first step for tasks like dialogue generation while using the results of the second step for tasks that benefit from clarity, like changing character expression and rendering text.

In our example of the expression of wonder, if the player managed to get the values for desire, hatred and love low enough to allow the value of wonder to have the largest value before all the affects hit the ceiling value, then wonder would remain the prevailing affect, given Puppitor's selection process, until its value dropped below the next largest value. If the player keeps performing the projected energy gesture, eventually desire, hatred, and love would reach the ceiling value. While this does not change the prevailing affect, it does cause Puppitor to produce a set of the possible affects containing wonder, desire, hatred, and love. This set of affects could then be used as finer grained input for a text generation system like Expressionist [54] to create

a line of dialogue that expresses, for example, love and hatred, in addition to wonder, giving the generated line more texture than simply generating dialogue to express a single emotional affect. The actual number of additional affects a line of dialogue could (programmatically) convey would be restricted by both the set produced by Puppitor as well as the number of affect tags available per terminal expansion (if Puppitor is indeed feeding tags to Expressionist).

## 5.7 Rules for Emotional Affects

A ruleset for Puppitor is expressed in a JSON file and then loaded into the part of the system responsible for updating the affect vector. An entry in the ruleset is defined as the named affect containing:

- the same set of actions and modifiers as the input mapping associated with floating point values
- a list of connected affects
- the equilibrium value of the emotional affect

The floating point values attached to actions are the update rate of the affect they are associated with per single call to the part of Puppitor's affect vector update functionality. The floating point values associated with the modifiers are the values multiplied with the action values of the affect. The specific multiplier and value is applied based on whichever action and modifier Puppitor interprets from a player's

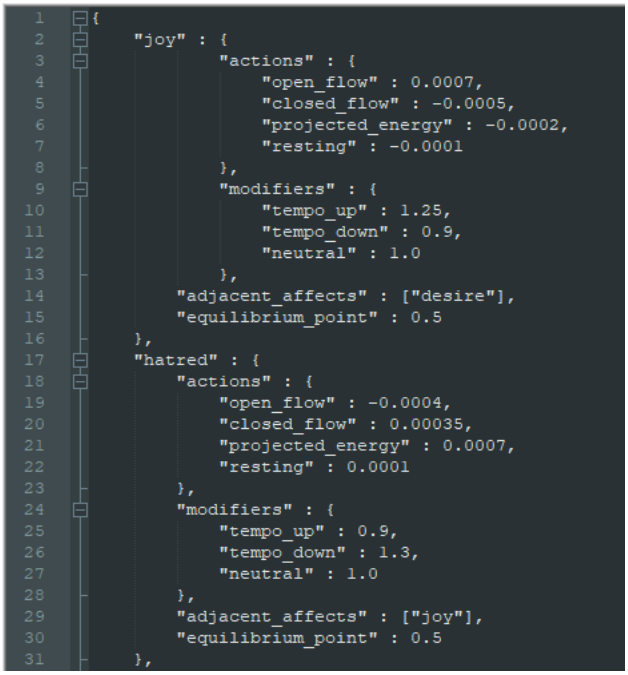


Figure 5.3: Excerpt of a rule file using Descartes' six universal passions as the set of affects. Each affect tracks how each possible energy state will update its value, how each tempo modifier will alter that update value, what (if any) other affects are connected to it, and the equilibrium point associated with the affect (for the target value for the affect value to trend towards while performing the resting gesture).

input. As Puppitor's update functionality clamps the affect vector's values and simply multiplies then adds the values from the ruleset file to the values in the affect vector, the ruleset file is where all the specific behavior is defined.

The connected affects element of Puppitor's affect definition exists specifically to give designers a way of partially reifying (and creating some consistency in) how a character's emotions feed into each other as part of the process of how the system chooses a prevailing affect. As we discussed in section 6, to choose a prevailing affect, Puppitor prioritizes deterministic approaches to characters changing their expressed emotional affect. Our designer-defined approach to the semi-deterministic choices Puppitor makes

when it has no other options allow for more consistent characterization through the ruleset.

The equilibrium values, like the connected affects, allow for designer defined characterization of what an individual character's default expressed affect should be. Our goal with the design of the ruleset files is to allow a relatively high degree of designer control over the way characters express themselves through the rest of the system, and by extension any media experience using Puppitor. In this light we view Puppitor as an approach to expressing characters through systems and interface, again taking inspiration from fighting games' connection between their interface and their characters [8]. Additionally, while our current usage of Puppitor only uses a single ruleset per character, the system supports creating any number of rulesests, meaning it is possible to have the rules for updating a character's affect vector change based on their currently prevailing affect, as an example. In this light, our authoring approach to rulesets has been using them as a way to reify a character's personality in Puppitor. As an example, a character with a more mercurial personality might have relatively high default affect update values to allow them to rapidly change their prevailing emotional affect in a similar way to the actor David Garrick would show off the range of the passions [52].

By putting our rulesets into an easily modified, authorable format like JSON, the core system of Puppitor itself is not tied to a specific set of emotional affects, or even specific actions and modifiers (though it does enforce the distinction between actions and modifiers at a core level and requires consistent terminology and metaphors across

its modules). Instead it is a framework for creating and applying rulesets of emotional expression using its metaphor of actions and modifiers.

The set of baseline emotional affects we picked for the current version of Puppitor are the six universal passions as defined by Descartes: wonder, love, hatred, desire, joy, and sadness [52]. Roach characterizes Descartes as a "machine-soul dualist" interested in how the passions illuminated the relationship between spirit and matter. For Descartes, the workings of the soul were responsible for calling forth one of the passions then animating the body into performing the passion appropriately. While Descartes' view of emotions is inaccurate when compared with modern psychology and physiology, his simplified metaphor for how humans express the passions, similar to Zeami's guides for actors [51], makes for an easier approach to implementing a system focused around expressivity and computational caricature rather than modeling completely naturalistic behavior.

## 5.8 Character Expression

To even begin approaching the task of dynamically animating characters in a 2D environment, we found ourselves referencing numerous design patterns from fighting games and Japanese animation. These notably include limited animation, a keyframe first approach to a full body animation, a lower animation frame-rate than the game's update loop, and changing the timing of frames in animation lists as a way to change a character's tempo. Another reason we draw heavy influence from fighting games is their

restricted camera angles, even found in 3D games like *Tekken*, lending some similarities to theatrical staging and the limited view of the audience in many traditional Western theaters, particularly those with a proscenium arch.[4]

Limited animation is a method of reducing the number of frames in an animation (often employed by the Japanese animation industry), thus reducing the number of drawings required to create a complete animation [44]. As we are hand drawing each frame of animation for use in Puppitor, our choice to employ this is not only a stylistic one, it is a practical one given the extremely small team size working on the system. This approach to animation is also aided by the dynamic animation frame-rate in the animation state machine by allowing the designer and animator more fine-grained control over the display of the individual frames. Additionally, the limited number of frames helps cut down on memory usage, as every frame must be loaded into memory to be used in realtime smoothly.

Due to the way we incorporate dynamic character expressions, each affect has its own frame list per modifier and action. While this is not the most efficient method of storing character expressions and gestures, it allows us to use the same approach (albeit with an additional layer of specificity) to updating the affect vector. This method of organization also means all of a character's drawings are stored in a single structure, making the management of the transitions between animations relatively simple to integrate with display code in other programs like Ren'Py.

The limited animation approach also puts more focus on our key poses in a

[4] A decorative arch between the stage and the auditorium, emphasizing the existence of the fourth wall.


Figure 5.4: Rough key pose sketches for one of the characters in the game being built using Puppitor. The poses marked 1 and 3 are the base poses for the looping sections of, in this case, the resting and closed flow gestures. The middle pose is a rough in between frame for use as part of the transition between the resting and closed flow gestures.

similar manner to fighting games [5, 26]. There is an assumption of a relatively higher degree of responsiveness from a player character when compared to NPCs in digital games [5]. While we are not making a combat focused game, given our interaction paradigm, there is a similar expectation of responsiveness from the characters implemented in Puppitor, and our emphasis with the animations is on three key poses per full animation cycle: the character's resting pose, their primary pose for looping one of our three other gestures, and a key pose they use when transitioning to another gesture (entailing them briefly returning to their resting pose).

We approach the process of implementing a gesture in a similar way to how the animators of *Skullgirls* [50] approached animating their characters' moves [5]. First we sketch our key poses and add them to the animation state machine to allow us to solidify

the base timings for each stage of the gesture's animation. We then add a number of in-between frames necessary to smooth out the key pose transitions and create the necessary level of energy before finally coloring the full animation and re-implementing it in Ren'Py and Puppitor.

Again as an extension of our small team size and the affordances we built into Puppitor's animation state machine, the changes in tempo for each gesture are primarily built around altering the timings of frames to convey changing speeds. This again reduces the number of frames needing to be drawn for any given animation while also allowing for relatively smooth transitions between animations in 2D, as we do not have any other animation blending technologies currently implemented for use with Puppitor. As Swink describes in *Game Feel*, players do not distinguish between the polish[5] and simulation elements of an experience [62]. Swink's observations—along with the various discussions of animation practices in fighting games [5, 26, 44]—are further support for our approach to creating different tempos of animated gestures and the design of Puppitor's animation state machine and frame lists.

While the animated gestures are our primary source of feedback for communicating what a button press does, feedback about the less immediate effects of a gesture comes in the form of facial expressions and text rendering. The only difference in animation frames of a specific energy state and tempo across all six of our emotional affects is the character's expression. This expression is determined by Puppitor's method of choosing a prevailing affect based on a character's affect vector (as discussed in section

[5]Polish is any effect such as improved quality of visual and auditory assets, that improves a player's perception of an experience without changing the underlying simulation [62].

2) and adds specificity to our more general purpose gestures without the need for entirely new animations. To add even more specificity to the gesture, we alter the speed, font size, and capitalization of a line of dialogue, using the affordances built into Ren'Py, based on the prevailing affect when the line needs to be displayed. Not only can this approach suggest a particular reading of a line—because of the asynchronous connection between animations and displaying lines of dialogue, Puppitor is capable of dynamically creating moments of subsiding emotion that a player can choose to accentuate.

# Chapter 6

# Future Work and Conclusion

## 6.1 Future Work

As mentioned throughout this thesis, we plan to continue building towards a digital model of physical and spatial acting using the theory and practices we have discussed as well as releasing a finished videogame featuring Puppitor. Once at least one of these interactive drama games is released, we hope to tackle the problem of creating a narrative nav-mesh using a similar design strategy to the rather open and flexible nature of Puppitor: namely marking up space using a rule file to translate movement and position into additional affect update values. These spatial rule files will likely take heavy inspiration from the Viewpoints, particularly those relating to architecture, spatial relationship, and topography. Puppitor was built to be modular, not only internally, but also to allow other systems to be used in parallel with it; the narrative nav-mesh system we hope to build will also help test this design.

There is also the question of tailoring lines of dialogue more than simply changing the line reading. In *Towards an Expressive Input for Character Dialogue in Digital Games*, we proposed that the system we built to model physical acting would be attached to James Ryan's context-free grammar system Expressionist [54] and even included such a link in the system diagram found in this thesis. We still plan to make this connection (and Puppitor was built with this link in mind). Due to the nature of writing dialogue, to make fully procedural conversations, an additional system focused around the internal structure of beats as well as tracking the outcomes of each beat would need to be developed. With this in mind, as with the current plans for integrating Puppitor into a released digital game, it makes sense to first develop the structure and rules for generating dialogue with Expressionist and then attempt to turn those rules and approach into a computational system.

An additional project that we found to be beyond the scope of this thesis is building artificial intelligence systems to interface with Puppitor (and eventually the other systems). Our primary concern with Puppitor was to build a system for human players to interact with but because if its flexible design and decoupling of input with updating emotional affect values, we believe the system can also serve as a test-bed for creating NPCs that have some understanding of physical acting (even if it is only in the simplified space Puppitor creates). While creating AI systems that actually have a degree of understanding of Puppitor's rules is still an open problem that we will attempt to address at a later date, the development of a digital game using Puppitor will provide a step towards this goal. To give the player an acting partner, even if it is only one who

reacts semi-randomly, we will need to allow the computer to interface with Puppitor in a similar way to the player. It remains to be seen how easy integrating well tested AI approaches like MCTS or A* search is to such an interface but we hope to at least provide a foundation for such integrations with the development of a digital game.

## 6.2 Conclusion

While Puppitor's modules produce relatively intricate behavior from a small set of actions and effects, we hope the authorability of its rulesets and interface metaphors allows for a wider variety of experiences possible than what we as the system designers initially built the system for. With this paper and the eventual release of a digital game using Puppitor (and the system itself), we hope to inspire more computational caricature-like modes of character interaction beyond combat. Puppitor may take a wealth of inspiration from combat focused games but it uses those inspirations and techniques to build towards a playable model of conversation. While this paper was being written, Mark Brown of Game Maker's Toolkit released a video asking the question "Can we make talking as much fun as shooting?" [19], highlighting our motivation behind the creation of Puppitor. With that said, the argument is still rooted around picking dialogue or dialogue like actions as the primary interaction. In contrast, Puppitor takes the fluidity of the embodied physical response of fighting games and combines it with the emotional interaction of a conversation. We feel the timely nature of Brown's question only further emphasizes the relevance of our approach to both academic and industry

focused designers.

The core structure and organization of Puppitor are necessarily immutable while its rulesets, animation system, and even interface metaphors are all easily editable. The organizational structure and higher level metaphors of action (from Stanislavsky) and modifier (from the Viewpoints) were what we found to be the minimum amount of structure necessary to keep each module of Puppitor (input mapping, the affect vector and its associated uses, and the animation state machine) using the same organizational structure. In comparison, the specific mappings we use for actions (open flow, closed flow, projected, and resting) and modifiers (tempo up, tempo down, neutral) were implementation details rather than necessary structure. As a result, we decided that our small set of actions and modifiers did not need to be the only set Puppitor was capable of using. In this way, the system is an interface framework that allows designers to decide what metaphors for interaction, within Puppitor's structure, they find most useful in the context of the experience they are building.

Though Puppitor is still relatively early in its development and requires further tuning and testing to build a complete media experience, we view the system as a solid first step on the path to creating the kind of player driven character acting and scene construction we propose in this thesis. Even by itself, Puppitor allows the creation of realtime, physicality focused, affective play. Additionally we feel Puppitor provides a starting point for the further exploration of building gameplay around character physicality and acting in digital games as well as providing another avenue of inquiry into the nature of the space of interactive drama.

# Bibliography


[1] James Bierman. Aristotle or Else.

[2] Anne Bogart and Tina Landau. *The Viewpoints Book: A Practical Guide to Viewpoints and Composition*. Theatre Communications Group, 2005.

[3] Capcom. Street Fighter, 1987.

[4] Capcom. Street Fighter II: The World Warrior, 1991.

[5] Mariel Cartwright. Making Fluid and Powerful Animations For Skullgirls. GDC.

[6] Michael Chemers. *Ghost Light*. Sothern Illinois UP, 2010.

[7] Kate Compton and Michael Mateas. Casual Creators. In *ICCC*, pages 228–235, 2015.

[8] Core-A Gaming. Analysis: How to Pick a Character.

[9] Core-A Gaming. Analysis: The Consequences of Reducing the Skill Gap.

[10] Core-A Gaming. Analysis: Why Fighting Games Are Hard.

[11] Steven Dow, Manish Mehta, Ellie Harmon, Blair MacIntyre, and Michael Mateas. Presence and engagement in an interactive drama. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1475–1484. ACM, 2007.

[12] eAdventureUCM. La dama boba (the foolish lady). el juego (the game), 2013.

[13] Bioware Edmonton. Dragon age: Inquisition, 2014.

[14] Ekaterina Tsvetkova. Chekhov's The Cherry Orchard, Renata Litvinova as Ranevskaya and Andrey Smolyakov as Lopakhin.

[15] Respawn Entertainment. Titanfall 2, 2016.

[16] Richard Evans and Emily Short. Versu - a Simulationist Storytelling System. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(2):113–130, 2014.

[17] Gonzalo Frasca. Videogames of the oppressed: Videogames as a means for critical thinking and debate. Master's thesis, School of Literature, communication, and culture, Georgia Institute of Technology, 2001.

[18] French Bread. Under Night In-Birth, 2012.

[19] Game Maker's Toolkit. Can We Make Talking as Much Fun as Shooting? — Game Maker's Toolkit.

[20] Irrational Games. Bioshock Infinite, 2013.

[21] Tea-Powered Games. Elemental Flow, TBD.

[22] Todd L Harper. *The art of war: Fighting games, performativity, and social game play.* PhD thesis, Ohio University, 2010.

[23] Mikhail Jacob, Alexander Zook, and Brian Magerko. Viewpoints AI: Procedurally Representing and Reasoning about Gestures. 08 2013.

[24] Jett. Universal Fighting Game Guide: How to Read Frame Data.

[25] Nick Junius, Michael Mateas, and Noah Wardrip-Fruin. Towards Expressive Input for Character Dialogue in Digital Games. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019.

[26] Toshiyuki Kamei. The Art Direction of Street Fighter V: The Role of Art in Fighting Games. GDC.

[27] Rachel Lee Knickmeyer and Michael Mateas. Preliminary evaluation of the interactive drama facade. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pages 1549–1552. ACM, 2005.

[28] Brenda Laurel. *Computers as Theater.* Addison-Wesley Professional, 2nd. edition, 2013.

[29] Irina Levin and Igor Levin. *The Stanislavsky Secret.* Colorado: Meriwether Publishing, 2002.

[30] George Henry Lewes. *On actors and the art of acting*, volume 1533. London Smith, Elder, 1875.

[31] Feiyu Lu, Kaito Yamamoto, Luis H Nomura, Syunsuke Mizuno, YoungMin Lee, and Ruck Thawonmas. Fighting game artificial intelligence competition platform. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, pages 320–323. IEEE, 2013.

[32] Brian Magerko, John E Laird, Mazin Assanie, Alex Kerfoot, Devvan Stokes, et al. AI characters and directors for interactive computer games. In *AAAI*, pages 877–883, 2004.

[33] Borja Manero, Clara Fernández-Vara, and Baltasar Fernández-Manjón. Stanislavky's System as a Game Design Method: A Case Study. In *DiGRA Conference*. Citeseer, 2013.

[34] Michael Mateas. A preliminary poetics for interactive drama and games. *Digital Creativity*, 12(3):140–152, 2001.

[35] Michael Mateas and Andrew Stern. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[36] Michael Mateas and Andrew Stern. Façade, 2005.

[37] Michael Mateas and Andrew Stern. Structuring Content in the Façade Interactive Drama Architecture. In *AIIDE*, pages 93–98, 2005.

[38] Josh McCoy, Mike Treanor, Ben Samuel, Brandon Tearse, Michael Mateas, and Noah Wardrip-Fruin. Authoring game-based interactive narrative using social games and comme il faut. In *Proceedings of the 4th International Conference &*

*Festival of the Electronic Literature Organization: Archive & Innovate*. Citeseer, 2010.

[39] Josh McCoy, Mike Treanor, Ben Samuel, Noah Wardrip-Fruin, and Michael Mateas. Prom week, 2012.

[40] Joshua McCoy and Michael Mateas. The Computation of Self in Everyday Life: A Dramaturgical Approach for Socially Competent Agents. In *AAAI Spring Symposium: Intelligent Narrative Technologies II*, pages 75–82, 2009.

[41] Joshua McCoy, Michael Mateas, and Noah Wardrip-Fruin. Comme il faut: A system for simulating social games between autonomous characters. 2009.

[42] Joshua Allen McCoy. *All the World's A Stage: A Playable Model of Social Interaction Inspired by Dramaturgical Analysis*. PhD thesis, University of California, Santa Cruz, Santa Cruz, CA, June 2012.

[43] Michael Brosilow. SITI Workshops Header.

[44] Junya C Motomura. GuiltyGearXrd's Art Style : The X Factor Between 2D and 3D. GDC.

[45] Namco. Tekken, 1994.

[46] Interplay Productions. Fallout: A Post Nuclear Role Playing Game, 1997.

[47] Monolith Productions. F.E.A.R. First Encounter Assault Recon, 2005.

[48] CD Projekt Red. The Witcher 3: Wild Hunt, 2015.

[49] Aaron A. Reed. *Changeful Tales: Design-Driven Approaches Toward More Expressive Storygames.* PhD thesis, University of California, Santa Cruz, Santa Cruz, CA, June 2017.

[50] Reverge Labs. Skullgirls, 2012.

[51] J Thomas Rimer, Masakazu Yamazaki, et al. *On the Art of the Nō Drama: The Major Treatises of Zeami; Translated by J. Thomas Rimer, Yamazaki Masakazu.* Princeton University Press, 1984.

[52] Joseph R Roach. *The player's passion: studies in the science of acting.* University of Michigan Press, 1993.

[53] Tom Rothamel. Ren'Py, 2004.

[54] James Ryan, Ethan Seither, Michael Mateas, and Noah Wardrip-Fruin. Expressionist: An authoring tool for in-game text generation. In *International Conference on Interactive Digital Storytelling*, pages 221–233. Springer, 2016.

[55] Ryukenden and C-Royd. Ryu (3S).

[56] Serdar Sali, Noah Wardrip-Fruin, Steven Dow, Michael Mateas, Sri Kurniawan, Aaron A Reed, and Ronald Liu. Playing with words: from intuition to evaluation of game dialogue interfaces. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 179–186. ACM, 2010.

[57] Ben Samuel. *Crafting Stories Through Play.* PhD thesis, University of California, Santa Cruz, Santa Cruz, CA, December 2016.

[58] Daniel G Shapiro, Joshua McCoy, April Grow, Ben Samuel, Andrew Stern, Reid Swanson, Mike Treanor, and Michael Mateas. Creating Playable Social Experiences through Whole-Body Interaction with Virtual Characters. In *AIIDE*, 2013.

[59] Adam M Smith and Michael Mateas. Computational caricatures: Probing the game design process with ai. In *Workshops at the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[60] Night School Studio. Oxenfree, 2016.

[61] David Surman. Pleasure, spectacle and reward in capcom's street fighter series david surman. *Videogame, Player, Text*, page 204, 2007.

[62] Steve Swink. *Game feel: a game designer's guide to virtual sensation*. CRC Press, 2008.

[63] Binary Systems. Starflight, 1986.

[64] Joshua Tanenbaum. Being in the story: readerly pleasure, acting theory, and performing a role. In *International Conference on Interactive Digital Storytelling*, pages 55–66. Springer, 2011.

[65] Tommy Thompson. The AI of BioShock Infinite's Elizabeth — AI and Games, 2017.

[66] Noah Wardrip-Fruin. *Expressive Processing: Digital fictions, computer games, and software studies*. MIT press, 2009.

[67] YUKO_M. Noh theater stage.

# Appendix A

# Input Mapping Module

```
#
# Animation_Structure contains nested dictionaries for storing frames of
    ↪ animation and where in the animation loop the simulation is
# the default arguments of Animation_Structure use theatrical terms for
    ↪ organization
#
# because of the way modifier_list and action_list are used in the
    ↪ construction of the nested dictionaries
# the order of the lists passed only determines the order of index
    ↪ arguments needed to access the values stored
# in either self.animation_frame_lists or self.current_frames
#
# NOTE: THE CURRENT DEFAULT OF final_frame_index IS A PLACEHOLDER MAKE
    ↪ SURE TO CHANGE IN PRODUCTION
# final_frame_index is strictly a bookkeeping variable and necessary for
    ↪ update_displayed_frame() to work properly
#
# frame_rate_delay is the number of frames to hold on before a new frame
    ↪ of animation is selected
# the default values assume a 60Hz rate of calling update_displayed_frame
    ↪ () and a desired animation rate of 10FPS
#
class Animation_Structure:
    def __init__(self, frame_rate_delay = 5, modifier_list = ['tempo_up',
        ↪  'tempo_down', 'neutral'], action_list = ['open_flow', '
        ↪ closed_flow', 'projected_energy', 'resting'], affect_list = ['
        ↪ joy', 'hatred', 'sadness', 'wonder', 'love', 'desire']):

        # self.animation_frame_lists is a dictionary that uses modifiers
            ↪ to access actions to access individual frame lists
```

```
self.animation_frame_lists = {}
for modifier in modifier_list:
    self.animation_frame_lists[modifier] = {}
for modifier in self.animation_frame_lists:
    for action in action_list:
        self.animation_frame_lists[modifier][action] = {}
for modifier in self.animation_frame_lists:
    for action in action_list:
        for affect in affect_list:
            self.animation_frame_lists[modifier][action][affect]
                ↪ = []

# used to track the frame index of all animation lists
self.current_frames = {}
for modifier in modifier_list:
    self.current_frames[modifier] = {}
for modifier in self.current_frames:
    for action in action_list:
        self.current_frames[modifier][action] = 0

# used to store the end points of each part of individual
    ↪ animation frame lists
# to update the values to allow the update_displayed_frame()
    ↪ function to properly work:
#
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['startup'] = 6
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['loop'] = 8
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['final frame'] = len(<
    ↪ Animation_Structure instance>.animation_frame_lists['<
    ↪ modifier>']['<action>']['<affect>']) - 1
#
# the final frame line in the example assumes the desire is to
    ↪ have the final frame delineator as the last frame in the
    ↪ corresponding <Animation_Structure instance>
    ↪ animation_frame_list
# Animation_Structure assumes that 'startup' value < 'loop' value
    ↪  < 'final frame' value
#
self.frame_index_delineators = {}
for modifier in modifier_list:
    self.frame_index_delineators[modifier] = {}
for modifier in self.frame_index_delineators:
    for action in action_list:
        self.frame_index_delineators[modifier][action] = {'
            ↪ startup': 0, 'loop': 1, 'final␣frame': 2} #
            ↪ default values to be swapped out after the
            ↪ Animation_Structure is built (use
            ↪ load_animation_list())

self.current_displayed_frame = None
```

```
        self.current_animation_action = action_list[-1]
        # used for switching over to a new animation corresponding to the
            ↪  current action the player is taking once the previous
            ↪ animation has finished
        self.reached_end_of_frame_list = False
        # used for setting the frame rate of animations stored in this
            ↪ structure
        # default values assume an update rate of 60Hz creating an
            ↪ animation frame rate of 10FPS
        self.frame_rate_delay = frame_rate_delay
        # internal counter for advancing the frame rate delay
        self.frame_rate_delay_count = 0

    # used for putting full animations into the animation structure
    # frame_list is the set of animation frames to be added to the
        ↪ initialized animation_frame_lists
    # modifier, action, affect all correspond to where in the
        ↪ animation_frame_lists the animation frames will be stored (and
        ↪  by default should be strings)
    # startup_end_frame_index, loop_end_frame_index, final_frame_index
        ↪ are integer values used to mark off the different sections of
        ↪ the animation
    def load_animation_list(self, frame_list, modifier, action, affect,
        ↪ startup_end_frame_index, loop_end_frame_index,
        ↪ final_frame_index):
        for animation_frame in frame_list:
            self.animation_frame_lists[modifier][action][affect].append(
                ↪ animation_frame)
        self.frame_index_delineators[modifier][action]['startup'] =
            ↪ startup_end_frame_index
        self.frame_index_delineators[modifier][action]['loop'] =
            ↪ loop_end_frame_index
        self.frame_index_delineators[modifier][action]['final␣frame'] =
            ↪ final_frame_index
        return

    # should be used to access the currently displayed frame
    # should be returning a Sprite if used with RenPy
    def get_displayed_frame(self):
        return self.current_displayed_frame

    # returns the next frame in an animation sequence as well as the
        ↪ previous frame
    # this is primarily a book keeping function meant to allow a
        ↪ persistent
    # should be returning a Sprite if used with RenPy
    def update_displayed_frame(self, modifier, action, affect):
        if self.frame_rate_delay_count < self.frame_rate_delay:
            self.frame_rate_delay_count += 1
        else:
            #self.displayed_frame_data = (modifier, self.
                ↪ current_animation_action, affect, self.current_frames[
```

```
    ↪ modifier][self.current_animation_action])
for mod in self.animation_frame_lists:
    # skip to returning to resting if the player has changed
        ↪ the energy state
    if self.current_animation_action != action:
        if self.current_frames[mod][self.
            ↪ current_animation_action] < self.
            ↪ frame_index_delineators[mod][self.
            ↪ current_animation_action]['loop']:
            self.current_frames[mod][self.
                ↪ current_animation_action] = self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop'] + 1
        else:
            self.current_frames[mod][self.
                ↪ current_animation_action] += 1
            if self.current_frames[mod][self.
                ↪ current_animation_action] > self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['final␣frame']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] = 0
                self.reached_end_of_frame_list = True
    # startup animation
    elif self.current_frames[mod][self.
        ↪ current_animation_action] < self.
        ↪ frame_index_delineators[mod][self.
        ↪ current_animation_action]['startup']:
        self.current_frames[mod][self.
            ↪ current_animation_action] += 1
    # loop animation
    elif self.current_frames[mod][self.
        ↪ current_animation_action] <= self.
        ↪ frame_index_delineators[mod][self.
        ↪ current_animation_action]['loop']:
        if self.current_animation_action != action:
            self.current_frames[mod][self.
                ↪ current_animation_action] = self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop'] + 1
        else:
            self.current_frames[mod][self.
                ↪ current_animation_action] += 1
            if self.current_frames[mod][self.
                ↪ current_animation_action] > self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] = self.
                    ↪ frame_index_delineators[mod][self.
                    ↪ current_animation_action]['startup'] +
                    ↪  1  # return to first frame of the
                    ↪ loop
```

```
            # return to default animation
            elif self.current_frames[mod][self.
                ↪ current_animation_action] <= self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['final␣frame']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] += 1
                if self.current_frames[mod][self.
                    ↪ current_animation_action] > self.
                    ↪ frame_index_delineators[mod][self.
                    ↪ current_animation_action]['final␣frame']:
                    self.current_frames[mod][self.
                        ↪ current_animation_action] = 0
                    self.reached_end_of_frame_list = True

        if self.reached_end_of_frame_list:
            self.current_animation_action = action
            self.reached_end_of_frame_list = False
        else:
            self.current_animation_action = self.
                ↪ current_animation_action
        self.frame_rate_delay_count = 0
    self.current_displayed_frame = self.animation_frame_lists[
        ↪ modifier][self.current_animation_action][affect][self.
        ↪ current_frames[modifier][self.current_animation_action]]
    return(self.animation_frame_lists[modifier][self.
        ↪ current_animation_action][affect][self.current_frames[
        ↪ modifier][self.current_animation_action]])
```

# Appendix B

# Affect Update Module

```
#
# Affecter is a wrapper around a JSON object based dictionary of affects
    ↪ (see contents of the affect_rules directory for formatting details
    ↪ )
#
# By default Affecter clamps the values of an Affect_Vector in the range
    ↪ of 0.0 to 1.0 and uses theatrical terminology, consistent with
# the default keys in gesture_keys.py inside of the actual_action_states
    ↪ dictionary in the Gesture_Interface class
#
# NOTE: WHICHEVER ACTION IS SPECIFIED AS equilibrium_action MUST HAVE A
    ↪ POSITIVE FLOAT VALUE ASSOCIATED WITH IT
# OTHERWISE THE LOGIC MOVING THE AFFECT VALUES TOWARDS THE
    ↪ equilibrium_value WILL NOT FUNCTION PROPERLY
#
import json
import random
import math

class Gesture_Affecter:
    # affect_rules_name must take the form of '_filename_.json'
    # affect_rules_directory must take the form of '_directoryname_/
        ↪ _directoryname_/.../_directoryname_/'
    def __init__(self, affect_rules_name, affect_rules_directory,
        ↪ affect_floor = 0.0, affect_ceiling = 1.0, equilibrium_action =
        ↪  'resting'):
        with open(affect_rules_directory + affect_rules_name) as entry:
            # affect_rules are organized as ['affect']['type']['action']
            # NOTE 'type' is either 'actions' or 'modifiers'
            # or ['affect']['adjacent_affects'] to get the adjacency list
```

```
            self.affect_rules = json.load(entry)
        self.floor_value = affect_floor
        self.ceil_value = affect_ceiling
        self.equilibrium_action = equilibrium_action
        self.current_affect = self.affect_rules.keys()[0] # TODO do
            ↪ something more consistent and robust

    # for use clamping the updated affect values between a given
        ↪ floor_value and ceil_value
    def _update_and_clamp_values(self, affect_value, affect_update_value,
        ↪  floor_value, ceil_value):
        return max(min(affect_value + affect_update_value, ceil_value),
            ↪ floor_value)

    # affect_vector is an Affect_Vector specified by class Affect_Vector
        ↪ in this file
    # the floats correspond to the strength of the expressed affect
    # current_action corresponds to the standard action expressed by a
        ↪ Gesture_Interface instance in its actual_action_states
    # NOTE: clamps affect values between floor_value and ceil_value
    # NOTE: while performing the equilibrium_action the affect values
        ↪ will move toward the equilibrium_value of the given
        ↪ affect_vector
    def update_affect(self, affect_vector, current_action,
        ↪ current_modifier):
        for affect in affect_vector.affects:
            current_action_update_value = self.affect_rules[affect]['
                ↪ actions'][current_action]
            current_modifier_update_value = self.affect_rules[affect]['
                ↪ modifiers'][current_modifier]
            current_equilibrium_value = self.affect_rules[affect]['
                ↪ equilibrium_point']

            # move towards resting value specified in affect_vector when
                ↪ updating the action associated with the the '
                ↪ equilibrium_action'
            if current_action == self.equilibrium_action:
                if affect_vector.affects[affect] >
                    ↪ current_equilibrium_value:
                    affect_vector.affects[affect] = self.
                        ↪ _update_and_clamp_values(affect_vector.affects
                        ↪ [affect], -1 * abs(
                        ↪ current_modifier_update_value *
                        ↪ current_action_update_value),
                        ↪ current_equilibrium_value, self.ceil_value)
                elif affect_vector.affects[affect] <
                    ↪ current_equilibrium_value:
                    affect_vector.affects[affect] = self.
                        ↪ _update_and_clamp_values(affect_vector.affects
                        ↪ [affect], abs(current_modifier_update_value *
                        ↪ current_action_update_value), self.floor_value
                        ↪ , current_equilibrium_value)
                else:
```

```
                continue
            else:
                affect_vector.affects[affect] = self.
                    ↪ _update_and_clamp_values(affect_vector.affects[
                    ↪ affect], current_modifier_update_value *
                    ↪ current_action_update_value, self.floor_value,
                    ↪ self.ceil_value)
        return

    # affect_vector must be an Affect_Vector
    # returns a list of the affects with the highest strength of
        ↪ expression in the given affect_vector
    # allowable_error is used for dealing with the approximate value of
        ↪ floats
    def get_possible_affects(self, affect_vector, allowable_error =
        ↪ 0.00000001):
        prevailing_affects = []

        for current_affect in affect_vector.affects:
            if not prevailing_affects:
                prevailing_affects.append(current_affect)
            elif affect_vector.affects[prevailing_affects[0]] <
                ↪ affect_vector.affects[current_affect]:
                prevailing_affects = []
                prevailing_affects.append(current_affect)
            # check if the affect magnitudes are approximately equal
            elif abs(affect_vector.affects[prevailing_affects[0]] -
                ↪ affect_vector.affects[current_affect]) <
                ↪ allowable_error:
                prevailing_affects.append(current_affect)
            #print(current_affect, affect_vector.affects[current_affect])
        #print(prevailing_affects)
        return prevailing_affects

    # chooses the next current affect
    # possible_affects must be a list of strings of affects defined in
        ↪ the .json file loaded into the Affecter instance
    # possible_affects can be generated using the get_possible_affects()
        ↪ function
    # the choice logic is as follows:
    #   pick the only available affect
    #   if there is more than one and the current_affect is in the set of
        ↪  possible_affects pick it
    #   if the current_affect is not in the set but there is at least one
        ↪  affect connected to the current affect, pick from that subset
    #   otherwise randomly pick from the disconnected set of possible
        ↪ affects
    def choose_prevailing_affect(self, possible_affects):

        connected_affects = []

        if len(possible_affects) == 1:
            self.current_affect = possible_affects[0]
```

```
            return self.current_affect

        if self.current_affect in possible_affects:
            return self.current_affect

        for affect in possible_affects:
            if affect in self.affect_rules[self.current_affect]['
                ↪ adjacent_affects']:
                connected_affects.append(affect)

        if connected_affects:
            self.current_affect = random.choice(connected_affects)
            return self.current_affect
        else:
            self.current_affect = random.choice(possible_affects)
            return self.current_affect

    # wrapper function around the get_possible_affects() to
        ↪ choose_prevailing_affect() pipeline to allow for easier, more
        ↪ fixed integration into other code
    # NOTE: this function is not intended to supercede the useage of both
        ↪  get_possible_affects() and choose_prevailing_affect()
    #       it is here for convenience and if the default behavior of
        ↪ immediately using the list created by get_possible_affects()
        ↪ in choose_prevailing_affect()
    #       is the desired functionality
    def get_prevailing_affect(self, affect_vector, allowable_error =
        ↪ 0.00000001):
        possible_affects = self.get_possible_affects(affect_vector,
            ↪ allowable_error)
        prevailing_affect = self.choose_prevailing_affect(
            ↪ possible_affects)
        return prevailing_affect

# a wrapper around a python dictionary to organize affects and setup
    ↪ rules for handling the 'resting' action
class Affect_Vector:
    # affect_names takes a list of strings
    def __init__(self, affect_names, equilibrium_values):
        self.affects = {}
        for affect in affect_names:
            # dictionary with strings specifying affects (as defined in
                ↪ Affecter) as keys and floats as values
            self.affects[affect] = equilibrium_values[affect]['
                ↪ equilibrium_point']
```

# Appendix C

# Animation State Machine Module

```
#
# Animation_Structure contains nested dictionaries for storing frames of
    ↪ animation and where in the animation loop the simulation is
# the default arguments of Animation_Structure use theatrical terms for
    ↪ organization
#
# because of the way modifier_list and action_list are used in the
    ↪ construction of the nested dictionaries
# the order of the lists passed only determines the order of index
    ↪ arguments needed to access the values stored
# in either self.animation_frame_lists or self.current_frames
#
# NOTE: THE CURRENT DEFAULT OF final_frame_index IS A PLACEHOLDER MAKE
    ↪ SURE TO CHANGE IN PRODUCTION
# final_frame_index is strictly a bookkeeping variable and necessary for
    ↪ update_displayed_frame() to work properly
#
# frame_rate_delay is the number of frames to hold on before a new frame
    ↪ of animation is selected
# the default values assume a 60Hz rate of calling update_displayed_frame
    ↪ () and a desired animation rate of 10FPS
#
class Animation_Structure:
    def __init__(self, frame_rate_delay = 5, modifier_list = ['tempo_up',
        ↪  'tempo_down', 'neutral'], action_list = ['open_flow', '
        ↪ closed_flow', 'projected_energy', 'resting'], affect_list = ['
        ↪ joy', 'hatred', 'sadness', 'wonder', 'love', 'desire']):

        # self.animation_frame_lists is a dictionary that uses modifiers
            ↪ to access actions to access individual frame lists
```

```
self.animation_frame_lists = {}
for modifier in modifier_list:
    self.animation_frame_lists[modifier] = {}
for modifier in self.animation_frame_lists:
    for action in action_list:
        self.animation_frame_lists[modifier][action] = {}
for modifier in self.animation_frame_lists:
    for action in action_list:
        for affect in affect_list:
            self.animation_frame_lists[modifier][action][affect]
                ↪ = []

# used to track the frame index of all animation lists
self.current_frames = {}
for modifier in modifier_list:
    self.current_frames[modifier] = {}
for modifier in self.current_frames:
    for action in action_list:
        self.current_frames[modifier][action] = 0

# used to store the end points of each part of individual
    ↪ animation frame lists
# to update the values to allow the update_displayed_frame()
    ↪ function to properly work:
#
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['startup'] = 6
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['loop'] = 8
# <Animation_Structure instance>.frame_index_delineators['<
    ↪ modifier>']['<action>']['final frame'] = len(<
    ↪ Animation_Structure instance>.animation_frame_lists['<
    ↪ modifier>']['<action>']['<affect>']) - 1
#
# the final frame line in the example assumes the desire is to
    ↪ have the final frame delineator as the last frame in the
    ↪ corresponding <Animation_Structure instance>
    ↪ animation_frame_list
# Animation_Structure assumes that 'startup' value < 'loop' value
    ↪  < 'final frame' value
#
self.frame_index_delineators = {}
for modifier in modifier_list:
    self.frame_index_delineators[modifier] = {}
for modifier in self.frame_index_delineators:
    for action in action_list:
        self.frame_index_delineators[modifier][action] = {'
            ↪ startup': 0, 'loop': 1, 'final␣frame': 2} #
            ↪ default values to be swapped out after the
            ↪ Animation_Structure is built (use
            ↪ load_animation_list())

self.current_displayed_frame = None
```

```
        self.current_animation_action = action_list[-1]
        # used for switching over to a new animation corresponding to the
            ↪  current action the player is taking once the previous
            ↪ animation has finished
        self.reached_end_of_frame_list = False
        # used for setting the frame rate of animations stored in this
            ↪ structure
        # default values assume an update rate of 60Hz creating an
            ↪ animation frame rate of 10FPS
        self.frame_rate_delay = frame_rate_delay
        # internal counter for advancing the frame rate delay
        self.frame_rate_delay_count = 0

    # used for putting full animations into the animation structure
    # frame_list is the set of animation frames to be added to the
        ↪ initialized animation_frame_lists
    # modifier, action, affect all correspond to where in the
        ↪ animation_frame_lists the animation frames will be stored (and
        ↪  by default should be strings)
    # startup_end_frame_index, loop_end_frame_index, final_frame_index
        ↪ are integer values used to mark off the different sections of
        ↪ the animation
    def load_animation_list(self, frame_list, modifier, action, affect,
        ↪ startup_end_frame_index, loop_end_frame_index,
        ↪ final_frame_index):
        for animation_frame in frame_list:
            self.animation_frame_lists[modifier][action][affect].append(
                ↪ animation_frame)
        self.frame_index_delineators[modifier][action]['startup'] =
            ↪ startup_end_frame_index
        self.frame_index_delineators[modifier][action]['loop'] =
            ↪ loop_end_frame_index
        self.frame_index_delineators[modifier][action]['final␣frame'] =
            ↪ final_frame_index
        return

    # should be used to access the currently displayed frame
    # should be returning a Sprite if used with RenPy
    def get_displayed_frame(self):
        return self.current_displayed_frame

    # returns the next frame in an animation sequence as well as the
        ↪ previous frame
    # this is primarily a book keeping function meant to allow a
        ↪ persistent
    # should be returning a Sprite if used with RenPy
    def update_displayed_frame(self, modifier, action, affect):
        if self.frame_rate_delay_count < self.frame_rate_delay:
            self.frame_rate_delay_count += 1
        else:
            #self.displayed_frame_data = (modifier, self.
                ↪ current_animation_action, affect, self.current_frames[
```

```
    ↪ modifier][self.current_animation_action])
for mod in self.animation_frame_lists:
    # skip to returning to resting if the player has changed
        ↪ the energy state
    if self.current_animation_action != action:
        if self.current_frames[mod][self.
            ↪ current_animation_action] < self.
            ↪ frame_index_delineators[mod][self.
            ↪ current_animation_action]['loop']:
            self.current_frames[mod][self.
                ↪ current_animation_action] = self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop'] + 1
        else:
            self.current_frames[mod][self.
                ↪ current_animation_action] += 1
            if self.current_frames[mod][self.
                ↪ current_animation_action] > self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['final␣frame']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] = 0
                self.reached_end_of_frame_list = True
    # startup animation
    elif self.current_frames[mod][self.
        ↪ current_animation_action] < self.
        ↪ frame_index_delineators[mod][self.
        ↪ current_animation_action]['startup']:
        self.current_frames[mod][self.
            ↪ current_animation_action] += 1
    # loop animation
    elif self.current_frames[mod][self.
        ↪ current_animation_action] <= self.
        ↪ frame_index_delineators[mod][self.
        ↪ current_animation_action]['loop']:
        if self.current_animation_action != action:
            self.current_frames[mod][self.
                ↪ current_animation_action] = self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop'] + 1
        else:
            self.current_frames[mod][self.
                ↪ current_animation_action] += 1
            if self.current_frames[mod][self.
                ↪ current_animation_action] > self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['loop']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] = self.
                    ↪ frame_index_delineators[mod][self.
                    ↪ current_animation_action]['startup'] +
                    ↪  1  # return to first frame of the
                    ↪ loop
```

```
            # return to default animation
            elif self.current_frames[mod][self.
                ↪ current_animation_action] <= self.
                ↪ frame_index_delineators[mod][self.
                ↪ current_animation_action]['final␣frame']:
                self.current_frames[mod][self.
                    ↪ current_animation_action] += 1
                if self.current_frames[mod][self.
                    ↪ current_animation_action] > self.
                    ↪ frame_index_delineators[mod][self.
                    ↪ current_animation_action]['final␣frame']:
                    self.current_frames[mod][self.
                        ↪ current_animation_action] = 0
                    self.reached_end_of_frame_list = True

        if self.reached_end_of_frame_list:
            self.current_animation_action = action
            self.reached_end_of_frame_list = False
        else:
            self.current_animation_action = self.
                ↪ current_animation_action
        self.frame_rate_delay_count = 0
    self.current_displayed_frame = self.animation_frame_lists[
        ↪ modifier][self.current_animation_action][affect][self.
        ↪ current_frames[modifier][self.current_animation_action]]
    return(self.animation_frame_lists[modifier][self.
        ↪ current_animation_action][affect][self.current_frames[
        ↪ modifier][self.current_animation_action]])
```

# Appendix D

# Example Rule File

```
{
    "joy" : {
            "actions" : {
                "open_flow" : 0.0007,
                "closed_flow" : -0.0005,
                "projected_energy" : -0.0002,
                "resting" : -0.0001
            },
            "modifiers" : {
                "tempo_up" : 1.25,
                "tempo_down" : 0.9,
                "neutral" : 1.0
            },
        "adjacent_affects" : ["desire"],
        "equilibrium_point" : 0.5
    },
    "hatred" : {
        "actions" : {
            "open_flow" : -0.0004,
            "closed_flow" : 0.00035,
            "projected_energy" : 0.0007,
            "resting" : 0.0001
        },
        "modifiers" : {
            "tempo_up" : 0.9,
            "tempo_down" : 1.3,
            "neutral" : 1.0
        },
        "adjacent_affects" : ["joy"],
        "equilibrium_point" : 0.5
```

```
},
"sadness" : {
    "actions" : {
        "open_flow" : -0.0004,
        "closed_flow" : 0.0009,
        "projected_energy" : -0.0002,
        "resting" : 0.0001
    },
    "modifiers" : {
        "tempo_up" : 0.95,
        "tempo_down" : 1.2,
        "neutral" : 1.0
    },
    "adjacent_affects" : ["hatred"],
    "equilibrium_point" : 0.5
},
"desire" : {
    "actions" : {
        "open_flow" : 0.0008,
        "closed_flow" : -0.0007,
        "projected_energy" : 0.0001,
        "resting" : 0.0001
    },
    "modifiers" : {
        "tempo_up" : 1.12,
        "tempo_down" : 0.7,
        "neutral" : 1.0
    },
    "adjacent_affects" : ["wonder"],
    "equilibrium_point" : 0.5
},
"wonder" : {
    "actions" : {
        "open_flow" : 0.0002,
        "closed_flow" : -0.0008,
        "projected_energy" : 0.0006,
        "resting" : 0.0001
    },
    "modifiers" : {
        "tempo_up" : 1.1,
        "tempo_down" : 1.05,
        "neutral" : 1.0
    },
    "adjacent_affects" : ["joy"],
    "equilibrium_point" : 0.5
},
"love" : {
    "actions" : {
        "open_flow" : 0.0001,
        "closed_flow" : -0.0009,
        "projected_energy" : 0.0007,
        "resting" : 0.0001
    },
```

```
        "modifiers" : {
            "tempo_up" : 0.6,
            "tempo_down" : 1.25,
            "neutral" : 1.0
        },
        "adjacent_affects" : ["desire"],
        "equilibrium_point" : 0.5
    }
}
```