A Programming Model for Boss Encounters in 2D Action Games

Kristin Siu[1], Eric Butler[2], and Alexander Zook[1]
[1]School of Interactive Computing, Georgia Institute of Technology
[2]Department of Computer Science & Engineering, University of Washington
{kasiu, a.zook}@gatech.edu; edbutler@cs.washington.edu

Abstract

Boss fights are a memorable and climatic point of many games. In this work we present a programming model for defining boss experiences in 2D action games. The domain we focus on is characterized by real-time movement through a continuous space in which the player and opposing boss damage each other through physical collisions, and player and boss behavior is governed primarily by finite state machines. Our programming model consists of primitive systems such as kinematic physics to store object state and detect collisions paired with finite state machines to define behavior. We describe our model and demonstrate its expressiveness with examples of three classic boss fights from games in *The Legend of Zelda*, *Castlevania*, and *Sonic the Hedgehog*. Our future goals include procedural generation of boss encounters, and we report on the research challenges involved in achieving this goal.

Introduction

Whether it is the first test of a player's skill or the climatic, final battle, boss fights are some of the most memorable moments in games. Compared to encounters with normal enemies, boss fights are intended to challenge the player, often acting as gatekeepers for the player's progression through the game. These encounters are used to ensure players have attained sufficient skill or completed the requirements necessary to progress further. When players overcome bosses, they are awarded in-game rewards, progress, and achievement, not to mention a sense of accomplishment.

We are interested in procedurally generating boss experiences similar to the boss encounters in classic 2D action or adventure games such as *The Legend of Zelda* series. These encounters are characterized by combat based around collisions with an enemy hitbox in a 2D plane using real-time motion in continuous space.

All content generation methods, at a broad level, reduce to searching a space of design possibilities (Shaker, Togelius, and Nelson 2015). For example, 2D level generation often is modeled as searching a space of possible assignments of tiles in a discretized 2D grid world, where the search is guided by features such as how the player moves through the space. While the methods for generation and objective functions can all be complex, the representation of a 2D grid-based level is clear.

However, with bosses, a representation suitable for generation is non-obvious. Traditionally bosses are implemented in general-purpose programming languages as they involve complex, state-based, and usually reactive behaviors. To generate these boss experiences we need a representation for bosses that is sufficiently expressive to capture the parts of the design space we care about while being limited and structured enough to make generation tractable.

In this paper, we present a programming model for representing boss experiences similar to classic 2D action games. *Boss experiences* are scenarios where a player interacts with a boss object that has patterned behavior; scenarios end based on termination criteria. Boss objects are physics objects enabling collisions—spanning roaming entities in the world (Moldorm in *The Legend of Zelda*) to massive set pieces (Sigma in *MegaMan X*). Behaviors govern the patterned movements and actions of bosses, and termination criteria govern when the experience ends (potentially in success or failure for the player).

Our programming model uses finite state machines to describe behavior paired with component-based systems to represent game state and perform calculations such as physics and hit points. This model is an important step towards generating boss experiences in this design space. The scope of this programming model is currently limited to specifying physical locations and controlling behavior. Aspects such as graphics and sound design, while important to boss design, are beyond the scope of this work.

To demonstrate the expressiveness of the model, we built a prototype that allows for the implementation of playable boss experiences. We describe three classic boss fights we replicated in it: Moldorm from *The Legend of Zelda*, Dracula from *Castlevania*, and Dr. Eggman from *Sonic the Hedgehog*. In the future we plan to explore techniques to automatically generate bosses using our model.

Related Work

Our programming model defines both the physical structure and behavior of a boss. Several researchers have generated in-game entity structures including ship hulls (Liapis, Yannakakis, and Togelius 2011), flower shapes (Risi et al.

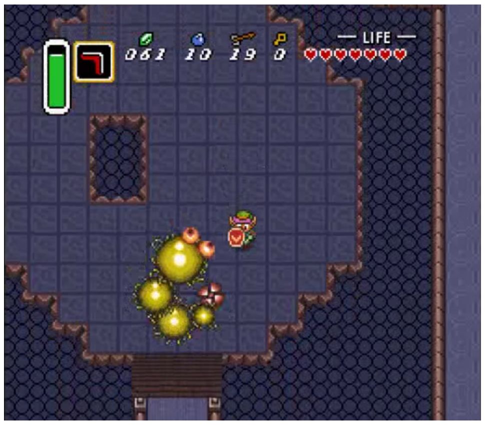


Figure 1: The Moldorm boss from *A Link to the Past*

2012), and ship weapon patterns (Hastings, Guha, and Stanley 2009). By contrast, our programming model targets a broad class of 2D entities and allows for connected parts with independently governed behaviors (though we do not yet generate entities).

Several languages have been developed to facilitate authoring behaviors for different domains. Examples include *Façade*'s A Behavior Language (Mateas and Stern 2002) for agent responses to players, the Versu storytelling platform's[1] (Evans and Short 2014) social model of small-group interactions, and *Prom Week*'s "social physics" model (McCoy et al. 2014) for characters' personal and social traits and events. By contrast, our model primarily models movement-based behaviors with simplified patterns that are amenable to generation rather than purely human scripting. Osborn et al. (2015) formally model combat in games in terms of operational logics (Mateas and Wardrip-Fruin 2009), but not at the level of a programming model. Instead, we target the specific entities in combat, rather than the scenario and systems encompassing combat as a whole.

Game description languages encompass the broader class of representations used to specify the entities, behaviors, and termination criteria associated with classes of games. Examples include turn-based competitive games in The Stanford Game Description Language (Love et al. 2008), puzzle games in Puzzlescript[2], and adversarial board games in the EGGG (Orwant 2000) and Ludi (Browne and Maire 2010). By contrast, our programming model targets a space of real-time, continuous behaviors. The Video Game Description Language (VGDL) (Schaul 2013) was developed to model arcade-style 2D games like *Frogger*, *Space Invaders*, and *Pacman*. Unlike the VGDL our programming model supports complex patterns of entity behavior and facilitates the creation of component-based composite entities. Casanova (Maggiore et al. 2012) is a DSL for game programming that allows for straightforward expression of update/draw loops and declarative rules, and supports efficient collision detection. Our programming is more high-level and more restricted, focusing on finite state machines as the primary update mechanism.

[1]`http://versu.com/`

[2]`http://www.puzzlescript.net/`

Domain

From ambient interactions to odious overlords, the space of boss experiences varies widely across games and genres. Here we focus on boss experiences in classic 2D action and adventure games. Classic 2D boss designs often use a 2D *top-down* camera perspective (e.g., *The Legend of Zelda* series and *Gradius*) or a *side-scrolling* perspective (e.g., *Castlevania*, *MegaMan*, and *Sonic the Hedgehog*). The combat in these games is characterized by real-time movement in continuous space, with attacks primarily based on collisions with a static or moving foe.

Boss experiences in these games share many structures. A player avatar operates in a 2D environment with an opposing entity (or entities)—the boss. Both the player avatar and the boss entity are given a range of 2D movement and a set of combat mechanics that each can use against each other. These combat mechanics are frequently (but not always) implemented through the use of *hit boxes* and *attack boxes*: the boss uses its body or projectiles to try to hit the player and cause damage, while the player tries to use its attacks to hit certain weak points in the boss and cause damage.

Classic 2D bosses act based on state-based scripted behaviors to follow patterns and change in reaction to the player's performance. For example, a boss may begin using new attacks after a player has successfully lowered its health parameter to a particular threshold. These behaviors follow patterns akin to finite state machines, with the boss transitioning through phases or attack patterns during the fight. Finite state machines are a typical representation used in implementations of bosses in this domain (Millington and Funge 2009). The experience ends when the player overcomes (or succumbs to) the boss in some way, such as reducing a health parameter to zero or surviving for a set amount of time. We capture common facets of 2D boss design in our modeling language for describing bosses.

Model Description

We now describe our programming model for bosses. Our model combines finite state machines to govern behavior with component-based systems such as physics and health. These systems expose primitive properties of the player, boss, and any other objects in the experience. A *kinematics physics system* exposes information about physical properties and geometry. A *health system* exposes information about boss vulnerabilities, and provides update routines such as collision detection.

Boss behavior is defined by a set of state machines, which we refer to as *behavior graphs*, whose actions modify the state of the primitives (e.g., controlling the velocity of an object) with transitions conditioned on the state of the primitives (e.g., reacting to a collision with the player). These behavior graphs can themselves have update functions and multiple behavior graphs can operate in parallel: for example, one graph may control how the main body of a boss should move, while another graph is responsible for tracking health and hit boxes. Both the player and bosses are fully described by the state of initial primitive objects (e.g., the physical shapes of the boss) and the behavior graphs that, together with built-in systems such as physical simulation, govern how the player and boss update over time and react to player actions.

As illustration we will construct the reoccurring Moldorm boss from *The Legend of Zelda* series using our model (Figure 1). We first describe the *physics* primitives that give the boss its form. We next describe the *health* primitives that allow the boss to interact with the player. Finally, we describe the *behavior graphs* that allow these primitives to change and thus define how the boss experience progresses.

Primitive State and Systems

The physics and health systems capture common elements shared by a large number of bosses. For example, all of the bosses we are interested in use kinematic 2D physics where collision detection plays a large role in boss behavior. The physics systems consist of two primary parts: a set of plain data types storing game state, and update functions that implement the mechanics of the system. This is similar to the component-entity model of representing game objects (Martin 2007), where state (e.g., physical location) is stored in components that are only data (no behavior). Behavior for all related components is implemented in the update function of a single system. For our prototype, we implemented two primary systems: 2D kinematic physics and a health system that tracks hit boxes and player/boss damage. However, this model is not strongly tied to these particular systems and can be instantiated with variations of these systems or even entirely new systems.

Physics. The player and bosses can have (potentially multiple) physical components comprised of a position and orientation in 2D space with linear and angular velocities and 2D shape. Our prototype supports circles and boxes for shapes and uses a simple kinematic model. The update function updates positions of all objects based on current velocities and detects collisions. Collisions are exposed and can be used as conditions for behaviors.

Now, consider Moldorm. Moldorm is a segmented worm typically composed of five pieces linked in a chain: a head, three body segments, and a tail. In our model, Moldorm is constructed from 5 physical objects, each with a circular shape of sequentially decreasing size. Figure 2a shows the initial state of Moldorm's physical objects.

Note that the physical environment of a boss experience can be an important aspect of boss design. Moldorm's boss area is a octogonal platform with a hole in the center. Part of the challenge is that Moldorm can easily knock the player off the edge or into the hole. Our model encompasses environments using the same elements as the player and boss. As this geometry is typically static, the key difference is that behaviors are not attached to these objects.

Health. Many bosses have health mechanics, where the player's victory condition in a boss experience is to damage a weak point of the boss until their health is reduced to zero. The health system stores this data. Health components are comprised of health statistics (such as hitpoints) and hitboxes to handle damage. Every physics object can be associated with a hit box, that can either damage the opponent and/or serve as a weak point to be damaged by an opponent. When a *damaging* hit box collides with a *damageable* hit box of an opponent, the owner of the damageable hit box takes damage. The health system's update function checks for such collisions using the output of the physics system. For Moldorm the entire boss is a damaging hit box and the player (Link) is a hit box, so the player is damaged when colliding with Moldorm. Moldorm has one damageable hit box: the final segment at the end of its tail. The player's sword swing creates a temporary damaging hit box capable of damaging the boss. The system's default reaction to damage is reducing health points; this behavior can be overridden with an arbitrary action for more complex behavior.

Node Type	**Action Description**
SelectRandom	Selects and assigns a random value (e.g., Moldorm's velocity).
Set	Sets a value (e.g., velocity).
ChaseTarget	Sets a target for an object (e.g., Moldorm's head is the target for its body parts).
DoNothing	Maintains current state (e.g., used to maintain current physics update).
SetHitbox(es)	Turns a hitbox on or off.
TriggerEmitter(s)	Creates game object(s).
RemoveObject	Destroys an object (e.g., a projectile).

Edge Type	**Transition Predicate**
OnAfter	True after some number of update ticks or frames.
OnCollision	True if a physical object collides with another.
OnValueEquals	True if a value equals a target (e.g., Dracula picks between two attacks).
OnDamaged	True if a damageable hit box is hit.
OnRModN	True when a value equals the remainder of a modulo n (e.g., used to create cycling attack patterns).

Table 1: Some common node and edge types used for behavior graphs in our model. Input parameters to nodes/edges have been omitted for brevity.

Behavior through State Machines

The primary constructs in our programming model for representing behavior are finite state machines, referred to as *behavior graphs*. In each state (node) of the behavior (graph), a different action is performed while the state is active. Transitions (edges) between states are taken when a condition predicate for a transition evaluates to true. Some states last for multiple game ticks, executing an action every game tick until a transition occurs. Other states can be instantaneous,

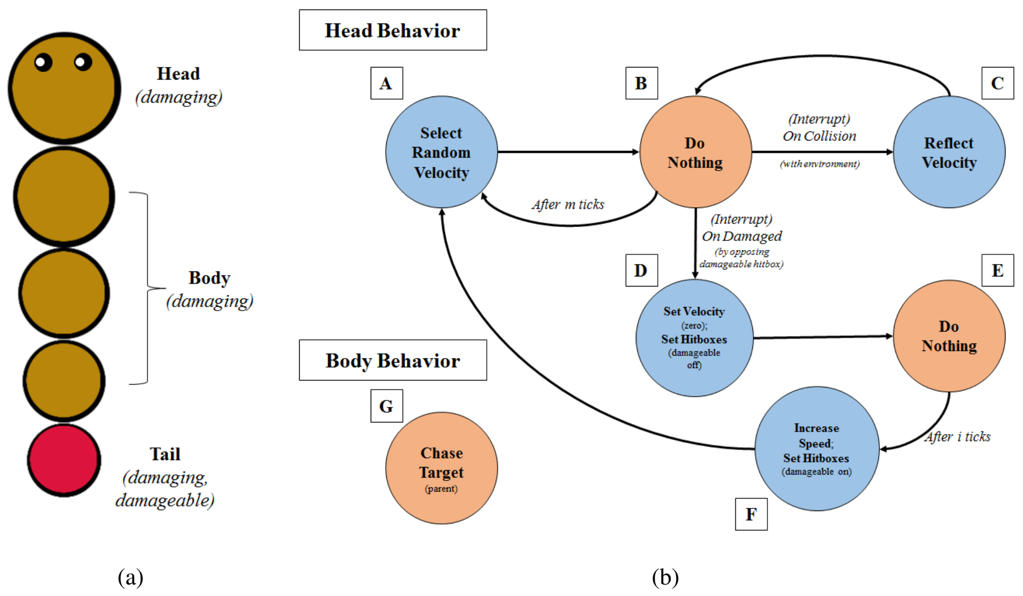


Figure 2: Definition of Moldorm. (*a*) shows the physical objects that make up Moldorm, and (*b*) shows the behavior graph. Blue nodes are instantaneous nodes while orange nodes update every frame until a transition occurs.

where the action is performed only once before immediately transitioning on the same game tick. Such states require at least one unconditional outgoing transition. Table 1 demonstrates some commonly used nodes and edges, most of which appear in our examples.

States correspond to different phases or attack patterns a boss may take. Moldorm moves through several states, illustrated in Figure 2b. Moldorm has two different high-level behaviors: first, it moves in random directions, changing its direction every few seconds or when hitting walls. Second, when damaged by the player (by being hit in the single weak point in the tail), it becomes invincible and stops moving for a few seconds. We represent this behavior in our model using a graph with 6 nodes.

Moldorm starts in the instantaneous state labeled **A** in the graph, where a random velocity vector at the current speed is chosen for Moldorm's head. Moldorm immediately transitions to state **B**, which performs no action during update. Note, however, that the physics system will update Moldorm's position according to its velocity. Two conditions cause Moldorm to move to different states: upon hitting a wall, Moldorm moves to node **C** which performs the instantaneous action of reflecting its velocity, then going back to **B**. Alternatively, after a delay of some specified time, Moldorm goes back to **A**, choosing a new random direction.

The second half of the graph is reached when the player damages Moldorm. Upon damage, assuming Moldorm's new health is above zero, Moldorm moves to instantaneous state **D**, which both stops its movement and disables its damageable hit box so it cannot be damaged. This temporary disabling of the hit box implements what is often called *invincibility frames*. Moldorm then does nothing in node **E** until a fixed time delay, where node **F** will re-enable the damageable hit box and increase Moldorm's speed. This immediately transitions back to **A** to choose a new direction for movement (at the new higher speed). If Moldorm is damaged during node **B** and its health drops to zero, the behavior will instead transition to termination, ending the encounter.

Bosses may be governed by multiple behavior graphs, either nested or in parallel. For example, Moldorm's body needs to follow the head. Because this happens regardless of the state of the boss, it can be represented as a parallel behavior graph with a single node, **G**. This node's update function performs the action of having the body parts follow behind the head, moving each body part to the location the head was at several game ticks previously.

Dynamic Object Creation

Creating and destroying game objects such as emitted projectiles is a common element of many boss encounters. Dynamic object construction is a straightforward extension of constructing the initial game state. Creating "objects" in this model consists of constructing a collection of one of more primitive objects (e.g., physical objects) and informing the systems responsible for updating them (e.g., the physics system) that these objects exist. To make this dynamic, actions in behavior graphs can construct objects using the same constructs as initial creation. Object destruction occurs by informing the relevant systems to stop updating the objects. In this model, we use *emitter* to refer to a physical object used as the source of dynamically created objects. Our Dracula example uses these to throw fireballs at the player.

Extending the Model

Physics and health objects can be viewed as primitive data types of the model, while behavior graphs are a general way to describe behavior conditioned on and modifying these primitives. The only interaction between the behavior graphs and these primitive objects is through reading/writing their mutable state; that is, the behavior graphs treat them as plain record data types (e.g., structs). Thus, the model can be straightforwardly extended with new data types (e.g., 3D physics). We chose primitive types that enabled us to express a particular class of 2D bosses.

Extensibility is an important property of the model since bosses are typically tailored to the mechanics of the game, particularly the actions the player can take. Therefore, it is important to be able to incorporate unique mechanics into the model. New systems can be added by providing one or more primitive struct types containing data and an update function that implements the mechanics.

This programming model can be viewed as being parameterized over the mechanics. That is, finite state machines are the common element of boss experiences in this domain, where the details of the systems such as physics or damage may vary from game to game. However, the model does not need to commit to any assumptions about these systems aside from them having some primitive components with state and an update function. Thus, the model can be instantiated for a variety of game mechanics. This work describes two extremely common such mechanics.

Examples

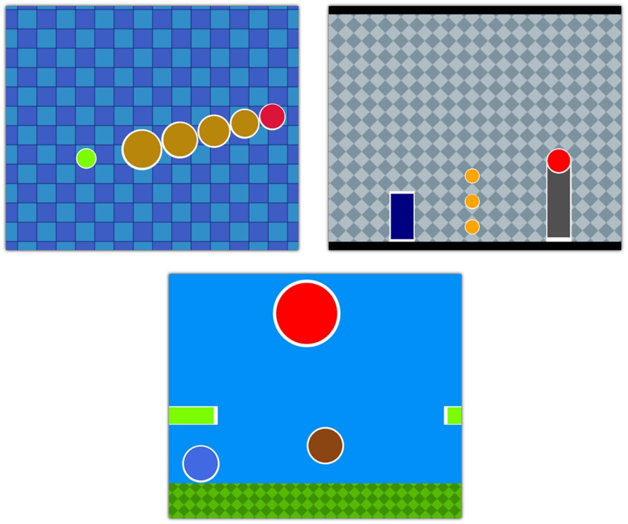


Figure 3: Screenshots of the three example bosses running in our prototype implementation of our model. Clockwise from the top left: Moldorm, Dracula, and Eggman.

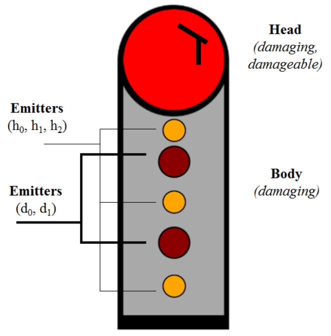
We implemented our model as a playable prototype and replicated several bosses from classic 2D games. We described Moldorm above; here we describe two additional bosses: Dracula from *Castlevania* and Eggman from *Sonic the Hedgehog*. While Moldorm uses a top-down camera perspective, these two bosses both use a side-scrolling camera. We omit descriptions of the player behavior for brevity. Players are defined using the same primitives and behavior graphs as bosses, be it Link, a Belmont, or Sonic.

Dracula from *Castlevania*. As an example of how our model can construct a 2D boss from a side-scrolling platforming game, we consider one version of the reoccurring *Castlevania* Dracula boss. In the *Castlevania* series, Dracula is traditionally the final enemy encountered by the player. The fight often takes place in multiple phases, and here we describe the first phase from the opening fight of *Castlevania: Symphony of the Night*.

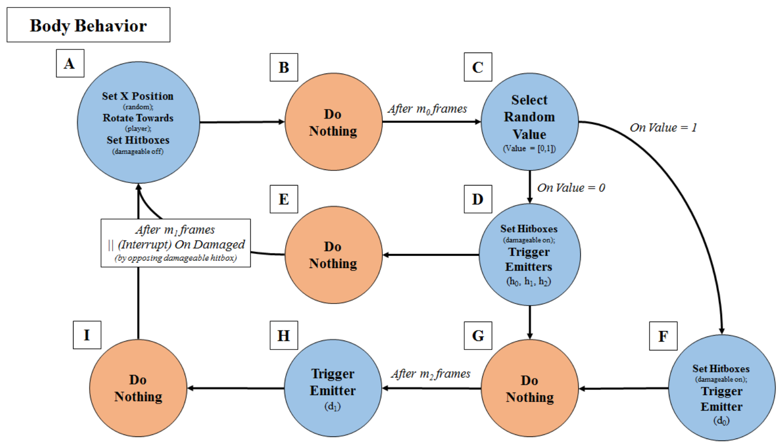
Figure 4 shows the physical components (a body and a head rigidly attached to that body) and behavior graph to define both movement and attacks. Dracula uses an additional parallel behavior graph to handle invincibility frames and death, which we omit for brevity. Dracula's body has five projectile emitters that fire two kinds of projectiles, his primary forms of attack. The first set of projectiles (referred to as "hellfire") are three small fireballs fired simultaneously horizontally at the player. The second set of projectiles (referred to as "dark inferno") are sequentially launched large, slow, circular fireballs. Dracula has a damageable hitbox on the head, only active when he is launching projectiles, and attack hitboxes on both the head and the main body.

In the boss experience, Dracula teleports to random positions while facing the player and randomly initiating one of the two attacks mentioned above. This repeats until the player reduces Dracula's health to zero (or succumbs to the attacks). The behavior graph begins in state **A**, which selects a random x position (along the floor) for the body and faces the player. The behavior graph then waits in state **B** while the teleport animation plays. The next instantaneous state **C** randomly selects a transition to choose an attack. Both attacks enable Dracula's damageable hit box. "Hellfire" creates 3 projectiles simultaneously with constant velocity (state **D**), one at each of three emitters, followed by a delay (state **E**), during which the boss is vulnerable, before returning to state **A** to repeat. "Dark inferno" triggers two emitters to launch in sequence; the first is followed by a short delay and the second is followed by a longer delay where the boss is vulnerable (states **F**, **G**, **H**, and **I**).

Eggman from *Sonic the Hedgehog*. Our third example is the boss from the first zone of the original *Sonic the Hedgehog* game. We omit illustration of the boss objects and its corresponding behavior graphs for brevity. Like *Castlevania*, this game uses a 2D side-scrolling perspective. The player (Sonic) can jump and collide with enemies to cause damage. The boss, Eggman, uses a flying capsule machine, augmented with a weapon. Typically only the capsule can be damaged, but at any time, with a short number of invincibility frames after each hit. The weapon in this first boss encounter consists of a swinging ball on a chain attached to the capsule. The ball always damages the player on contact, but the capsule only damages the player if they are not performing their jumping attack; otherwise the boss is damaged. Platforms in the arena help the player attack the boss capsule.



(a)


Body Behavior
A
Set X Position (random); Rotate Towards (player); Set Hitboxes (damageable off)
B
Do Nothing
After m_0 frames
C
Select Random Value (Value = [0,1])
On Value = 1
On Value = 0
D
Set Hitboxes (damageable on); Trigger Emitters (h_0, h_1, h_2)
E
Do Nothing
After m_1 frames || (Interrupt) On Damaged (by opposing damageable hitbox)
F
Set Hitboxes (damageable on); Trigger Emitter (d_0)
G
Do Nothing
After m_2 frames
H
Trigger Emitter (d_1)
I
Do Nothing

(b)

Figure 4: Definition of Dracula. (*a*) shows the physical objects that make up Dracula, and (*b*) shows the behavior graph.

The boss has two physical objects and its behavior is comparatively simple to the previous examples: the boss is not reactive except for when health points are reduced to zero. Eggman's capsule moves at a constant velocity from one side of the stage to the other while the ball swings in an arc below. Thus (excepting boss death) the only behavior state transitions are time-based, and the only actions are adjusting the velocities of the capsule and ball. The ball uses a parallel behavior graph to calculate its position analytically as a function of time and the capsule position.

Discussion

In this paper, we described a programming model for constructing boss experiences in 2D action and adventure games. We model boss encounters characterized by real-time movement in a continuous 2D space, where the outcome is determined primarily by collision with hit boxes, and behavior is controlled through finite state machines.

As our intent is enabling boss generation using this model, we have a number of limitations. We focused on boss battles based on physical combat, so our model does not readily support boss experiences focusing on systems like resource management. We also assume a simple mapping between player input and the player character's actions and cannot capture cases like a menu-based GUI in RPG battles, though we can model the mechanics of the battle itself.

Our model focuses exclusively on the rules and behavior of objects, while ignoring aspects such as graphical representation, sound design, and character design. Many boss experiences are not about the particular behavioral details of the player or enemy, instead emphasizing characterization of the opponent. One important research question for generation is how to model and incorporate semantic background knowledge into models of boss design. Our work focuses on one domain of many in a class of broader aspects of games that are not typically modeled, and we hope to see more work in other such domains.

Future Work

We have two primary areas of future work: boss generation and designing a game with procedural bosses. Boss generation can use a number of different paradigms. Our model defines a grammar for bosses, but admits variable-size boss representations. As such, generative methods that randomly expand grammars (Togelius, Shaker, and Dormans 2015) or perform heuristic search over a grammar are likely appropriate (Togelius and Shaker 2015). We imagine a search in the space of grammar expansions that evaluates bosses in terms of working implementation (as a constraint) and a continuous optimization function for boss encounter quality. A boss is defined in relation to the player in terms of what actions and skills the player is capable of. Thus any system for generating bosses will have to reason about the interactions between player and opponent, using simulated or real player information. Defining general (or at least reusable) metrics for measuring the quality of a boss experience remains an open research question.

Our model can also support offline, mixed-initiative design, where the same generation process can provide boss suggestions to a designer based on given player mechanics or high-level descriptors of intended player experience. This can in turn create bosses grounded in interesting characterization or with interpretable meaning for the player, as projects like Game-O-Matic (Treanor et al. 2012) have done for game designs.

Boss generation can enable games where facing an ever-changing, ever-adapting set of bosses is the core game mechanic. Compared to boss-centric games like *Shadow of the Colossus* or *Monster Hunter* our system has the potential to *generate* a series of bosses that responds to player performance and preferences. We envision a game where each new boss adjusts its physical body, behavior graph, or represented systems, possibly based on a continually-updating model of player performance.

References

Browne, C., and Maire, F. 2010. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games* 2:1–16.

Evans, R., and Short, E. 2014. Versu a simulationist storytelling system. *IEEE Trans. Computational Intelligence and AI in Games* 6(2):113–130.

Hastings, E.; Guha, R. K.; and Stanley, K. 2009. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games* 1:245–263.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2011. Neuroevolutionary constrained optimization for content creation. In *IEEE Conference on Computational Intelligence and Games*.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical report, Stanford University.

Maggiore, G.; Spanò, A.; Orsini, R.; Bugliesi, M.; Abbadi, M.; and Steffinlongo, E. 2012. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, 287–292. ACM.

Martin, A. 2007. Game balance concepts. `http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/`.

Mateas, M., and Stern, A. 2002. Architecture, authorial idioms and early observations of the interactive drama Façade. Technical report, Carnegie Mellon University.

Mateas, M., and Wardrip-Fruin, N. 2009. Defining operational logics. In *DiGRA*.

McCoy, J.; Treanor, M.; Samuel, B.; Reed, A.; Mateas, M.; and Wardrip-Fruin, N. 2014. Social story worlds with Comme il Faut. *IEEE Trans. Computational Intelligence and AI in Games* 6(2):97–112.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann.

Orwant, J. 2000. Eggg: Automated programming for game generation. *IBM Systems Journal* 39(3.4):782–794.

Osborn, J. C.; Lederle-Ensign, D.; Wardrip-Fruin, N.; and Mateas, M. 2015. Combat in games. In *10th International Conference on the Foundations of Digital Games*.

Risi, S.; Lehman, J.; D'Ambrosio, D. B.; Hall, R.; and Stanley, K. O. 2012. Combining search-based procedural content generation and social gaming in the petalz video game. In *8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence in Games*.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2015. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Togelius, J., and Shaker, N. 2015. The search-based approach. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Togelius, J.; Shaker, N.; and Dormans, J. 2015. Grammars and l-systems with applications to vegetation and levels. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, 11:1–11:8. New York, NY, USA: ACM.