Proceedings, The Thirteenth AAAI Conference on
Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)

HyPED: Modeling and Analyzing Action Games as Hybrid Systems

Joseph C. Osborn, Brian Lambrigger, Michael Mateas
Computational Media
University of California, Santa Cruz
1156 High St
Santa Cruz, CA 95064

Abstract

Platformers and action-adventure games have high-dimensional state spaces with difficult, non-linear constraints on character movement; even worse, game environments often respond to the player in complex ways that can cause exponential expansion of the planning search space. Planning problems in these high-dimensional spaces generally require domain-specific knowledge and manually abstracted models of game rules to replicate the intuition of human designers or playtesters. In this work, we outline a system for modeling these complex games at a precise and low level in terms of hybrid automata. With this representation, standard incremental search algorithms can be used to answer reachable-region queries, taking advantage of the domain information embedded in the system.

Introduction

Recent work in game design support has successfully argued that games' emergent qualities—the chaos that results when players interact with games' complex rules—leave a substantial role for automation in the game design process. The educational puzzle game *Refraction* used model checking to ensure that all solutions to a puzzle required the use of necessary mathematical concepts (Smith, Butler, and Popovic 2013). Some continuous-time games incorporate such solution-finding techniques into their game design itself: *CloudberryKingdom* generates new game levels on the fly but ensures that they can be won by a player with bounded reaction time (Fisher 2012).

Besides solution search, one general approach has become increasingly popular in recent years: visualizing (approximations of) reachable regions (Bauer and Popović 2012; Shaker, Shaker, and Togelius 2013; Tremblay, Borodovski, and Verbrugge 2014; Isaksen, Gopstein, and Nealen 2015). In this paper, we aim to improve the availability of these sorts of tools without committing to specific game engines or game-making tools.

We follow after the game description language *cum* game engine *cum* model checker BIPED (Smith, Nelson, and Mateas 2010), which leverages declarative game specifications to enable both human-playable prototypes and machine-analyzable models. BIPED is specialized for games with discrete state spaces and fairly coarse discrete time. What would a BIPED-like system for continuous-time games with hundreds of real-valued variables look like? We answer this question by combining folk approaches of action game makers—state machines, object pooling, collision detection, game physics simulation, and so on—with formal approaches from the hybrid systems literature and recent work in action game design support.

HyPED (Hybrid BIPED) is a modular formalism for defining action game characters, grounded in the theories of operational logics (Mateas and Wardrip-Fruin 2009) and hybrid automata (Alur et al. 1993) (*hybrid* here refers to the fact that these systems hybridize continuous and discrete behavior). Operational logics combine abstract processes like collision restitution or state transition systems with a strategy for communicating those processes to players—for example, characters which play different animations in different states. HyPED models game entities from graphical logic games, privileging collision logics and physics logics. These games center simulations of continuous space and time, collisions between objects, and objects with small sets of discrete variables whose behaviors change under different circumstances (generally indicated by changes in visual appearance). HyPED entities and environments can be readily converted to Unity or other game engines.

Action games (whose mechanics focus on movement and collision detection) constitute arguably the most prevalent genre of digital games. Accordingly, many popular game-making tools focus on them: *GameMaker*, *Unity 3D*, *Unreal*, and other systems offer user interfaces and programming APIs supporting collision checking, simple physics, and in some cases state machines. At the same time, none of these offers significant support for model checking, visualization of possible system states, parameter synthesis, or other features that the available domain knowledge would seem to enable. These tools could be of great use to game makers, reducing the need for expensive manual testing and design space exploration.

In the following sections, we define the action game modeling language HyPED and show examples of the diverse entities it can define. We also show how incremental search (specifically, Rapidly-exploring Random Trees) can be applied to the reachable-regions problem for HyPED games (a more expressive class of games than those to which these

techniques have been previously applied). Moreover, we show how deep knowledge of system dynamics can improve the performance of RRT for games, mainly through dimensionality reduction.

Related Work

Some game-making tools do support partial declarative definitions of game entity behaviors. GameMaker provides for entities with behaviors driven by events (such as collisions or timer elapse) that trigger handlers supporting various conditional responses (e.g. changing velocity or incrementing a variable). These entities also have varying animations at different times and collision areas which correspond to the animations. Unfortunately, game entities with atomic behaviors outside of that predefined set are inexpressible in the declarative style; the GML scripting language is provided as an imperative escape hatch for such cases.

Game designers often narrate entities' behaviors in terms of state machines, and this trend is captured in both the academic (Siu, Butler, and Zook 2016) (our work may be seen as a rigorous grounding of the same fundamental idea) and game design literature (Swink 2009). The 3D game-making tools Unreal and Unity both provide for explicit state machines specialized for character animation, but in both engines most atomic behaviors are implemented in imperative code, and none of the above tools has a formal semantics.

This state machine-like description addresses entities' physical dynamics, audiovisual representation, and discrete variables like health or ammunition. These are not formal transition systems: they may have undecidable transition relations (due to various combinations of dynamics and transition guards) or their discrete state spaces may be infeasibly large. Some of these variations can be captured by other types of automata, but games generally employ state machine-flavored discrete systems and not formal automata.

Hybrid Automata

Hybrid automata combine a discrete transition system (a finite state machine) with a set of continuous variables and a switched set of differential equations over these variables called *flows* (Alur et al. 1993). In each state, a different subset of the flows is applied to the continuous variables until the state is exited along a transition, which may be guarded on continuous or discrete variables; transitions may also instantaneously modify variables' values. The usual semantics for hybrid automata is that they alternate between periods of continuous flow (called *delay* or *continuous transitions*) and instantaneous *discrete transitions*.

Hybrid automata have seen extensive use in modeling cyber-physical systems where linear (or simpler) dynamics adequately describe the partially known or complex true dynamics of a system whose behavior is different at different times. They have many varieties and syntactic extensions, and often their dynamics are restricted in one way or another (Henzinger 2000). These restrictions are helpful because even under very simple physical laws, the question of state reachability becomes intractable or even undecidable (though semi-decision algorithms exist).

Key analysis questions here include *safety* or state reachability (safety is often phrased as never reaching an unsafe, e.g., illegal or stuck state); calculating a *reachable region* (what possible values can the continuous variables take?); *controller synthesis* (can we generate a control policy or AI player which will be safe or meet some optimality criterion?); and *parameter synthesis* (given an automaton with some unknown parameters—jumping speed, gravity, platform height—can we find values for those parameters satisfying some constraint?).

Hierarchical Hybrid Automata Game entities often have highly structured behaviors; Super Mario's grounded movement comprises walking, running, and standing still, while his aerial movement has distinct rising and falling behaviors composed in parallel with moving left and right in midair. Modeling these distinct flows requires dozens of states, most of which are slight variations on each other.

Game designers generally handle these overlapping concerns by introducing hierarchy and concurrency into their state machines. This may be realized using groups of Boolean variables that have allowed and disallowed combinations (exactly one of `on_ground`, `jumping_up`, and `falling` is true at a given time, independently of `crouching`). These approaches to modularity have also been explored in hybrid systems; the CHARON language (Alur et al. 2001) is a very generic approach with well-defined semantics.

Besides the complexity of individual entities, hundreds of game entities may be created and destroyed during play; this alone puts a significant stress on existing hybrid automata tools which generally scale poorly with the extremely high-dimensional state space. Hybrid systems researchers call adding or removing automata from the system at runtime "reconfigurability", and CHARON's derivative R-CHARON addresses this along with ways for automata to reference each other explicitly (Kratz et al. 2006). HyPED can be seen as a specialization of R-CHARON for games.

While hybrid automata seem to be a natural fit for modeling action games, previous attempts to apply them have failed to scale either in terms of expressiveness or performance past toy examples due to limitations of the modeling languages and tools used (Aaron, Ivančić, and Metaxas 2002). We believe HyPED resolves many of these issues.

Rapidly-exploring Random Trees

Rapidly-exploring Random Trees (RRT) is a commonly-used incremental search algorithm for planning in high dimensional spaces (LaValle 1998). The basic RRT algorithm is to repeatedly sample points from the *configuration space* $\mathcal{C}_S$ of possible combinations of values of all the variables in the world, finding each time the closest node in the tree to the sampled point and growing it towards the sampled point; if this can be done without violating any constraints, the resulting node is added to the tree and the process continues. In the case where we want to search for all possible states rather than a specific path, RRTs are useful because they quickly expand the tree to cover the state space. This is due to their implicit *Voronoi bias*: newly sampled states

are most likely to be in the largest Voronoi region induced by the tree built so far. Additionally, for problems where a solution is required and traditional searches are not feasible (e.g., in high-dimensional state spaces), RRTs are an attractive choice because they require little knowledge of the actual dynamics of a system.

RRTs have previously been applied to the reachability question for hybrid automata (Branicky et al. 2003); most previous work in this area uses a lexical distance metric and assumes a single automaton. In our case, we set a high penalty for mismatched discrete modes but do not use a lexical distance because not all mode transitions of the same transition system distance are equally easy to make (consider taking a door from the `locked` to `unlocked` state versus taking a character from `standing` to `crouching`: both are one-step transitions but the latter is much easier to realize). Note also that our constraint violations are conditions such as player death or going out of bounds, rather than assuming any collision is a constraint violation—we are not moving robots around a factory floor, but game characters through a level.

Our work also forgoes an accurate cost-to-go metric (used to locate the nearest node in the tree), finding other ways to maintain the Voronoi bias that makes RRT effective. This is achieved in part through extensions to the RRT algorithm which are built for that purpose, and in part through analysis of the hybrid system to find a tight approximation of the reachable configuration space, preventing infeasible states from being sampled.

Game design support

RRTs are commonly used for pathfinding in games (Algfoor, Sunar, and Kolivand 2015). These approaches generally work on an abstraction of the concrete dynamics, e.g., by assuming a top-down 2D world where entities can move freely in straight lines, ignoring dynamic objects in the environment. In general, however, dynamic objects may not simply be obstacles to avoid, but also bridges or platforms that *must* be collided with in order to reach a goal location. This is a marked departure from traditional application areas for RRT, and one which merits further study in the realm of simulation and state space exploration.

RRTs have also been used to aid in level design and solution verification as a substitute for time-consuming testing by designers or playtesters. They are useful in these cases because their bias towards exploring the state space completely accounts for the wide range of possible states a player may find themselves in.

Bauer *et al.* used RRTs to trace jump trajectory and landing position of game agents in *Treefrog Treasure* (2012), creating a visualization of reachable states and paths available to the player throughout the level. This implementation, however, required advance knowledge of the parabolic motion of the character and the static world geometry. In contrast, the combination of HyPED's modeling formalism and RRT presented here allows for a wide range of both level and entity dynamics.

Tremblay *et al.* used RRT in more dynamic environments (2014). This work experimented with various search

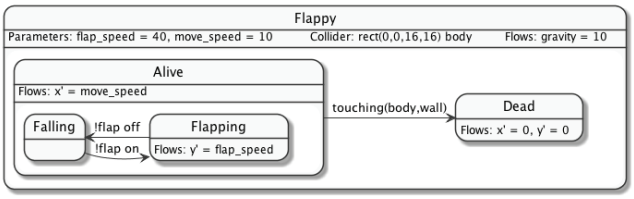

Figure 1: Flappy Bird.

algorithms, most notably the combination of RRT and local goal-directed planners like A* and Monte Carlo Tree Search. Some of the test environments had dynamic level geometry, such as moving platforms, to add a degree of difficulty above that of Bauer's work. Like the *Treefrog Treasure* work, planning was performed on an abstraction of the game character's true physics.

As Tremblay *et al.* noted, their RRT algorithm only samples in $\mathbb{R}^2$ (a designated player character's x and y position) and compares 2-D Euclidean distance, losing completeness (or at least causing extremely slow convergence) for levels with reactive elements. Many games have design elements whose state depends on player input, like collapsing bridges or moving platforms that stay inert until the player touches them. These reactive elements are troublesome for search algorithms as they exponentially increase the state space, preventing successful search if not carefully considered in state sampling.

Consider, for example, a sampled state on the other side of a gate which depends upon a switch to change the `locked` state of the gate. RRTs may try to expand nodes which, while close by euclidean distance, cannot progress without first backtracking to the switch; if the discrete state of the gate is not taken into account in sampling or the distance metric an RRT planner could easily become trapped. This expansion is addressed in our work by sampling in the full state space but also reducing the dimensionality of this space as much as possible via static analysis.

HyPED

The main design goal of HyPED is to translate concepts from hybrid control theory to the theory of action games so that tools and techniques from the former can be applied in the latter. There are substantial differences between classical hybrid automata and game character state machines, some of which have been detailed above. Here, we present a high-level account of HyPED's syntax and semantics for a game design audience. This explanation is self-contained, but more complete documentation can be found at our source code repository (Osborn and Lambrigger 2017).

Our immediate goal was to reduce repetition using hierarchical and parallel composition of behavioral modes. Fig. 1 illustrates hierarchical (but not parallel) modeling with a simple *Flappy Bird*-like entity. Flappy has two mutually exclusive top-level states: `alive` and `dead`. In the `alive` state, the entity moves to the right according to the value of its `move_speed` parameter (we call this type of continuous variable evolution a *flow*); this state is itself split into

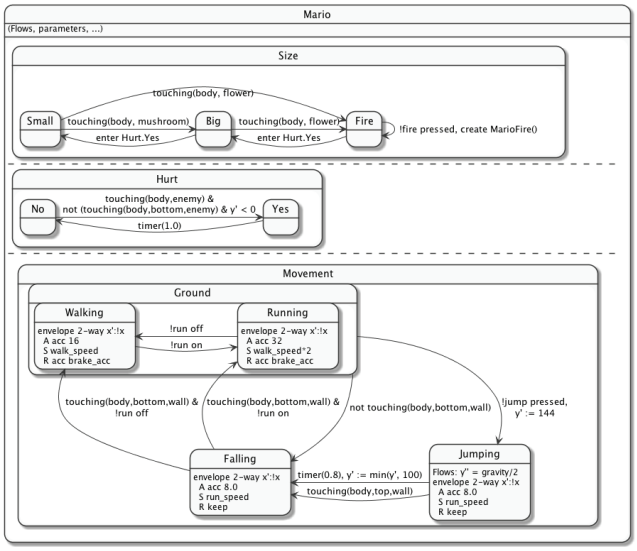

Figure 2: Super Mario (abbreviated).

`flapping` and `falling` states which alternate when the `flap` button is pressed or released respectively. Implicitly, Flappy's acceleration in y is `gravity`, but this is overridden by the `flapping` state which fixes y velocity.

Flappy has one square-shaped `collider` positioned at the entity's origin (entities may in general have several colliders, which can be conditionally active or inactive) tagged with the `body` type; when it touches something tagged `wall` the entity transitions into the `dead` top-level state no matter whether it's `falling` or `flapping`. Metadata in the HyPED game definition describe which tags are checked for collision with which other tags, and whether collisions should also reset the implicated velocity components to zero.

Flows defined in a state also apply in its descendants unless overridden (as in the `dead` state's $y' = 0$, which overrides the effects of gravity). Compared to classical hybrid automata, HyPED automata are extended mainly by admitting external theories like collision and button input.

For a more complex example, consider Mario (Fig. 2). Mario has several concurrent modes, including his `size`, whether he has temporary invulnerability after being `hurt`, and his `movement` (separated by dashed lines). The flows of a entity with multiple active parallel modes are the union of those flows; conflicting assignments to a single variable are illegal (in the case of Mario, only the movement mode applies any flows). To keep this rule straightforward, HyPED only allows concurrency at the top level of an entity.

The `Size` behavior group introduces two new concepts. First, whenever the entity `enters` the `Hurt.Yes` state, it becomes smaller; dying is the responsibility of a different behavior group (not shown). Second, the self-transition in the `fire` state produces a new `MarioFire` entity at Mario's current position. The `Hurt` group also introduces a new type of edge condition: the `timer`, which becomes true if the edge's origin state (in this case, `Hurt.Yes`) has been active for more than the given amount of time.

Finally, the `movement` group in Mario is much more complicated than it was for Flappy. The main behaviors here are `ground` movement and the two aerial modes which differ in gravity and air control. Note that these states do not for the most part define flows directly, but instead define *envelopes* in the sense used by Swink (Swink 2009). These attack/decay/sustain/release envelopes, inspired by audio processing and control theory, are like miniature automata on their own. They characterize (in this case) the left/right mirrored control over x velocity typical of platform game characters with acceleration (attack phase) up to a maximum speed (sustain phase) which quickly decreases to 0 when the buttons are let go (release phase). A character like *The Legend of Zelda's* Link would be moved by a four-way envelope activated by the `x` and `y` control axes and controlling Link's x and y velocity. Note that while Mario's ground movement has a deceleration when buttons are released, aerial horizontal movement `keeps` the current velocity instead.

The last new feature in this diagram is the edge update (e.g., $y' := \min(y', 100)$) which instantaneously changes a variable's value during a transition. This is used to set Mario's initial jumping speed (on the transition into `jumping`) and also to clip that speed when the jump button is released to get Mario's characteristic fine jumping control.

A HyPED game (or *world*) consists of a number of entity definitions as described above along with an *initial configuration* defining static level geometry (e.g., tilemaps, possibly imported from the Video Game Level Corpus (Summerville et al. 2016)) and the initial *instances*, if any, of each entity definition. HyPED further divides up the world into distinct *spaces* which are linked together in the style of *Zelda* rooms, each with its own population of entities.

HyPED is very expressive; it can describe not only entities like those above but also completely different ones including Link, moving platforms, doors, *Treefrog Treasure's* jumping frog, a top-down racecar, and even (in theory) three dimensional entities and spaces, though this is not yet supported by our interactive player program. To support these and other entities, HyPED also provides for references between entities (as in R-CHARON) and for discrete variables which may only be updated during transitions. Discrete variables provide for quantities like ammunition or keys.

Static Analysis

Because we have deep knowledge of game entities' discrete and continuous dynamics, we can derive useful information to simplify game entities or to improve performance. For example, it is straightforward to combine the dimensions of static level geometry with information about entities' colliders to determine what positions they might be able to inhabit. We can also leverage the structure of entities to generate fast cache-aware data representations or low-level code.

Consider a horizontally moving platform which smoothly animates left and right between two fixed positions using a `left` state and a `right` state. Some useful properties are immediately obtained: we know from the flows of each state that this platform's y position will be constant and that its x velocity will always be one of two fixed values. We also can learn (by looking at the transition guards) that the

platform's x position will always be within a fixed interval. In the present work, we use this information to reduce the dimensionality of configuration space, which helps RRT find better solutions faster without losing completeness. Keeping the sampled configuration space closer to the true reachable space of the system is a good way to maintain Voronoi bias.

In the future we hope to extend this to approximate entities' behaviors, e.g., by finding closed-form equations for cycles through an automaton; this could be used to turn Mario's jump into a (piecewise) parabola suitable for model-predictive control, reducing the number of discrete dimensions of the system. We are especially excited at the prospect of generalizing these techniques to help generate hierarchical plans, treating a game like *Zelda* as the combination of a low-level moving-and-attacking game with a high-level graph navigation game. Smoothly climbing up and down between different abstractions of a game entity is key to this approach, as is finding concrete propositions which, if they were to become true or false, would help achieve an intermediate goal. HyPED models contain the information necessary to perform these sorts of transformations.

Translation to Unity

Although we have an interactive player application, HyPED is not a complete game engine—that is explicitly *not* a goal of the system. HyPED is for prototyping and analyzing game worlds and entities. We expect that HyPED entities and levels will be exported or translated to other engines once their design is sufficiently well-understood; here we outline that approach for the Unity engine.

The static geometry currently used in HyPED is a simple tilemap for which a translation to Unity is obvious. We focus on translating HyPED entities to Unity prefabs. An entity combines several concurrent top-level behaviors with some colliders from a fixed set, of which only some might be active at a given time. These colliders correspond to e.g., Unity `BoxColliders` and have similar parameters. We envision one `MonoBehavior` per HyPED entity type to manage collider positioning relative to the entity and to activate or deactivate these colliders as necessary.

Posit one additional `MonoBehavior` subclass per concurrent top-level behavior group; it should carry a bitfield or a set of Boolean values describing which child states are currently active. In `FixedUpdate`, each active mode's continuous flows and envelopes are applied and transition guards are checked; if any guard is satisfied, that transition is taken (updating the active states) and instantaneous updates are applied, skipping the rest of the guards. The behavior class should also track which modes were entered and exited during this update cycle. The low-level implementation of these classes should closely mirror the Python version.

The Unity object obtained by assembling these behaviors and colliders together should exactly realize the HyPED definition. Game-specific aspects including graphical properties can be added on top of this base object.

RRT Improvements

A key argument for formal modeling languages is the ready availability of algorithms and software tools for querying or verifying properties of the model. Incremental search has proved successful in analyzing dynamical systems, so we wondered how well these techniques would work in this somewhat more complex hybrid domain.

We explored three main improvements to rapidly-exploring random trees for high-dimensional or highly-constrained domains. Resolution-Complete RRT (RC-RRT) back-propagates failure signals similarly to Monte Carlo Tree Search in order to prune unsuccessful branches from consideration (Cheng and LaValle 2002). In our tests, dynamic level geometry creates many situations which can trap progress. With RC-RRT, branches leading to bad states are less likely to be expanded, improving coverage.

Reachability-Guided RRT creates an approximate hull of reachable points around the leaves to help determine if, for any sampled state and nearest node, there is an input which can control the node to the state (Shkolnik, Walter, and Tedrake 2009). This is useful for complex constraints as it prevents the selection of nodes that, while close to a sampled state, may be blocked or trapped.

Environment-Guided RRT (EG-RRT) uses the back-propagating failure of RC-RRT in tandem with the reachability approximation of RG-RRT (Jaillet et al. 2011). This improves search performance in situations where both alternatives fail. Our test cases contain both complex hybrid dynamics and non-static geometry, so we would expect EG-RRT to perform the best here as well.

Refined Sampling

Because we know the system dynamics, we can constrain the configuration space without losing completeness. Naively, we might imagine that for each entity with 2D position, velocity, and acceleration along with a discrete mode we would require six continuous and one discrete dimension in configuration space. If we knew in advance that some entity was (e.g.) a platform that only moved with constant positive or negative velocity between two x coordinates with a fixed y position, we could easily reduce this down to one continuous dimension with narrow bounds (x) and one Boolean discrete dimension. We calculate such reduced spaces automatically by simple inspection of flows after constant propagation in each discrete space. Our reduced space $\mathcal{C}_{SR}$ is still an over-approximation (no actually-valid states will be outside $\mathcal{C}_{SR}$), so this simplification does not lose completeness.

Evaluation

For our testing, we crafted several high-dimensional planning problems with non-trivial solutions, both to test the expressiveness of HyPED's models, and to evaluate solution planning in the complex spaces afforded by hybrid automata representation. Each RRT-based planner (RRT, RC-RRT, RG-RRT, EG-RRT) was given a budget of 20 seconds to find solutions in the test levels, with the average time to find a solution calculated and recorded in the following tables. We also tried variants of these algorithms which only sampled in $\mathbb{R}^2$, but these were strictly dominated by sampling in the full configuration space.

Compared to Tremblay and Bauer's results, our computation times are higher. We believe this is mainly because we

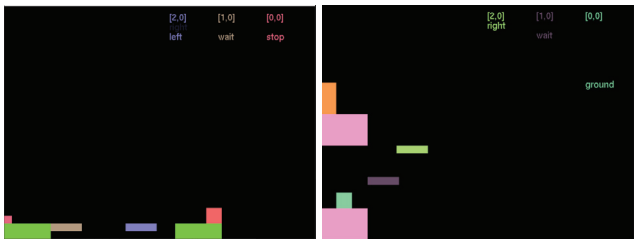

Figure 3: Tests 1 and 2.

work on real (and arbitrary) system dynamics of more complex characters and environments. For similar reasons, we were not able to successfully apply A* as a local planner.

Test 1

In Test 1 (see Fig. 3), we introduce reactive level elements with a simple timing puzzle. Starting on the far left, the agent must navigate across the platforms to the colored goal square on the right. The agent itself may only move right, left, or stay still. Platform 2 (in blue) on the middle-right continually moves between the middle of the gap and the other island, while Platform 1 (in brown) remains stationary until the agent touches it, after which time Platform 1 acts similarly to Platform 2. To navigate to the goal, the agent must time their boarding so that Platform 1 meets Platform 2 in the middle of the gap, then board Platform 2 and ride it to the goal. Humans can do this after a few tries, but heuristic search performs very poorly since it must do nothing for a while before making progress (it is a surprisingly challenging problem despite its visual simplicity).

In our tests, EG-RRT succeeds most consistently (see Tab. 1); the failure propagation metric prunes any branches which fall off the level or activate platform 1 at poorly-chosen times. EG-RRT's success is further increased by the refined sampling described above.

Naive Bounds				
	#Nodes	Goal Reached		#Success
		Time	#Nodes	
RRT	1087	4.87s	334	27
RC-RRT	1083	4.94s	308	22
RG-RRT	580	3.34s	109	20
EG-RRT	771	7.91s	315	32
Reduced Bounds				
RRT	1048	3.31s	219	24
RC-RRT	1018	3.11s	219	24
RG-RRT	505	2.44s	69	13
EG-RRT	814	9.8s	365	46

Table 1: Results for Test 1; timeout at 20 seconds (50 runs)

Test 2

In Test 2 (see Fig. 3), the difficulty is increased further, as the timing puzzle introduces both an obstacle and a means of progress. In this test, the agent is a Mario-like agent which can jump with complex dynamics. Once again, Platform 2 (in green) moves horizontally between the upper geometry and a position about 200 pixels to the right. Platform 1 behaves as before, remaining at rest until touching the player.

In this test (see Tab. 2), EG-RRT and RG-RRT perform poorly without reduced bounds due to the fact that if the sampled state is not closer to the reachable hull than to some node in the tree, they will choose a new sample state and begin the process again, evidenced by the much smaller number of nodes explored compared to RRT and RC-RRT. Only when the state space is reduced does EG-RRT once again outperform the rest; it improves even more than the other algorithms do. It is interesting to note that (at least for this problem) the less-informed search algorithms can obtain results much faster when they are able to obtain results at all, but their success rate is much lower.

Naive Bounds				
	#Nodes	Goal Reached		#Success
		Time	#Nodes	
RRT	224	1.61s	23	21
RC-RRT	221	2.53s	32	24
RG-RRT	99	0.62s	4	8
EG-RRT	113	1.19s	7	13
Reduced Bounds				
RRT	204	2.75s	45	40
RC-RRT	199	3.55s	45	32
RG-RRT	92	4.62s	25	28
EG-RRT	107	6.33s	38	43

Table 2: Results for Test 2; timeout at 20 seconds (50 runs)

Future Work

Bringing down constant factors and getting more RRT exploration steps for a given time budget is our immediate next step. We are currently using a naive implementation of RRT nearest-neighbor finding to make it easy to try out variations on the algorithm; improving this is obvious and should give a substantial performance boost (Atramentov and LaValle 2002); alternately, we could adapt HyPED to leverage a standardized planning library. Using a different distance function is also worth exploring. Right now we use squared distance, but this may not be a good choice for high-dimensional data. Later, we would like to further refine the configuration space by deeper static analysis and also to explore non-incremental-search approaches to the reachable-regions problem (for example, bounded model-checking).

We also hope to incorporate more RRT variants including RRT+ (Kim and Esposito 2005) and A*-guided RRT, and to characterize automatically how well our RRT search trees cover the configuration space. Wrapping these search algorithms in useful interfaces for design support is also important future work. Finally, we believe we could improve the coverage of our technique by using random forests instead of individual random trees to address differences in starting position or game state.

References

Aaron, E.; Ivančić, F.; and Metaxas, D. 2002. Hybrid system models of navigation strategies for games and animations. In *International Workshop on Hybrid Systems: Computation and Control*, 7–20. Springer.

Algfoor, Z. A.; Sunar, M. S.; and Kolivand, H. 2015. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology* 2015:7.

Alur, R.; Courcoubetis, C.; Henzinger, T. A.; and Ho, P.-H. 1993. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*. Springer. 209–229.

Alur, R.; Grosu, R.; Lee, I.; and Sokolsky, O. 2001. Compositional refinement for hierarchical hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, 33–48. Springer.

Atramentov, A., and LaValle, S. M. 2002. Efficient nearest neighbor searching for motion planning. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, 632–637. IEEE.

Bauer, A. W., and Popović, Z. 2012. Rrt-based game level analysis, visualization, and visual refinement. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Branicky, M. S.; Curtiss, M. M.; Levine, J. A.; and Morgan, S. B. 2003. Rrts for nonlinear, discrete, and hybrid planning and control. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 1, 657–663. IEEE.

Cheng, P., and LaValle, S. M. 2002. Resolution complete rapidly-exploring random trees. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 1, 267–272. IEEE.

Fisher, J. 2012. How to make insane, procedural platformer levels. *Gamasutra*.

Henzinger, T. A. 2000. The theory of hybrid automata. In *Verification of Digital and Hybrid Systems*. Springer. 265–292.

Isaksen, A.; Gopstein, D.; and Nealen, A. 2015. Exploring game space using survival analysis. *Foundations of Digital Games*.

Jaillet, L.; Hoffman, J.; Van den Berg, J.; Abbeel, P.; Porta, J. M.; and Goldberg, K. 2011. Eg-rrt: Environment-guided random trees for kinodynamic motion planning with uncertainty and obstacles. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, 2646–2652. IEEE.

Kim, J., and Esposito, J. M. 2005. Adaptive sample bias for rapidly-exploring random trees with applications to test generation. In *American Control Conference, 2005. Proceedings of the 2005*, 1166–1172. IEEE.

Kratz, F.; Sokolsky, O.; Pappas, G. J.; and Lee, I. 2006. R-charon, a modeling language for reconfigurable hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, 392–406. Springer.

LaValle, S. M. 1998. Rapidly-exploring random trees: A new tool for path planning.

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

Osborn, J. C., and Lambrigger, B. 2017. Hyped.

Shaker, M.; Shaker, N.; and Togelius, J. 2013. Ropossum: An authoring tool for designing, optimizing and solving cut the rope levels. In *Proceedings of the Ninth Aaai Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Shkolnik, A.; Walter, M.; and Tedrake, R. 2009. Reachability-guided sampling for planning under differential constraints. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, 2859–2865. IEEE.

Siu, K.; Butler, E.; and Zook, A. 2016. A programming model for boss encounters in 2d action games. In *Experimental AI in Games Workshop*.

Smith, A. M.; Butler, E.; and Popovic, Z. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *FDG*, 221–228.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of IEEE Conference on Computational Intelligence and Games*.

Summerville, A. J.; Snodgrass, S.; Mateas, M.; and Ontanón, S. 2016. The vglc: The video game level corpus. *arXiv preprint arXiv:1606.07487*.

Swink, S. 2009. Game feel. *A Game Designers Guide to Virtual Sensation. Burlington, MA* 1.

Tremblay, J.; Borodovski, A.; and Verbrugge, C. 2014. I can jump! exploring search algorithms for simulating platformer players. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.