Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2016)
Ladislav Kavan and Chris Wojtan (Editors)

Large-Scale Finite State Game Engines

Matt Stanton[1] Sascha Geddert[1] Adrian Blumer[1] Paul Hormis[1] Andy Nealen[2] Seth Cooper[3] Adrien Treuille[1]

[1]Carnegie Mellon University [2]New York University [3]Northeastern University

Figure 1: *Our precomputation method enables the automatic reuse of film assets and rendering pipelines in interactive games. This frame from our prototype game consists of roughly 900 million geometric primitives and includes complex lighting and dynamics, yet runs in realtime.*

Abstract

This paper presents a new model reduction technique that exploits large-scale, parallel precomputation to create interactive, real-time games with the visual fidelity of offline rendered films. We present an algorithm to automatically discretize a continuous game into a large finite-state machine that can be pre-rendered in the film world. Despite radical differences from existing game engines, our finite-state approach is capable of preserving important characteristics of continuous games including smooth animation, responsiveness to input, triggered effects and passive animation. We demonstrate our technique with a 30-second interactive game set in an award-winning short film.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation I.6.8 [Simulation and Modeling]: Types of Simulation—Gaming

1. Introduction

The past two decades have seen graphics hardware driving spectacular advances in game graphics and dynamics. Today however, we are witnessing the emergence of a new *cloud computing regime* characterized by powerful, on-demand parallel computation, large-scale content distribution networks, and thin clients. This new architecture will reshape the way computer graphics approaches space-time trade-offs, dynamics, and rendering. It also presents a grand challenge—using the cloud to infuse mobile games with film-quality graphics.

This paper addresses this challenge through *state graphs*, large-scale data structures that approximate continuous dynamical systems as large finite-state machines. While this model reduction technique has been successfully applied to isolated phenomena such as clothing [KKN*13] and fluids [SHK*14], we seek now to generalize this approach to the more rigorous requirements of game engines. In particular, this paper focuses on creating responsive games that integrate many dynamical and rendering systems, as one finds in commercial games.

Our method begins with an offline rendered film and a computer game, both created using standard production tools. We precom-

DOI: 10.2312/sca.20161239

pute a large number of possible paths through the game, effectively creating a large-scale *finite-state* description of every outcome in the game. Finally, we pre-render frames for all these states, re-using geometric assets and the offline rendering pipeline from the underlying film.

Our finite-state approach is radically different from existing game engines, which typically compute each frame on the fly. However, we still wish to preserve the *feeling* of traditional games, which we summarize into five essential properties that the game should exhibit: (1) smooth, jitter-free animation, (2) fast reaction to user input, (3) *event triggers* in which the world reacts discontinuously (and sometimes dramatically) to player action, (4) rich visual and audio effects that react physically to the main character, and (5) background animations that proceed independently of the character's action, which make the world feel as though it has a life of its own.

We present techniques to satisfy all five properties while simultaneously controlling the total number of states precomputed—the key limiting constraint of this state graph approach. To demonstrate our technique, we created a 30-second interactive game that exercises all five properties, set in the "runner" genre. We then rendered the game in the world of an award-winning short film, *The Rise and Fall of Globosome*, by Sascha Geddert. We directly re-used geometry and animation from the film to render a world with 900 million geometric primitives, consuming about 1 million hours of precomputation time. A frame from the game is shown in Fig. 1.

Contributions. This paper presents the first finite-state game engine that can tabulate games written in an existing game engine, while satisfying rigorous responsiveness and smoothness constraints, correctly modeling discrete triggered events, and incorporating film-quality offline graphics with complex secondary effects and background animations. Our demonstration serves as an existence proof that finite-state games can be used to bring interactivity to the worlds of animated films by faithfully approximating complex continuous games within reasonable state budgets.

2. Related Work

Game engines—codebases and frameworks for developing video games—are widely used, thanks to affordable creation tools for 2D and 3D games such as the popular GameMaker: Studio [YG16] and Unity [Uni16]. Our work uses such engines and takes them in a radical new direction. We design our game in Unity, but rather than running the game on the fly, we *transform* the game into a large-scale finite state representation suitable for offline rendering and cloud streaming.

Many game engines focus on specific categories of games, such as the Cocos2D-x engine for two-dimensional games [Coc15], or CryEngine for first person 3D games [Cry09]. Engines can also be used to craft discrete offline-rendered games in which the player has limited control, akin to a "choose-your-own-adventure" novel, such as Dragon's Lair [Cin83] and Myst [Cya93]. We apply this pre-rendering approach, for the first time, to much more fluid and interactive games. In particular, we optimized our engine for the *runner* genre, which is quite large and encompasses many successful games such as Temple Run [IS11], Canabalt [SSS09], and Flappy Bird [Ngu13]. Unlike previous approaches, our method requires no explicit computation of discrete states. Instead, our algorithm automatically discretizes a continuous state space, and allows for vastly more states with much denser connectivity. In other words, everything is precomputed, but still allows for real-time, interactive response to player actions.

Realtime graphics typically use specialized GPU algorithms to model soft shadows [HLHS03], subsurface scattering [HV04], and water [IDYN06, MM13]. A complementary approach is to exploit example-based precomputed methods. Linear subspace models have leveraged precomputation for lighting [SKS02], deformable solids [BJ05, AKJ08], and fluid dynamics [TLP06]; however, they place severe constraints on the form of the underlying dynamics and it is difficult to characterize their error. An alternative form of precomputation that avoids these pitfalls is large-scale tabulation of state graphs to model isolated phenomena such as cloth and lighting [JF03], clothing [KKN*13] and fluids [SHK*14]. Our work scales this approach up from modeling isolated phenomena to capturing an entire game engine, a much more complex task that requires carefully modeling important characteristics of real games (Sec. 3). We demonstrate how to *create* a finite state game by combining some of the rich authoring tools available to game and film creators today, including the Unity game engine [Uni16], 3DS Max [Aut14], V-Ray [CG15], and Nuke [TF14]. As a result, our technique has allowed us, for the first time, to set a highly responsive interactive game in an offline rendered film, reusing geometric assets and rendering pipelines from that world.

3. Overview

Our goal is to create an interactive game set in an existing animated film world. In essence, our approach is to precompute a large number of possible state trajectories through the game, pre-render these frames exactly reusing assets and rendering pipelines from the underlying movie, and blending them at runtime in response to user input.

The following sections describe how we used our state graph method to simultaneously satisfy all five properties we described in Section 1. These properties are: (1) jitter-free animation, (2) responsiveness to user input, (3) discontinuous event triggers, (4) rich secondary effects that respond to the character's action, and (5) independent background animation. In Sec. 4, we demonstrate how to achieve the first two of these properties in the tabulation phase. In Sec. 5, we then show how to achieve the final three properties by designing the state appropriately. Finally, in Sec. 6, we describe our experience applying this approach to a real-world demonstration game that exhibits these five properties.

Our 30 second demonstration game, set in *The Rise and Fall of Globosome*, an animated film by Sascha Geddert, follows a small spherical character, called a *globosome*, rolling down a canyon while controlled by the player. The globosome's motion is smooth and responsive (properties 1 and 2), but if the globosome rolls through certain areas, discontinuous events are triggered, such as the growth of a magical bridge (property 3). The world exhibits rich secondary animation, such as grass bending and water rippling in response to the globosome's motion (property 4), while background plants bend and wave in the wind (property 5).

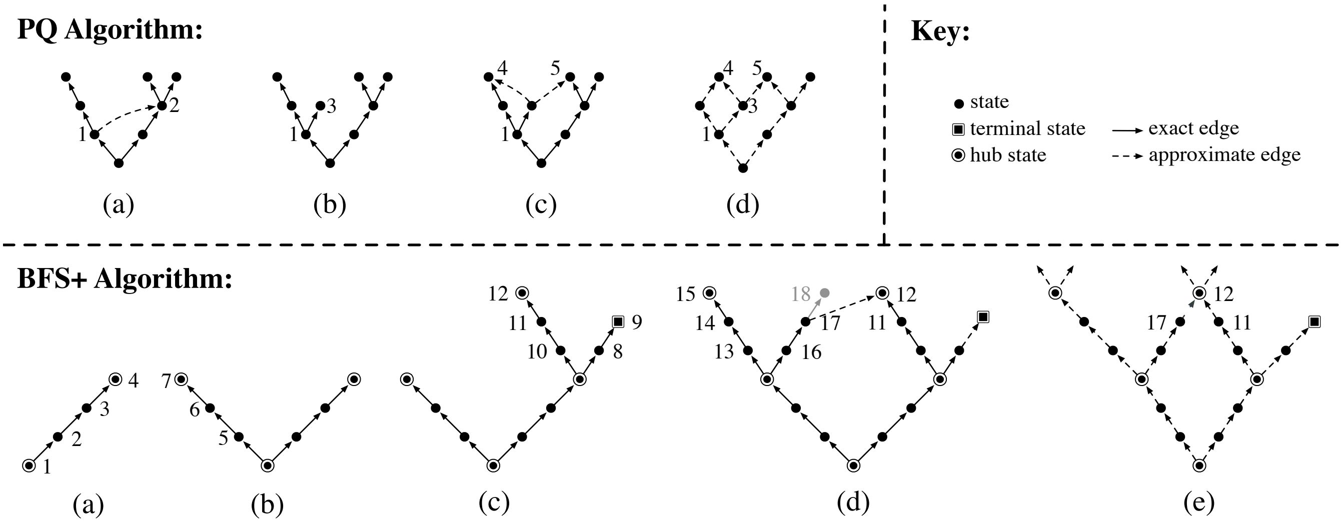

Figure 2: *We compare two algorithms for transforming a continuous computer game into a finite state machine. For simplicity, this figure depicts graphs with 2 controls ($|\Sigma| = 2$) and BFS+ state chain length 3 ($n = 3$). In our experiments, $|\Sigma| = 3$ and $n = 8$.* ***Above, PQ:*** *(a) The worst approximate edge is $1 \to 2$. If we cannot improve this transition by linking to a different state, instead (b) we simulate a new state 3 and add the exact edge $1 \to 3$. (c) We close the loop by finding optimal outgoing approximate edges from 3 back into $\hat{Q}$: $3 \to 4$ and $3 \to 5$. (d) Once the graph is completed, a smoothing step (Sec. 4.4) removes discontinuities.* ***Below,*** *BFS+: (a) We build a n-length state chain $1 \cdots 4$. and repeat again (b) for another control $1 \to 5 \cdots 7$. (c) We continue in a breadth-first manner, growing n-length chains from hub states. If a chain does not reach a terminal state (such as 9), we enqueue its final state as a new hub state (such as 12). (d) Alternatively, when a chain $7 \to 16 \cdots 18$ gets within ε of an existing hub state (such as 12), we redirect the chain to this state by introducing $17 \to 12$. (e) Finally, smoothing (Sec. 4.4) removes the discontinuities that redirection can create.*

4. Game Tabulation

Our goal is to tabulate all possible paths through a game *a priori* so that we can pre-render cinema-quality graphics. We now describe a process to automatically convert a computer game written in an existing game engine into a finite state representation. The goal of this process is to satisfy the first two constraints of Sec. 1 (smoothness and responsiveness) while at the same time controlling the total number of states.

We formalize this problem in the language of automata theory [Sip12]. A *continuous state game* G is similar to a finite state automaton, but can take on continuous states: $G = (Q, \Sigma, \delta, q_0, T)$. The continuous (infinite) *state space* Q describes all possible game configurations. The (finite) *control alphabet* Σ describes which button was pushed. For example, $\Sigma = \{left, right, neither\}$. The transition function

$$\delta : Q \times \Sigma \to Q$$

encodes interactive game logic at discrete timestep granularity. For example, pressing the *right* button in state q yields state $\delta(q, right)$ at the next time instant. We assume that the game's dynamics are provided by a transition function $\delta(q, \sigma)$, which can be queried for any state q and control σ, and that δ is continuous in q. The game starts in state $q_0 \in Q$ and terminates if we reach a terminal state $T \subseteq Q$, for example, by dying or completing the level.

A *finite state game* $\hat{G}$ approximates a continuous state game G as a large finite state machine—the entire game logic and rendering is reduced to a lookup table. Formally, a finite state game is a tuple $\hat{G} = (\hat{Q}, \Sigma, \hat{\delta}, q_0, T)$ where $\hat{Q} = \{q_0, q_1, \ldots, q_N\} \subseteq Q$ is a *finite* set of states which approximate every possible outcome in the game. Importantly, the game logic

$$\hat{\delta} : \hat{Q} \times \Sigma \to \hat{Q}$$

is now restricted to only take transitions within this finite set of states. Our goal is to tabulate the best possible finite state approximation $\hat{G}$ for G.

4.1. Two Tabulation Algorithms

We now compare two tabulation algorithms, *PQ* and *BFS+*, which attempt to preserve smoothness and responsiveness. *PQ* is derived from the priority queue based approach of Kim et al. [KKN*13]. While this algorithm created small state graphs with low error in the complex state spaces of cloth and liquid dynamics [SHK*14], when applied to our runner game it frequently produced unacceptable latency in response to user control changes. In response, we created *BFS+*, which uses a more predictable branching structure to achieve more predictable latency guarantees on our game dynamics.

Both algorithms represent $\hat{G}$ as a graph whose vertices are the finite set $\hat{Q}$ of states and whose labeled, directed edges, denoted $q \xrightarrow{\sigma} q'$, define the transition function: $q' = \hat{\delta}(q, \sigma)$. The algorithms measure the quality of each edge in $\hat{G}$ using an *error measure*

$$E : \hat{Q} \times \Sigma \to \mathbf{R}_{\geq 0}.$$

The error vanishes whenever the continuous game and finite state approximation agree exactly on the transition:

$$E(q,\sigma) = 0 \quad \text{iff} \quad \delta(q,\sigma) = \hat{\delta}(q,\sigma),$$

in which case we call this an *exact edge*. Otherwise, the transition has some error and is called an *approximate edge*. The exact form of the error is defined in Sec. 4.2.

Both algorithms start with a trivial graph which they iteratively "grow" into closer and closer approximations to the continuous game G. The first algorithm, *PQ*, operates by repeatedly improving the highest error transition (Fig. 2, top).

Algorithm 1 ***PQ*:** Begin with a trivial game $\hat{G}$ with only one state: $\hat{Q} = \{q_0\}$ and only self-loops $\hat{\delta}(q_0,\cdot) = q_0$. At each iteration, find the maximum error edge $q \xrightarrow{\sigma} q'$ in $\hat{G}$ and improve it to $q \xrightarrow{\sigma} q''$ as follows. First, search for a better destination state $q'' \in \hat{Q}$. If this search is successful, we can improve the edge by redirecting it, and we are done. Otherwise *create* a new exact edge by assigning $q'' := \delta(q,\sigma)$ and adding this new state q'' to $\hat{Q}$. If $q'' \notin T$ is non-terminal, also find optimal outgoing transitions from q'' back into $\hat{Q}$. In both cases, update $\hat{\delta}$ to reflect these new transitions. Repeat this process until the state budget is exhausted. Note that we can efficiently query the worst edge $\operatorname{argmax}_{q,\sigma} E(q,\sigma)$ by maintaining a priority queue, giving this algorithm its name.

This greedy, maximum-error approach is attractive because it was successfully employed by Kim et al. [KKN*13] to tabulate cloth and by Stanton et al. [SHK*14] to model fluids. However, we found that this approach yielded insufficiently responsive games, which at times registered noticeable delays in responding to player key presses. We therefore developed a second algorithm, *BFS+*, with a rigorous responsiveness guarantee: an upper bound n on the number of edges the player can cross before seeing an exact transition. We enforce this constraint by defining a set of *hub states* $H \subseteq \hat{Q}$ that are guaranteed to have only exact outgoing transitions ($E(q,\sigma) = 0$ for all $q \in H$) and that are separated from each other by no more than than n transitions (Fig. 2, bottom).

Algorithm 2 ***BFS+*:** Begin with a trivial game $\hat{G}$ with only one state ($\hat{Q} = \{q_0\}$) and place this state into a FIFO queue of hub states. At each iteration, pop a hub state q off the queue. Pick a control $\sigma \in \Sigma$ at random and follow it for n steps:

$$q \xrightarrow{\sigma} q_1 \xrightarrow{\sigma} \cdots \xrightarrow{\sigma} q_{n-1} \xrightarrow{\sigma} q_n,$$

adding these new states to the graph and halting only if we reach a terminal state. If any state q_i in this chain falls within ε of an existing hub state $q' \in H$, then terminate the chain at q_i and redirect it to q' by changing the destination of its final edge. (That is, assign $q_i := q'$ and use the transition $q_{i-1} \xrightarrow{\sigma} q'$ if this yields $E(q_{i-1},\sigma) < \varepsilon$.) Otherwise, add q_n as a new hub state by pushing it onto the queue. Link all the new states into a chain $q_i \xrightarrow{\cdot} q_{i+1}$ regardless of control, but allow early transitions *into* existing hub states if they improve the error. (More formally, for each control σ_i, let $q_i \xrightarrow{\sigma_i} \operatorname{argmin}_{q \in \{q_{i+1}\} \cup H} E(q_i,\sigma_i)$. q_{i+1} will usually have the lowest error—however, when σ_i is not the control σ picked to create the chain, a state in H may be better.) Now pick another control at random (to prevent bias) and grow another n-length chain, repeating this process until all controls are exhausted. Work on the state q is now complete. Pop another hub state from the queue and continue this breadth-first-search iteration until either the queue empties (in which case the game is fully tabulated), or the state budget is exhausted (in which case we throw an exception).

Note that *BFS+* allows merging only into hub states in H, not into ordinary states in $\hat{Q} \setminus H$. Also note that hub states are never more than n transitions apart, by construction, thus enforcing our bound n on the number of edges the player can cross before seeing an exact transition. Qualitatively, we observe that *BFS+* (with $n = 8$) produces finite state approximations which are much more responsive than *PQ*. We quantify this observation in Sec. 4.3. *BFS+* tabulations are not always smooth, but the jitter produced can be smoothed to acceptable levels in a post process (Sec. 4.4).

4.2. Error

We complete the algorithm descriptions by defining error. Both *PQ* and *BFS+* use the same error metric

$$E_{\hat{G}}(q,\sigma) = ||w \otimes (q - q')||^2 + \alpha\, \texttt{is\_backward}(q,q'), \qquad (1)$$

where $q' = \hat{\delta}(q,\sigma)$, w is a weight vector. The indicator function `is_backward` returns 1 if and only if q' lies behind q in the level's principle direction of motion (the negative z axis), and 0 otherwise. α is a large positive constant designed to penalize "backwards progress," and thus prevent cycles in the graph, which are highly undesirable for games, like ours, where the player should constantly be moving forward.

4.3. Algorithm Evaluation

We tabulated two different graphs with *PQ* using two different weight vectors: w_p ($PQ(P)$), with high weight on position, and w_v ($PQ(V)$), with high weight on velocity. We tabulated a single graph using *BFS+* using weight vector w_b, with intermediate position and velocity weights. Each algorithm variant was able to produce a game of the desired length in the desired number of states. However, $PQ(P)$ resulted in a game that did not meet the responsiveness criteria (Fig. 3b), and $PQ(V)$ only improved responsiveness slightly while resulting in far worse smoothness (Fig. 3c). Further weight adjustment for greater responsiveness resulted in games with unacceptable smoothness, demonstrating that a new algorithm would be required to fulfill both criteria simultaneously. *BFS+* produced a much more responsive gameplay experience with only a small additional smoothness cost, which we could mitigate using an offline smoothing step (Sec. 4.4) to create a satisfying gameplay experience.

We computed smoothness and responsiveness across a set of gameplay traces, or recorded paths through the game consisting of a sequence of state-control pairs (Fig. 4). We recorded these paths only once, in the continuous Unity game, and reconstructed them in each tabulated graph. Given a trace $t = \{(q_0,\sigma_0),\ldots,(q_f,\sigma_f)\}$, we create its reconstruction $r = \{(\hat{q}_0,\hat{\sigma}_0),\ldots,(\hat{q}_f,\hat{\sigma}_f)\}$ under a finite game game function $\hat{G}$ by initializing $\hat{q}_0 = q_0$ and $\hat{\sigma}_0 = \sigma_0$, and iteratively extending r by following controls in the graph: $\hat{q}_{i+1} = \hat{\delta}(\hat{q}_i,\hat{\sigma}_i)$. The control $\hat{\sigma}_i$ at frame i is found by taking the control σ_j at frame j in t, where q_j is the closest state in t (measured in position only) to $\hat{q}_i$.

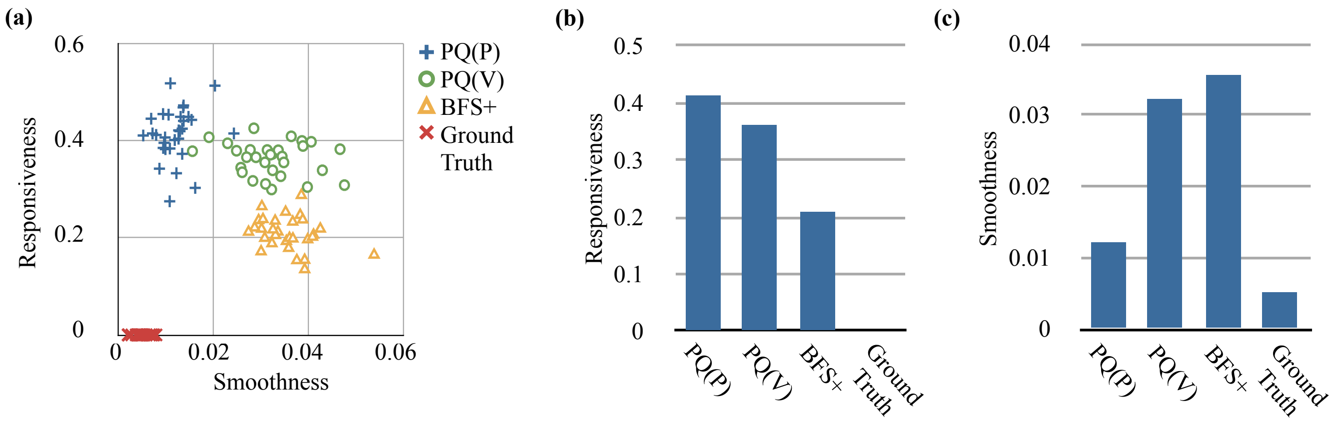

Figure 3: *An empirical analysis of tabulation algorithms demonstrates that PQ cannot achieve acceptable levels of responsiveness without reaching unacceptable levels of smoothness, while BFS+ provides both smoothness equivalent to PQ(V) and much improved responsiveness. All measurements were performed on a set of 30 playthroughs as described in Sec. 4.3. Smoothness is evaluated using RMS jerk (lower values are smoother), and responsiveness is evaluated using the fraction of approximate edges—prior to smoothing—traversed (lower values are more responsive).* ***(a)*** *Smoothness and responsiveness for each algorithm for all playthroughs.* ***(b)*** *Average responsiveness across all playthroughs.* ***(c)*** *Average smoothness across all playthroughs.*

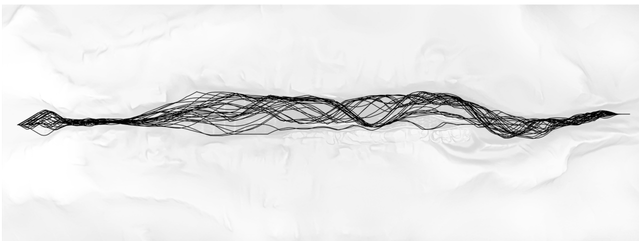

Figure 4: *A visualization of the 30 player traces we used to evaluate our game, overlaid on a top-down view of our game level.*

We measure responsiveness using the fraction of edges traversed by the player that are approximate edges—more approximate edges mean less responsive play—and smoothness using the RMS value of jerk, evaluated using first-order backward finite differences on player positions, during the playthrough. The results for all three algorithms as well as the ground-truth player traces are summarized in Fig. 3.

4.4. State Smoothing

Although the distance threshold used in state merging ensures that the error introduced by constructing $\hat{\delta}$ is bounded, in practice the ability to set a higher threshold can be beneficial to controlling the state space explosion, and even small errors in continuous dynamics can lead to noticeable discontinuities and jitter in the finite state game. We propose a method to "smooth out" the states in $\hat{Q}$ to distribute these small errors and reduce larger errors as much as possible. We refer to this as the *smoothing* step.

For time and memory efficiency, we smooth each state variable independently. We estimate the ith state variable's contribution to the total error in the graph as the sum of its squared residuals:

$$\sum_{q\in\hat{Q},\sigma\in\Sigma} ||\hat{\delta}(q,\sigma)-\delta(q,\sigma)|| \tag{2}$$

We then define the delta table Δ, which caches the distance from each source to destination state using the game simulation function δ:

$$\Delta(q,\sigma)=\delta(q,\sigma)-q \tag{3}$$

Using this table, we can construct a linear system composed of the following equation for each graph edge:

$$\hat{\delta}(q,\sigma)-q=\Delta(q,\sigma) \tag{4}$$

We constrain q_0 and the terminal states T to their original values, and compute smoothed state values as the least-squares solution in q to the resulting overconstrained system. In our demonstration game, we smooth only variables that immediately affect the rendered frame: position, velocity, and the global timer (Table 1). In practice, smoothing may not remove every instance of jitter, but does considerably reduce it. Please refer to the accompanying video for a side-by-side comparison of *BFS+* before and after smoothing.

5. State Design

We have shown how *BFS+* preserves two important game properties in the discrete setting: smoothness and responsiveness. Sec. 1, however, also lists three more properties: (3) event triggers, (4) secondary effects, and (5) background animations. Our demonstration game exhibits all these effects, including triggered growth of a bridge, grass that bends as a secondary effect, and background animation (e.g. rivulets of water running down ravine walls and leaves blowing in the breeze).

Preserving these important properties in the finite state setting requires careful coordination between the continuous game and the renderer. The challenges in coordinating these modules primarily manifest themselves in the design of the state vector, which forms the interface between them. In general, our strategy will be to first augment the continuous game with explicit state variables to support these three properties, then to tabulate the game including these state variables, and finally to use these variables to drive a high-fidelity rendering. As in Sec. 4, the key constraint will be controlling the total number of states.

We present a trio of techniques for incorporating these properties. We handle triggered events (constraint 3) by inserting *event timers* (Sec. 5.1) into the state. We handle secondary effects by designing *history-free animations* (Sec. 5.2). Finally, we describe how to compute a consistent notion of *global time* to create immersive background animations (Sec. 5.3).

5.1. Triggered Events

Triggered events are discontinuous changes in the game state caused by player actions, for example, the appearance of a magical bridge when the player approaches the edge of a cliff. Our goal is to support such events without creating too many states. A more subtle challenge is to handle such discontinuous effects in the presence of a tabulation and smoothing steps which assume continuity of the simulation function. (In Sec. 4, the transition function $\delta(q,\sigma)$ is assumed to be continuous for all q.)

To add dynamic complexity to our game, we included three different discontinuous events that the player can trigger: a gust of wind that causes hanging plants in a cave to sway, the growth of a vine bridge from a ramp to the cave, and a splash when the player lands in a pool of water. These events are either not visible at all in the Unity demo, or are approximated only crudely.

These events must still be represented in the state vector, however, so that we can include them during rendering and prevent merging between states that are similar except for event visibility. We represent each event e in the state vector using an event timer variable t_e. (These timers are distinct from the global timer described in Sec. 5.3.) An event timer t_e is initialized to $t_e = 0$ and remains paused at that value until the player passes through an event-specific trigger volume, which starts the timer. Once the timer reaches an artist-provided maximum value $T_{\max,e}$, indicating the latest point in time at which the absence or presence of the event e can be visible in the rendered game, the timer resets to $t_e = 0$, "forgetting" whether or not the animation was triggered and preventing unnecessary state duplication.

5.2. Secondary Effects

The globosome is placed at the position recorded in the state vector. However, globosome rotation is not present in the state vector, since it introduces a large number of unnecessary degrees of freedom. The rotation must therefore be inferred from the globosome's position in the level. We infer rotation along the globosome's horizontal axis parallel to the image plane only, and in fact rotate the globosome slightly slower than physically accurate, since we determined that this effect was more visually pleasing.

To increase the dynamism and responsiveness of the game world, we also introduce a variety of visual effects that are keyed off of the globosome position, but have no direct representation in the state vector. These include grass bending as the globosome rolls through it and water rippling as the globosome passes through a puddle. These effects are produced by constructing geometry deformation fields that are placed centered at the player position, and oriented using the player's velocity vector. Using the velocity vector allows us to introduce asymmetric deformation fields that, for example, depress the grass much farther behind the globosome than ahead of it. Without these deformation fields, blades of grass, for example, would remain rigid as the globosome approached and rolled over them.

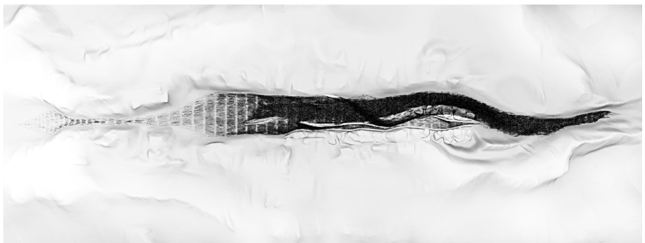

Figure 5: *A visualization of the 861,903 states computed for our demonstration game, overlaid on a top-down view of the game level.*

5.3. Background Animations

Background animations are effects that do not depend on the game state, such as flowing fountains and fluttering leaves. Since our approach requires that all animation depends on state, we include a time variable t in the state to drive these background animations. (This variable is called `global_timer` in Table 1.) Ideally, each successive state would see t incremented by the time-step, but tabulation may break this constraint leading to unnerving speedups or slowdowns in the background animation. Therefore, we treat the time variable specially.

We propose a simple heuristic solution which works for our demonstration game, but note that a more general solution is still an open problem. The key is to tie "time" to a another state variable which behaves well. For our demonstrate game, we pick the z variable which measures the globosome's progress in the principal direction of the level. To determine the canonical time for each position, we play the game several times, measuring time as a function of progress down the level, forming as set $\{(t,z)\}$. We use a smoothing nearest neighbor function average these measurements into a "time template," $t(z)$. After tabulation, we remap time according to this template, smooth the time variable (Sec. 4.4), relearn a new time template $t_{smoothed}(z)$ by averaging over all tabulated states, and again remap time. This process gives the entire game smooth background animation throughout, as can be seen in our accompanying video.

6. Results

We created a 30-second interactive game to demonstrate and analyze our technique. The game is in the "runner" genre, and follows a small, player-controlled sphere down a hill (Sec. 2). The resulting game simultaneously exhibits all five essential game characteristics we describe in Sec. 1: smoothness, responsiveness, triggered events, secondary effects, and background animation. To create this

name	**type**	**smoothed**	**visible**	**description**

`pos_x`
`pos_y`
`pos_z`
	`float`	yes	yes	player position

`vel_x`
`vel_y`
`vel_z`
	`float`	yes	yes	player velocity

`ang_vel_x`
`ang_vel_y`
`ang_vel_z`
	`float`	no	no	player angular velocity

`event_timer_0`
`event_timer_1`
`event_timer_2`
	`float`	no	yes	triggered event timers (Sec. 5.1)

`global_timer`	`float`	yes	yes	global timer (Sec. 5.3)
`cam_0` `cam_1`	`float`	no	yes	authored camera path weights; for blending
`speed_zone`	`int`	no	no	used for catchup mechanic (Sec. 5.3)
`no_ctrl`	`bool`	no	no	indicates that player control is disabled; e.g. when airborne

Table 1: *The 17 state variables for our demo game. The "smoothed" column indicates whether the variable is affected by our global smoothing process (Sec. 4.4), and the "visible" column indicates whether the variable is used directly during rendering. Two states that differ only in non-visible variables will produce identical rendered images. It is still necessary to store these variables during tabulation, in order to deserialize game states that we can use as starting points for further simulation.*

demonstration, three teams worked simultaneously on tabulation, rendering, and real-time playback. This section describes how we created and evaluated these three components.

6.1. Tabulation

The game was designed in Unity [Uni16], and instrumented to enable serialization, deserialization, and one-step simulation from any state and control. The 17-dimensional state encodes position, velocity, angular velocity, a global timer (Sec. 5.3), three event timers (Sec. 5.1), and camera coefficients and camera flags (Table 1). The tabulation software, written in Python, treats the game as a black box with three operations: saving the game state to a state vector, loading the game state from a state vector, and updating the internal game state over one timestep. We explored two different tabulation algorithms, *PQ* and *BFS+*, settling upon the latter, and created an 861,903 state tabulation (visualized in Fig. 5), well within our 1M state budget. This process took 8 hours on a single 2.6 GHz 4-core machine (an AWS EC2 `c4.xlarge` instance), including graph construction (Sec. 4.1, single-threaded, $\sim$ 7.5 hours) and smoothing (Sec. 4.4, multithreaded, $\sim$ 30 minutes).

Evaluation. The smoothed *BFS+* tabulation closely resembles the continuous-state Unity game. We invite the reader to verify this claim by running the playable demos associated with this submission (Sec. 6.3). We also evaluated these algorithms numerically, finding that while *PQ* can be tuned to trade off smoothness and responsiveness, *BFS+* is much more responsive while maintaining an acceptable level of smoothness, especially after the smoothing pass (Sec. 4.4).

Figure 6: *A user playing the prototype on a tablet.*

6.2. Rendering

We created a film-quality representation of the prototype level in Autodesk 3DS Max, parameterized by the 17 dimensional state vector. The game significantly reuses geometry, animation and the rendering pipeline from *The Rise and Fall of Globosome* by Sascha Geddert—a form of direct information transfer not possible using standard game development techniques. The scene makes heavy use of subdivision surfaces, consists of roughly 900 million primitives, and requires up to 24 GB of RAM. We rendered the entire game in preview quality, and about 30% of the frames in full quality, over multiple weeks, using up to 400 machines at a time from the Amazon EC2, Microsoft Azure and the Google Compute Engine clouds, all coordinated by the Deadline compute management system [TS16]. Average render time per image was 30 minutes on a 8-core machine; we used approximately 1 million core hours for this task.

Evaluation. With complex geometry, animation, indirect illumination, and volumetric effects, our demonstration meets or exceeds the visual quality of current AAA game titles (Fig. 1). Unlike a AAA title, however, our demonstration does not require expensive graphics processing hardware or, indeed, virtually any computation at runtime. We invite the reader to inspect our visual results in the accompanying video.

6.3. Playback

Ultimately, we expect this kind of pre-rendered large-scale finite state game will be stored in a data-center and streamed to clients, an important future direction for this work. For this work, we wrote a small playback client that runs on a Microsoft Surface Pro 3 tablet (Fig. 6). This device was chosen because it provides touch controls while also offering enough fast storage for over 400 GB of image data. The playback client loads the tabulated state machine into memory and traverses it based on user input. Pre-recorded sound effects are triggered and mixed at runtime based on annotations added to the state graphs. To assist in evaluating our method, we have made three playable Unity-rendered versions of our game available at http://graphics.cs.cmu.edu/projects/finite-state-game-engines/: one with ground truth dynamics, one with *PQ* tabulated dynamics, and one with *BFS*+ tabulated dynamics.

7. Limitations, Discussion, and Future Work

We created our demonstration game as a proof-of-concept that a modern animated film could be converted into an interactive video game through state tabulation. While we have emphasized "real-world" considerations such as responsiveness, triggered events, and background animations, our approach still has a number of limitations which future research can address.

Limited complexity. Our demonstration shows that a real-world game in the runner genre can be fully tabulated and pre-rendered. However, introducing even a modest amount of additional complexity—for example, a second player character, or a jump control—could cause a combinatorial explosion that would make tabulating the game infeasible. One solution would be decomposing and pre-rendering different game elements independently and composing them at runtime. Going a step further, a hybrid system could add dynamically rendered elements into the precomputation. Such developments would mirror previous work in model reduction in which monolithic models [TLP06, AKJ08] are decomposed into smaller pieces and recombined [WST09, KJ11].

Cost. This project presents perhaps the largest finite-state discretization ever demonstrated in the graphics literature. At this scale, the cost of rendering becomes significant. Our 860k state demo, for example, has a rendering cost equivalent to that of an 8 hour animated movie. At 30 minutes per frame, this cost could be significant for a small game studio. On the other hand, by allowing existing film assets and pipelines to be re-used in game development, our technique avoids one of the principal costs of game development: porting assets and worlds between significantly different workflows and software pipelines, or worse, regenerating assets from real life. Nevertheless, decreasing the cost of rendering remains an important and exciting area of future work for finite-state games. Decomposition (see above) could decrease the cost by decreasing the number of states required. The discrete nature of our games might even enable new rendering strategies, for example, rendering just a fraction of the frames at full resolution and interpolating the rest by combining geometric and raster information. More speculatively, game rendering could be distributed, with players "paying" for the game by rendering frames on their home devices.

Genre specificity. Elements of our technique are general, such as the tabulation algorithm (Sec. 4), and its compatibility with black-box game engines. Other aspects, however, are more genre-specific, such as our global time remapping technique (Sec. 5.3)—a result of optimizing for the runner genre. We believe that an exciting area for future research is to generalize our approach to animation to new genres and to new display devices, including Virtual Reality headsets. One possible idea is to replace linear time with a superposition of several cyclic time variables. This would require generalizing our tabulation, smoothing, and time remapping techniques to the toroidal topology of these cyclic time variables.

Lack of hysteresis. To control the number of states in the game, we rely on triggered animation (Sec. 5.1) and positional effects (Sec. 5.2) to approximate complex history-dependent phenomena. While state tabulation techniques have been shown handle complex, time-dependent phenomena [KKN*13, SHK*14], naively combining these approaches with our own could lead to a state explosion. An interesting avenue for future research would be to use traditional techniques to simulate complex phenomena, such as fluids, on top of a fixed finite-state game topology. The topology of the graph would generalize a typical two-point boundary control problem [FL04] into a multiway boundary problem defined by the topology of the graph.

Distribution. Storing our 30-second demo requires requires 400 GB of JPEG image data. At that rate, a real runner with 15 minutes of original gameplay would require about 12 TB to store, which would be impossible to distribute to or store on modern game consoles or mobile phones. Even a tenfold size decrease from video compression (optimistic due to short branching intervals) would still produce a difficult-to-distribute game. One possible solution would be to distribute this content as a physical arcade game containing the data. Our method also maps particularly well to cloud gaming. The server could store and distribute the data from the cloud performing essentially no computation per user, which would likely mean that many users could be multiplexed onto a single server—cutting hardware and compute costs dramatically and overcoming a significant shortcoming of previous cloud game approaches such as OnLive. The cloud context also presents significant and exciting new compression challenges, such as pre-compressing the image data directly on the discrete graph topology, or trying to decrease total bandwidth by storing some information (a small number of template frames, for example) locally on the client.

8. Conclusion

This paper presents a technique to combine rich animated films with highly interactive computer games. The key to our approach is a new method for approximating a continuous video game as a large finite state machine. This technique preserves smoothness and responsiveness, and can be carried out without specific knowledge of the game dynamics or implementation. The continuous game can be treated as a black box supporting only state serialization, deserialization, and single-step simulation. Our approach can bring offline rendered films to life, reusing geometry, animations, and rendering pipelines directly, rather than converting (or re-generating) these assets and then mapping the game dynamics to complex modern graphics processors.

We demonstrate this technique by designing and discretizing a 30-second runner game, set in the world of *The Rise and Fall of Globosome*, an award-winning short film by Sascha Geddert. Our technique allowed us to reuse geometric assets, animations, and rendering pipeline information from the original film without conversion. Our game can be played on a Windows tablet, demonstrating the feasibility of our approach. We also explore several methods of creating the discretization through tabulation, including a priority-queue based algorithm (PQ) based on previous work [KKN*13, SHK*14] and a new breadth-first-search based algorithm (BFS+) designed specifically to provide interactivity guarantees required for our game. We demonstrate both quantitatively and qualitatively that our new algorithm outperforms previous work, and show how careful state management allows us to implement important aspects of real games in the discrete setting, including event triggers, secondary effects, and background animation. We view this work as particularly important for cloud-based graphics. Some of the largest recent developments in computing have been the rapid spread of large-scale compute as a service, thin mobile clients, and cloud application logic. Our approach is ideally suited to the new model of cloud-based computation that these developments enable. Pre-rendering is a compute intensive but embarrassingly parallel task which can inexpensively be carried out in cloud data centers. The data, once computed, can be streamed to thin clients, overcoming their limited graphics capabilities. Moreover, because all game logic and rendering are precomputed, game data can be stored at the edge of the cloud in content distribution networks, and large numbers of clients could, in principle, be multiplexed on a single server. Multiplexing would overcome an economic hurdle to previous cloud gaming efforts, which required a single high-performance server instance dedicated to each player. We are excited to explore how tabulation can be applied to even larger-scale problems and more complex dynamic effects, and how tabulation could be extended through state decomposition—techniques we hope will inspire a new generation of interactive content and bring beautiful virtual worlds to life.

Acknowledgements

This material is based upon work supported by Amazon, Microsoft, Google, Thinkbox, and Chaos Group. We would like to thank the anonymous reviewers for their valuable comments.

References

[AKJ08] AN S. S., KIM T., JAMES D. L.: Optimizing cubature for efficient integration of subspace deformations. *ACM Transactions on Graphics 27*, 5 (Dec. 2008), 165:1–165:10. doi:10.1145/1409060.1409118. 2, 8

[Aut14] AUTODESK: 3D Studio Max 2015, 2014. http://area.autodesk.com/3dsmax2015/. 2

[BJ05] BARBIČ J., JAMES D. L.: Real-time subspace integration for St. Venant-Kirchhoff deformable models. *ACM Trans. Graph. 24*, 3 (July 2005), 982–990. doi:10.1145/1073204.1073300. 2

[CG15] CHAOS GROUP: V-Ray for 3ds Max 3.2, 2015. http://www.chaosgroup.com/en/2/vray.html. 2

[Cin83] CINEMATRONICS: Dragon's Lair, 1983. 2

[Coc15] COCOS2D-X: The Cocos2d-x game engine, 2015. http://www.cocos2d-x.org/. 2

[Cry09] CRYTEK: Crytek releases CryENGINE®3, 2009. http://www.crytek.com/news/crytek-releases-cryengine%C2%AE-3 2

[Cya93] CYAN: Myst, 1993. 2

[FL04] FATTAL R., LISCHINSKI D.: Target-driven smoke animation. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 441–448. doi:10.1145/1015706.1015743. 8

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (2003), 753–774. doi:10.1111/j.1467-8659.2003.00722.x. 2

[HV04] HAO X., VARSHNEY A.: Real-time rendering of translucent meshes. *ACM Transactions on Graphics 23*, 2 (Apr. 2004), 120–142. doi:10.1145/990002.990004. 2

[IDYN06] IWASAKI K., DOBASHI Y., YOSHIMOTO F., NISHITA T.: Real-time rendering of point based water surfaces. In *Proceedings of the 24th International Conference on Advances in Computer Graphics* (Berlin, Heidelberg, 2006), CGI'06, Springer-Verlag, pp. 102–114. doi:10.1007/11784203_9. 2

[IS11] IMANGI STUDIOS: Temple Run, 2011. 2

[JF03] JAMES D. L., FATAHALIAN K.: Precomputing interactive dynamic deformable scenes. *ACM Transactions on Graphics 22*, 3 (July 2003), 879–887. doi:10.1145/882262.882359. 2

[KJ11] KIM T., JAMES D. L.: Physics-based character skinning using multi-domain subspace deformations. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (New York, NY, USA, 2011), SCA '11, ACM, pp. 63–72. doi:10.1145/2019406.2019415. 8

[KKN*13] KIM D., KOH W., NARAIN R., FATAHALIAN K., TREUILLE A., O'BRIEN J. F.: Near-exhaustive precomputation of secondary cloth effects. *ACM Transactions on Graphics 32*, 4 (July 2013), 87:1–87:8. doi:10.1145/2461912.2462020. 1, 2, 3, 4, 8, 9

[MM13] MACKLIN M., MÜLLER M.: Position based fluids. *ACM Trans. Graph. 32*, 4 (July 2013), 104:1–104:12. doi:10.1145/2461912.2461984. 2

[Ngu13] NGUYEN D.: Flappy Bird, 2013. 2

[SHK*14] STANTON M., HUMBERSTON B., KASE B., O'BRIEN J. F., FATAHALIAN K., TREUILLE A.: Self-refining games using player analytics. *ACM Transactions on Graphics 33*, 4 (July 2014), 73:1–73:9. doi:10.1145/2601097.2601196. 1, 2, 3, 4, 8, 9

[Sip12] SIPSER M.: *Introduction to the Theory of Computation*. Cengage Learning, 2012. 3

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*, 3 (July 2002), 527–536. doi:10.1145/566654.566612. 2

[SSS09] SEMI-SECRET SOFTWARE: Canabalt, 2009. 2

[TF14] THE FOUNDRY: Nuke 9, 2014. https://www.thefoundry.co.uk/products/nuke/. 2

[TLP06] TREUILLE A., LEWIS A., POPOVIĆ Z.: Model reduction for real-time fluids. *ACM Transactions on Graphics 25*, 3 (July 2006), 826–834. doi:10.1145/1141911.1141962. 2, 8

[TS16] THINKBOX SOFTWARE: Deadline 8.0, 2016. http://deadline.thinkboxsoftware.com/. 7

[Uni16] UNITY: Unity 5, 2016. https://unity3d.com/. 2, 7

[WST09] WICKE M., STANTON M., TREUILLE A.: Modular bases for fluid dynamics. *ACM Transactions on Graphics 28*, 3 (July 2009), 39:1–39:8. doi:10.1145/1531326.1531345. 8

[YG16] YOYO GAMES: GameMaker: Studio, 2016. http://www.yoyogames.com/studio. 2