To appear in ACM TOG 33(4).

Self-Refining Games using Player Analytics

Matt Stanton[1] Ben Humberston[1] Brandon Kase[1] James F. O'Brien[2] Kayvon Fatahalian[1] Adrien Treuille[1]

[1]Carnegie Mellon University [2]University of California at Berkeley

Figure 1: *A self-refining liquid control game uses player analytics to guide precomputation to the most visited regions of the liquid's state space. The game's quality continuously improves over time, ultimately providing a high-quality, interactive experience.*

Abstract

Data-driven simulation demands good training data drawn from a vast space of possible simulations. While fully sampling these large spaces is infeasible, we observe that in practical applications, such as gameplay, users explore only a vanishingly small subset of the dynamical state space. In this paper we present a sampling approach that takes advantage of this observation by concentrating precomputation around the states that users are most likely to encounter. We demonstrate our technique in a prototype *self-refining game* whose dynamics improve with play, ultimately providing realistically rendered, rich fluid dynamics in real time on a mobile device. Our results show that our analytics-driven training approach yields lower model error and fewer visual artifacts than a heuristic training strategy.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation; I.5.1 [Pattern Recognition]: Models—Statistical; I.6.8 [Simulation and Modeling]: Types of Simulation—Gaming;

Keywords: games, data-driven animation, player models

1 Introduction

In interactive simulation, data-driven techniques trade precomputation time for runtime speed and detail, enabling stunningly realistic animation of curling smoke [Treuille et al. 2006; Wicke et al. 2009], flowing cloth [Guan et al. 2012; Kavan et al. 2011], and deforming bodies [Barbič and James 2005]. The shift towards cloud computing services provides interesting new opportunities for data-driven techniques by making it easier to perform ever more massive precomputations based on large quantities of data [Kim et al. 2013].

Data-driven methods, however, are only as good as their precomputation. Even with vast computational resources, dynamical spaces are so large that we cannot precompute everything. Fortunately, exhaustive precomputation is unnecessary: user interactions are typically structured and thus explore only a vanishingly small subset of the configuration space. This is particularly true for the focus of this paper, *games*, whose objectives can strongly shape player interactions. The main challenge is to automatically discover structure from crowdsourced interaction data and exploit it to efficiently sample the dynamical state space.

To address this challenge, we have developed a model *self-refining game* whose dynamics improve as more people play. The gameplay, controls, and objective are simple: the player tilts their mobile device and tries to cause a simulated liquid to splash through a target area of the domain (Fig. 1). Points are awarded according to the volume of the fluid passing through the target. Although the game is simple, the dynamics are not: free-surface fluids exhibit rolling waves, droplet sprays, and separating sheets which cannot be simulated and rendered in real time on today's mobile devices.

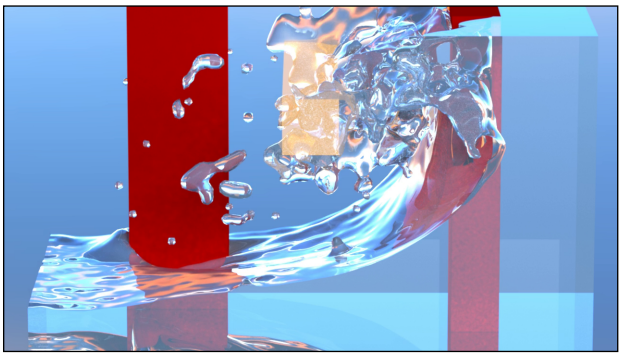

Figure 2: *Our data-driven approach enables the high quality interactive simulation of free-surface fluids on a mobile device.*

Our data-driven solution is general, applicable to any dynamical system whose controls can be represented as a selection of discrete choices at discrete time intervals. We model the game as a *state graph* whose vertices are states and whose edges are short transitions between states. At runtime, the control (in this case, phone tilt) determines which transition to follow. Typically, each transition is simulated, but because we can only precompute a finite number of transitions, some edges *blend* simulations, returning to a previously computed state. Following Kim et al. [2013], our precomputation process interactively grows the state space by successively replacing blend edges with real simulation edges and new states.

The question then becomes: which states should we explore? We show that naïve growth strategies construct vast state graphs that only barely overlap with states explored by real players; these graphs also contain significant visual errors. Using player data, however, enables a novel form of crowd-based sampling which concentrates on those states players actually visit, building significantly better state graphs with far fewer visual artifacts.

Our main contributions are as follows. We present *self-refining games*, whose dynamics are discretized into a state graph, along

with a sampling method to improve this graph based on player analytics. Such games exhibit increased fidelity as more player data is collected. Our continuous improvement process exploits a new sampling strategy incorporating real player data in order to significantly outperform previous strategies. [Kim et al. 2013] We present an algorithm, STATERANK, which estimates the global probability of each state relative to a player model, and we show how to bootstrap game construction with a simple *a priori* player model. We adapt this framework to free-surface fluids using a novel similarity metric and blending technique. Finally, we discuss the systems challenges in constructing, maintaining, compressing, and analyzing large-scale simulation data in the cloud.

2 Related Work

Crowdsourcing has become a major research topic with applications including text recognition [von Ahn et al. 2008], drawing classification [Eitz et al. 2012], and performing user studies [Kittur et al. 2008]. One important subgenre of this research studies *games* which intrinsically motivate players to perform tasks from labeling images [von Ahn and Dabbish 2004; von Ahn et al. 2006] to designing biomolecules [Cooper et al. 2010; Lee et al. 2014].

Another major application of crowdsourcing to games is using player data to improve the gameplay experience. Zook et al. [2014] tune game parameters based on gameplay traces, the DrawAFriend game [Limpaecher et al. 2013] uses data from previous players to build a drawing improvement engine for later players, and Microsoft Research's Drivatar uses traces from a racing game to improve in-game driving controllers [Microsoft 2013]. Smith et al. [2011] describe a spectrum of different player models; in their taxonomy, our bootstrap model (§4.1) is a Universal Synthetic Generative Action model, and our learned model (§4.2) is a Universal Induced Generative Action model. Learned player-specific models have been used to build player-adaptive AI [Houlette 2003], and to generate customized levels [Zook et al. 2012] and stories [El-Nasr 2007; Thue et al. 2007].

We present a new application, *self-refining games*. These games use player models learned from crowdsourced gameplay data to improve the accuracy and fidelity of the game dynamics. Self-refining games improve gameplay by generating a continually-improving sampling of the game state space, which is then played back at runtime. This playback mechanism is similar to video textures [Schödl et al. 2000], although we are not limited to a single fixed input video. It is also reminiscent of the video-playback mechanics of *Dragon's Lair* [Cinematronics 1983], although since we do not depend on human animators we are capable of generating vastly larger data sets. Our game improvement method is automatic and generalizes across a large class of games.

While our approach is general, this paper focuses specifically on precomputed fluids – a topic of extensive recent graphics research. Early monolithic fluid models [Treuille et al. 2006] have been generalized to include fluid rendering [Gupta and Narasimhan 2007], control [Barbič and Popović 2008], domain decomposition [Wicke et al. 2009], and to capture non-polynomial dynamics [Kim and Delaney 2013; Stanton et al. 2013]. Despite these advances, modeling free-surface fluids remains a challenge for data-driven simulation, due to the complexity of reducing the pressure discontinuity at the surface. We successfully model-reduce such liquids using a state-tabulation approach described below.

Our method explicitly tabulates arbitrary dynamics and rendering in an offline process. This approach was pioneered by James and Fatahalian [2003] who tabulated the dynamics of deformable models driven by a small palette of impulse forces. Kim et al. [2013] extended this approach to cloth dynamics and used far greater computational resources to form a near exhaustive portrait of clothing motion on a moving character. Our graph structure and growth process are similar to those of Kim et al., although we adapt these ideas to liquids using a new similarity measure and blend function. Rather than compress three-dimensional data, we render the liquid from a fixed viewpoint producing tens of thousands of tiny video clips which can be concatenated to form animation. In principle, however, rendering could be decoupled and performed on either the client or in the cloud. Most importantly, we show how to use player data to counteract the effectively infinite complexity of the state space.

Data-driven methods are only as good as their precomputation data. Kim and James [2009] solve this problem by incrementally building a reduced model for a specific simulation trajectory. By contrast, we attempt to capture an entire space of trajectories through a continuous state sampling process which uses game analytics to focus on that subset of the dynamics that players really explore. To achieve this goal, we estimate state visit probabilities using an algorithm, STATERANK, which computes the stationary distribution of a Markov chain [Feller 1968] with transition probabilities derived from a player model, similar to the PAGERANK algorithm of Page et al. [1999].

Other researchers have used user data to refine generative animation models. McCann and Pollard [2007] used gameplay traces to select transition from a fixed motion graph, and Cooper et al. [2007] focused on computer-in-the-loop sampling of human motion capture data for a fixed set of known objectives. In contrast, we learn models of human player behavior in order to refine game dynamics through adding new transitions to a continually-expanding state graph.

Conventional simulation has also been adapted to capture real-time fluids. Approaches include 2D shallow-water methods [Šťava et al. 2008; Thurey et al. 2007; Chentanez and Müller 2010], particle-based methods [Macklin and Müller 2013], grids [Crane et al. 2007], and hybrid methods [Chentanez and Müller 2011]. Typically, conventional fluid simulation offers greater flexibility than data-driven methods in exchange for far higher runtime costs. By contrast, our crowdsourcing approach enjoys the same runtime efficiency as other data-driven methods while enabling, in principle, simulation of arbitrary trajectories.

3 State Graphs

We now describe a method for creating self-refining games based on learned player behavior. The foundation for these games is the *state graph*, whose vertices are game states, and whose edges are transitions induced by player actions. These graphs are similar to the secondary motion graphs of Kim et al. [2013]. A game could be represented by a single graph, or could use multiple graphs to represent different areas, levels, or challenges. To initialize a new state graph we use a heuristic player model to bootstrap synthesis of a minimally playable experience. Then, we make the game available to players and collect traces of the paths they take through this graph during play. We use these paths to learn a model of player behavior that is used to prioritize graph growth, adding states and increasing fidelity in those regions players are most likely to visit. By repeatedly collecting player data, updating our player model, and using the updated model to grow the graph, we create a game that continually improves over time. This process is general, and can be applied to any precomputed game with discrete control.

In this section, we describe our game model and the mechanics of graph growth. In the next section (§4), we describe how we learn

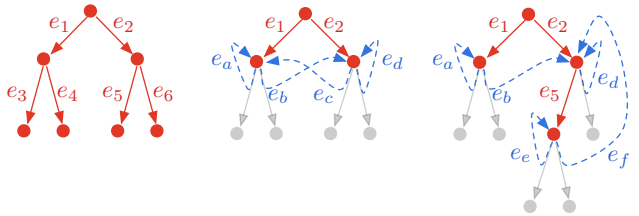

Figure 3: *State graph initialization and growth. Solid edges correspond to simulations and dashed edges correspond to blends. Left: We initialize the graph by sampling a tree of simulation data. Center: We transform the tree into a complete state graph by replacing dead-end edges with blends. For example, e_c is formed by blending dead end e_5 with e_1. Right: We expand the graph to remove error along blend e_c by removing e_c, re-inserting e_5, and simulating new animations from e_5 for each control. The new simulations form new dead ends that are blended to create e_e and e_f. A self-refining game repeats this last step continuously.*

player behavior models from collected player data. The following section (§5) applies this general framework to free-surface fluid simulation.

In a state graph, each edge is associated with an animation connecting its source and destination states. The format of these animations is application-dependent, but they could be video clips, sequences of triangle meshes, or any other encoding of the dynamics that we wish to display. We consider games that sample player input from a set of N discrete controls, so that each vertex has N outgoing edges. At runtime, the system replays edge animations, determining which branch to take based on player control. If the transition edges are sufficiently short (1/3 of a second in our application) the game feels interactive. We discuss the choice of transition edge durations further in §8.

To initialize a new state graph, we begin at a start state and simulate every possible outgoing transition. We continue this process, generating N new simulations from each state until we create a small N-ary tree (Fig. 3, left). Leaf edges represent dead ends in the state graph which we eliminate by blending with interior edge transitions leading back to an internal node. This blending procedure turns the tree into a complete state graph (Fig. 3, center). Since we begin with only a small tree, it is likely that many of these blends were between dissimilar edges, and therefore of low quality.

We improve the quality of the graph by growing it using new simulation data, similar to [Kim et al. 2013]. We grow the graph by replacing blend edges with simulation edges. To add a simulation edge e to the graph, we simulate the N outgoing transitions which follow e and blend these simulations into interior simulation edges (Fig. 3, right) selected to minimize an estimate of the perceptual error of the blend. Growing the graph can be a continuous process, going on as long as we have space to store the results of new simulations. The key challenge is determining which blend edges to replace.

We can view this question as one of graph quality evaluation. If we can quantify each edge's contribution to the quality of the graph, a simple strategy to reduce error is to greedily replace the blend edge most detrimental to the quality of the graph. The behavior of this strategy depends strongly on which measure of graph quality is selected. Kim et al. [2013] use a worst-case quality measure: $\max_{e \in B}(\mathrm{err}(e))$, where B is the set of blend edges in the graph and err is an application-defined estimate of a blend edge's perceptual error. This metric, which we call BASELINE, suggests that we should always replace the blend edge $e_{\max}$ with the highest error.

In a simulation-based game, however, the game occupies a huge state space, and the game objective encourages players to pursue strategies that lead to rewards. Therefore, it is likely that players will never visit the vast majority of the state space, rendering most of BASELINE's additions to the graph wasteful. Instead, we propose a different metric, STATERANK, which measures the *expected* error: $\sum_{e \in B} P(e)\mathrm{err}(e)$, where $P(e)$ is the probability of traversing the edge e. This metric suggests we replace the blend edge $e_{\exp}$ with maximum expected error $P(e_{\exp})\mathrm{err}(e_{\exp})$. We infer $P(e)$ from a *player model* $P(c|v)$ giving conditional probabilities of controls c at each vertex v. Similar to the PAGERANK [Page et al. 1999] procedure for ranking webpages, STATERANK computes edge probabilities $P(e)$ as the normalized first eigenvector of the transition matrix implied by $P(c|v)$.

If STATERANK correctly predicts edge probabilities, then this procedure will improve graph quality around precisely those states players are most likely to visit. To accurately estimate $P(e)$, however, we must have accurate estimates of the player control probabilities $P(c|v)$. In the next section we discuss how we learn this player model $P(c|v)$ from analytics data.

4 Player Model

The previous section explains how we can use a player model $P(c|v)$ to improve our sampling of huge state spaces. In this section we describe both how we can learn player models from data and how we use these models to create self-refining games. We use two different player models: a heuristic model to bootstrap the simulation, and a learned model to guide our exploration.

4.1 Bootstrap Model

When the game is first created, no player data exists. To bootstrap state graph growth, we use a heuristic player model $P_h(c|v)$ which essentially guesses what players will do. Many heuristics are possible, and the best heuristic will vary by application. In this study, we maintain the current control with probability α and otherwise choose an alternate control uniformly at random:

$$P_h(c|v) = \begin{cases} \alpha & \text{if } c = c_v \\ (1-\alpha)/N & \text{otherwise.} \end{cases} \quad (1)$$

Combining this heuristic player model with STATERANK produces a growth strategy we call SR-HEURISTIC. This simple model can initialize a state graph, but performs poorly when used exclusively to generate a full game (§7.2). We therefore propose using the heuristic model only for bootstrapping, then growing the graph using a player model learned from gameplay traces.

4.2 Player Analytics Model

Once we have a bootstrap model, we can begin to collect gameplay traces to learn a more accurate player model. We learn our player model from traces of player traversals of the state graph, each trace consisting of a list of vertices visited and the control selected at each vertex. Let $P_{\mathrm{obs}}(c|v)$ be the observed conditional control probabilities computed by normalizing control counts at v, and $P_{\mathrm{obs}}(v)$ be the observed probability of visiting v, obtained by normalizing the number of visits to v by the total number of vertex visits. To generalize our model to unvisited states, we assume that players will take similar actions in states that resemble each other closely. This observation leads us to implement our player model using a kernel

density estimator combined with a Markov prior with weight ϵ:

$$P(c|v) \propto \sum_{\substack{u \in V \\ c_u = c_v}} w_u P_{\text{obs}}(c|u) P_{\text{obs}}(u) \tag{2}$$
$$w_u = k_{\text{tri}}\left(r, \text{pdist}\left(u, v\right)\right) + \epsilon,$$

where $k_{\text{tri}}(r, x)$ is a triangular kernel with radius r, c_u and c_v are the controls of the simulation clips generating u and v, V is the set of vertices in the graph, and pdist is an inexpensive distance function (§5.2). Note that the condition $c_u = c_v$ in the summation effectively creates a different player model for each control. Also observe that this model can be evaluated even when v has not been visited by any player. As a result, this model can be used to guide sampling deep into the graph without having to wait for new player data at every step. The model can even be used to transfer predicted player behaviors gathered on one graph to explorations of other similar graphs.

Combining this player model with our StateRank technique described in the previous section yields a crowdsourced graph quality measure, SR-Crowd, which we can use to select blend edges to replace as we grow the graph. We evaluate state graphs generated using this player model in detail in §7.

5 Application to Liquids

In this section, we describe our construction of a liquid simulation game using the generic game precomputation framework that we described in §3. Our liquid simulations are generated using PCISPH [Solenthaler and Pajarola 2009]. We represent the liquid state at graph vertices v as lists of liquid particle positions and velocities, and k-frame animations along graph edges as sequences of signed distance functions $e = \left[\phi^1, \ldots, \phi^k\right]$, generated from particle data using the method of Zhu and Bridson [2005]. Each edge also has an associated video rendered using Mitsuba [Jakob 2010]. In our application $k = 10$, which yields transitions of 1/3 of a second. This latency in response to changes in player control is acceptable for our liquids game, however, different game mechanics entail different latency requirements. [Claypool and Claypool 2010]

We use two different metrics. For blending, we use a function $\text{dist}(e_i, e_j, c)$ based on energy and detailed liquid shape information (§5.1). Our player model, however, requires more frequent distance computations, so we use a more efficient metric $\text{pdist}(e_i, e_j)$ which compares only coarse shape descriptors (§5.2). Finally, we describe our clip blending function $\text{blend}(e_i, e_j)$ in §5.3.

5.1 Edge Distance

We define $\text{dist}(e_i, e_j, c)$ to be a perceptually-motivated error function incorporating information both about the liquid's shape and its energy:

$$\text{dist}(e_i, e_j, c) = \text{norm}_e(e_i, e_j)\left(\text{dist}_s\left(e_i, e_j\right) + w_e \text{dist}_e\left(e_i, e_j, c\right)\right). \tag{3}$$

Here, dist_s and dist_e denote the parts of distance attributable to the shapes and the energies of the two states, respectively; norm_e is a normalization term that increases distance at low energies, reflecting that fact that errors are easier to perceive when the liquid is moving more slowly. The weight w_e controls the relative priority of the shape and energy terms of dist. We set w_e so that for edges r_i and r_j where the fluid is nearly at rest,

$$\text{dist}_s\left(r_i, r_j\right) \approx w_e \text{dist}_e\left(r_i, r_j, c\right).$$

Shape distance. The dist_s metric penalizes the blending of animations which contain liquid in very different shapes. It is the sum of the volumes of the symmetric difference ($X \triangle Y = X \cup Y \setminus X \cap Y$) between each animation's liquid volumes at each frame:

$$\text{dist}_s(e_i, e_j) = \sum_{f=1}^{k} \text{vol}\left(\phi_i^f \triangle \phi_j^f\right). \tag{4}$$

Energy distance. The dist_e metric penalizes the blending of animations that have very different energies, and it strongly penalizes blending an animation with low energy into an animation with high energy, thus enforcing conservation of energy. Omitting dist_e can result in the formation of small loops in the state graph far away from energy minima, which look extremely unnatural.

We define energy at a vertex v as $E(v, c) = T(v) + V(v, c)$, where T is kinetic energy, V is potential energy and c is the incoming control. Notice that energy depends on the current control since selecting a gravity vector will change the potential energy. Let v_i and v_j be the destination vertices (final frames) of e_i and e_j. The energy error between edge e_i and e_j is given by

$$\text{dist}_e(e_i, e_j, c) = \gamma |E(v_i, c) - E(v_j, c)| \tag{5}$$
$$\gamma = \begin{cases} c_{\text{gain}} & \text{if } |E(v_i, c) - E(v_j, c)| < T_0 \\ c_{\text{loss}} & \text{if } |E(v_i, c) - E(v_j, c)| \geq T_0 \end{cases}$$

where $c_{\text{gain}} \gg c_{\text{loss}}$, and T_0 is approximately the residual kinetic energy of the fluid when it is visually at rest. We attempt to match the energies as closely as possible, rather than anticipating an energy loss, since we are comparing energies between two clips at identical points in time. We place the threshold between the minor energy loss penalty and the major energy gain penalty at T_0 to avoid penalizing blends between visually indistinguishable animations of static fluid.

Energy normalization. We normalize the previous two terms by multiplying them by norm_e. Let v_i and v_j again be the destination vertices of e_i and e_j, c_i and c_j be their controls, and $T_{\text{avg}} = \frac{1}{2}\left(T(v_i) + T(v_j)\right)$. Then

$$\text{norm}_e(e_i, e_j) = \begin{cases} 0 & \text{if } T_{\text{avg}} < T_0 \text{ and } c_i = c_j \\ \frac{1}{\sqrt{T_{\text{avg}} + T_0}} & \text{otherwise.} \end{cases} \tag{6}$$

Note that this implies that the distance between two edges with the same control and kinetic energy below T_0 is 0. Again, this threshold prevents the unnecessary exploration of liquid states that are visually at rest. This unnecessary exploration would otherwise consume the vast majority of our exploration effort since norm_e goes to infinity as the fluid energy goes to zero.

5.2 Player Model Distance

StateRank requires us to perform neighbor searches using a brute-force scan of all vertices in the graph, so the function we use to compute vertex distances must be fast. We therefore use a more efficient coarse shape similarity function pdist in our player model. We compute a shape descriptor d_i for each edge e_i by dividing the fluid domain into a $6 \times 6 \times 6$ grid and computing the average fraction of each cell that is occupied by liquid, and define

$$\text{pdist}(e_i, e_j) = ||d_i - d_j||_2. \tag{7}$$

5.3 Blending

We construct animations for blend edges by blending signed distance functions. A simple linear interpolation works well in cases where the fluid surfaces do not contain many fine features. However, in the presence of droplets, splashes, and thin sheets, linear interpolation can cause popping artifacts at the beginning and end of the blend. We remedy this problem by blending, using convex combinations of three signed distance functions: the source, ϕ_s, the destination, ϕ_d, and the union of their shapes, $\min(\phi_s, \phi_d)$. We use the following blend function, where $0 \leq t \leq 1$ denotes the position in the blend, clip clips its argument to lie between 0 and 1, and ℓ is a parameter that limits the blending coefficient applied to the union:[1]

$$\begin{aligned} \text{blend}(\phi_s, \phi_d, t) &= w_s\phi_s + w_d\phi_d + w_{s\cup d}\min(\phi_s, \phi_d) \qquad (8) \\ w_s &= \text{clip}\left((1 + \ell - 2t)/(1 + \ell)\right) \\ w_d &= \text{clip}\left((2t + \ell - 1)/(1 + \ell)\right) \\ w_{s\cup d} &= 1 - w_s - w_d \end{aligned}$$

In our implementation, we use $\ell = 0.1$ to avoid perceptible increases in liquid volume during blends.

6 Implementation

We constructed a distributed simulation system to carry out the large-scale state graph explorations required for our work. This system consists of a pool of worker nodes, which perform simulation and render animations, and a master node, which orchestrates computation by maintaining the graph structure, computing edge priorities and distances, and assigning work to the workers. We deploy this system on Amazon EC2 in configurations featuring up to 40 worker nodes. In total for our high-viscosity and low-viscosity experiments, this system performed over 8600 CPU-hours of computation and generated over 1.6 TB of data, at a cost of approximately $500 in compute time.

The system initializes graph exploration with a minimal state graph containing one vertex and N edges. Graph expansion then proceeds as described in §3. During exploration, the master maintains a work queue enumerating blend edges to explore. When workers become available, the master extracts the highest priority blend from the queue and assigns the corresponding simulation tasks to a worker. When workers return simulation results to the master, the master computes blends for the newly simulated edges, then updates the graph. In StateRank-based explorations, the master periodically discards blend-edge priorities, recomputes StateRank on the current graph, then uses the results to re-initialize blend edge priorities.

6.1 Optimizations

A number of key optimizations were necessary to achieve high system performance, as well as to ensure high-quality graphs.

Lazy relinking. Each time a new edge is added to the graph, nearest neighbor relationships among edges in the graph may change. Since it would be costly to recompute optimal graph blends after each new inserted edge, our implementation does not attempt "relinking" of existing blend edges when new edges are created. Instead, prior to exploring any blend edge, the system first attempts to relink this edge with existing edges in the graph. If a superior blend can be found in the graph at this time, the new blend is immediately created and used to replace the existing blend. Lazy relinking ensures that simulation is only performed on edges for which there are no good blend candidates in the graph, but it does not incur the overhead of unnecessarily reevaluating edge nearest neighbors after new edges are simulated.

Pre-publish relinking. In our experiments we make graph data available for play ("publishing") at select checkpoints during exploration. In order to provide the best possible play experience, before playing a graph we attempt to relink every blend edge. This process ensures that all graph blends are the best possible blends given available data at the time of play.

Animation caching. The primary scaling bottleneck of our system is the high cost of distance computations during edge nearest-neighbor search. These computations must be performed by the master each time a worker returns new simulation data (to create blends), and they are expensive because they require fetching fluid-volume occupancy data from network storage. Rather than incur substantial system complexity by parallelizing the master node's execution, we were able to accelerate distance computations needed for nearest neighbor search by caching voxel occupancy information in memory (our implementation uses `memcached`).

Energy pre-filtering. Even with the caching optimizations described above, nearest-neighbor search remains expensive. (For example, linear-time nearest neighbor search makes graph relinking a quadratic-time operation, which is unacceptable even for graphs of moderate size.). To accelerate nearest-neighbor search we only perform full distance evaluation on the k-closest graph edges according to our energy distance metric described in §5.1. It is important to set k appropriately; setting k too low can result in a failure to find good blends even if high-quality blend targets do exist in the graph. We have found $k = 100$ to work well in practice.

6.2 Mobile Client

We make games available to players using an Android client application. The key feature of this client is its ability to continuously play back short ($1/3$ second) videos without lag between them. When a player selects a game, the client downloads the most recent version of the game's state graph, then downloads and caches any videos for edges in the current graph that it has not already cached. We use device accelerometer data to select game controls. After each play session, the client uploads a list of visited graph vertices and the control selected on each visit to our server.

7 Evaluation

We use the self-refining liquid control game described in §1 to evaluate the utility of StateRank and player models learned through crowdsourced data to explore a large dynamic space. Recall that the player's objective in this game is to interactively tilt their device so as to splash fluid through a target region of the domain. The game admits three possible controls (N=3) corresponding to holding the device level and tilting it to the left and right. A sequence of screen shots from the game is shown in Fig. 1, with the target region in the upper-middle part of the space highlighted.

We grew state graphs for our fluid game using three different graph error measures. The SR-Heuristic and SR-Crowd measures described in §4 prioritize growth using StateRank and either a heuristic or crowdsourced player model, respectively. The simplified Baseline measure does not use StateRank to globally prioritize exploration. Instead, it only prioritizes growth using local conditional control probabilities, $P(c|v)$. To provide a better comparison with StateRank, we scale the maximum error in our Baseline metric by the conditional probabilities in Equation 1

[1] The results in the accompanying video show an incorrect version of the blend function that shortens the blend time by one frame at each end of the transition.

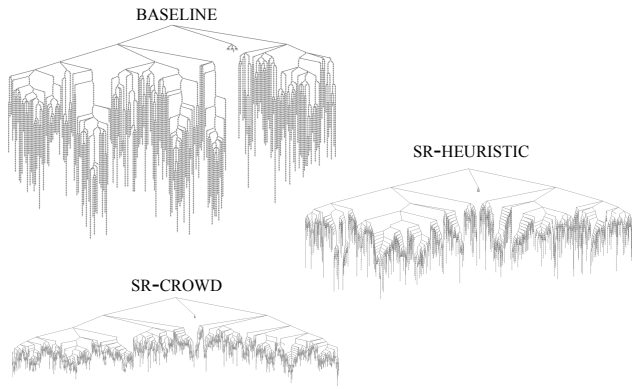

Figure 4: *200K-frame state graphs.* BASELINE *explores long chains of low energy states. In contrast, both* SR-HEURISTIC *and* SR-CROWD *prioritize more likely high energy states, yielding a shallower graph structure.*

with $\alpha = 0.8$, but we do not calculate edge probabilities as in SR-HEURISTIC or SR-CROWD.

All explorations use the energy-based fluid distance metric and fluid animation blending techniques described in §5. Our fluid simulations are 42K-particle PCISPH simulations. In addition to the results analyzed here, we performed experiments on a high viscosity fluid configuration in order to observe system behavior under a second set of fluid parameters; we show results in the video.

We grew each of our graphs until graph size reached 200K frames. (Approximately 4,300 CPU-hours were used, per graph, to compute each graph's 1.8 hours of animation). We paused graph expansion at 20K, 50K, 100K, and 200K frames so that the graphs could be played by a group of six test players, yielding gameplay traces for all graphs at these checkpoints. After collection, the checkpoint traces for the SR-CROWD graph were used to calculate per-vertex conditional probabilities that informed the exploration of SR-CROWD until the next checkpoint.

7.1 Predicting Player Behavior

To assess the predictive power of our models, we used the 200K-frame BASELINE graph to evaluate SR-HEURISTIC's and SR-CROWD's predictions of player behavior and overall graph quality against the observed data from gameplay. Fig. 8 (left, center) illustrates predicted play behavior by coloring graph edges according to predicted visit densities. At right, we show the observed probabilities from player data (ground truth). It is clear that SR-CROWD predicts behavior far more accurately than SR-HEURISTIC. This entails a better estimate of the expected blend error that players encounter: SR-CROWD's prediction is within 10% of the observed value, while SR-HEURISTIC underestimates it by nearly 50%. Note also that very little of the graph is actually visited by players since BASELINE has explored many "unimportant" regions of the state space. Guiding exploration with player data stands to create a graph that better samples the played regions.

Fig. 4 shows the 200K-frame state graphs produced by the BASELINE, SR-HEURISTIC, and SR-CROWD; the graphs indeed exhibit noticeably different structure. Specifically, the BASELINE graph is approximately twice as deep as the STATERANK-produced graphs. While the BASELINE graph samples many long sequences of constant-control play (as per the heuristic model), our game's design does not encourage this form of play, so these paths do not appear in the SR-CROWD graph.

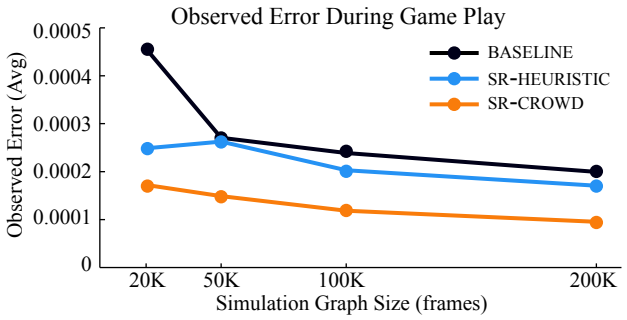

Figure 5: *On average, test players observed the lowest error while playing the sequence of graphs generated by* SR-CROWD.

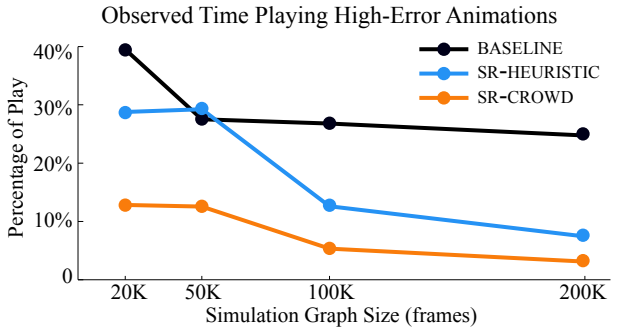

Figure 6: *Animations observed by players of the 200K-frame* SR-CROWD *graph exceeded our empirical high-error threshold during only 3% of play time. Gameplay for the similarly-sized* BASELINE *graph presented high-error animations ten times as often.*

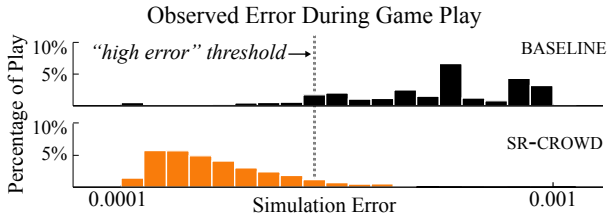

Figure 7: *Histogram of blend errors observed by players on two of the 200K-frame graphs. In contrast to* BASELINE*, the error distribution for* SR-CROWD *is largely concentrated below our high-error threshold.*

The unique structure of the SR-CROWD state graph results in a higher quality game experience. Fig. 5 plots the average error observed by players at all graph sizes. (Error is defined using the perceptually-motivated metric described in §5.) While SR-HEURISTIC alone provides a modest benefit over the BASELINE method, average observed error is nearly a factor of two lower for SR-CROWD games. The gameplay data acquired at each checkpoint during the graph growth process helps focus simulation effort on the state space regions that players are most likely to encounter.

7.2 Error Analysis

While state explorations prioritized by STATERANK successfully reduce observed error on average, they risk leaving severe blend edges in the graph if they are deemed unlikely to be visited. While small animation errors are difficult for a player to notice, large er-

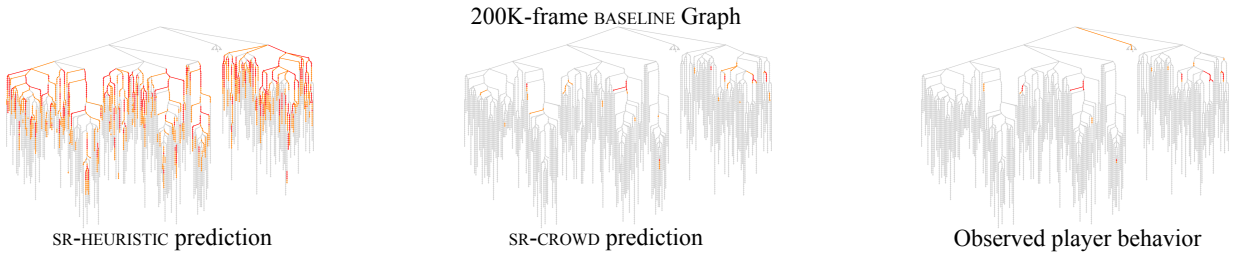

Figure 8: *Visualization of edge-visit densities predicted by* SR-HEURISTIC *(left), by* SR-CROWD *(center), and the densities from recorded gameplay (right) for a 200K-frame* BASELINE *graph. Edges visited at least 10% as frequently as the most visited edge are red, edges visited 2-10% as frequently are orange, and all others are gray. Blend edges are hidden to reduce clutter.* SR-CROWD *accurately predicts player visit densities while* SR-HEURISTIC *does not, indicating the value of player analytics in informing* STATERANK*'s probability estimates.*

rors, even if infrequent, can significantly reduce the perceived quality of the game. Through gameplay testing, we empirically determined that blend animations with error scores exceeding 0.0005 corresponded to visually objectionable animations. We plot the fraction of time that players spent viewing these high-error animations in Fig. 6.

In the large 200K-frame graph produced by SR-CROWD, players view high-error animations significantly less often than in the other methods (only 3% of the time). In fact, gameplay for the small 20K-frame SR-CROWD graph showed high-error animations less often than play through a BASELINE graph ten times as large. The fraction of observed high-error frames does not significantly diminish in the BASELINE graphs as they grow beyond 50K frames. A more detailed view of the distribution of errors encountered when playing 200K-frame BASELINE and SR-CROWD graphs is provided in Fig. 7.

Our experiences playing the games corroborate these numerical results. In all cases, playing the SR-CROWD games showed the highest quality animation. Play through the 200K-frame SR-CROWD graph for high viscosity simulations reveals virtually no artifacts at all. In low viscosity simulations, the fluid exhibits more geometric variety, and artifacts are occasionally visible (as indicated by Fig. 6), but their frequency is greatly reduced in SR-CROWD compared to other methods. As desired, animation is of the highest quality when the player plays with intention, attempting to score maximum points, and thus conforming closely to the actions of previous players. We refer the reader to the accompanying video to inspect the quality of the generated animations.

As a final experiment, we also measured SR-CROWD's ability to predict player behavior on the 200K-frame graph that it produced. This prediction is compared to ground truth observed player data in Fig. 9. Interestingly, although SR-CROWD still produces an expected edge error within 10% of the observed value, its predictions of player visit behavior on this final graph are relatively poor as compared to its predictions on the baseline graph (Fig. 8). We hypothesize that although our player model is accurate, self-referentially growing a graph based on this model amplifies small inaccuracies, even while significantly lowering error. Thus, Fig. 9 suggests that creating more sophisticated models of player behavior from the acquired play data could aid in sampling the game state space even more efficiently.

8 Limitations

Our game was designed as a simple research vehicle both to explore the potential of leveraging crowdsourced player data to efficiently sample large state spaces, and to demonstrate a new form of self-refining game using complex 3D liquid dynamics as a primary element. Our results demonstrate that models built from player data can concentrate precomputation in an important (but tiny) subset of the full state space, a result that overcomes a major hurdle in scaling data-driven techniques. Nevertheless, our current approach has several limitations which we hope further research can address.

Total dynamic complexity. To our knowledge, our fluid example represents one of the most complex systems ever precomputed at a large scale. However, even simple generalizations of the dynamics would overwhelm our system. For example, inserting floating objects would explode the state space. We emphasize that our completely monolithic technique of precomputing everything about a simulation state represents just one (extreme) end of a spectrum of approaches. Precomputed and live elements can be decomposed, composited together, and even coupled. Precomputed systems could also be generalized (e.g., using multi-way blends), potentially turning our discrete state graph representation into a continuous space of precomputed dynamics.

Limits of control. Our system offers only a small number of discrete controls and samples control to 1/3 of a second. Increasing the temporal or spatial control resolution not only explodes the state space, but also causes specific technical problems. Because our state graphs have complete N-way branching at every vertex, the blend edge fraction is $(N-1)/N$. In the limit of increasing control resolution, nearly every clip will be a blend. We believe the solution is to sparsely sample control, inserting control branches only when player data indicates they are necessary. At run-time, the system would trade off simulation and control error. It may be possible to even scale this approach to (multi-dimensional) continuous controls. Similarly, clip length has implications for blending. As the clip length approaches one frame, blends become jump discontinuities. We believe this issue could be addressed by globally optimizing state graphs to ensure smooth transitions.

Range of applicable phenomena. Our approach assumes it is possible to meaningfully blend two simulations without obvious visual artifacts. Fluids work well because the eye often ignores errors in turbulence. Some phenomena might be less forgiving. We observe, however, that blending has been successfully applied to a wide range of phenomena, from human motion graphs to image morphing. Blending should work for any sufficiently well-sampled continuous phenomenon, making the ability to densely sample important subsets of the state space – the very goal of this paper – even more important.

Single-viewpoint rendering. We chose to include rendering in our precomputation to achieve rich visual effects. This decision constrains our game to single-viewpoint rendering. However, rendering could be decoupled from the precomputation and performed either on the server or the client. A server-based rendering system would stream rendered images from any viewpoint. Client-based

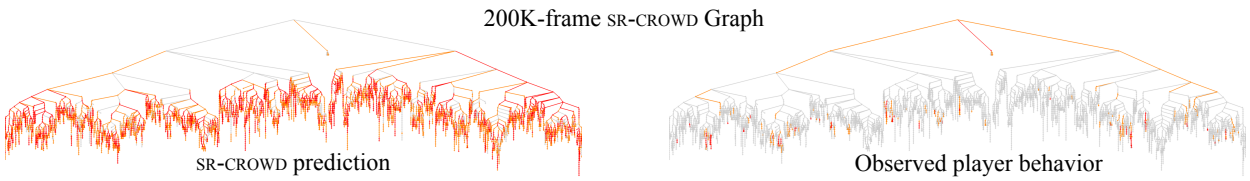

Figure 9: *Edge-visit densities predicted by* SR-CROWD *(left) and the recorded densities from actual gameplay (right) for a 200K-frame* SR-CROWD *state graph. While its prediction of the expected edge blend error remains accurate,* SR-CROWD *no longer predicts player behavior on this graph well, suggesting an opportunity for better prediction models based on the acquired player data.*

rendering is also possible, although efficient compression of 3D data would be required, a particular challenge for fluid data with temporally changing surface topology.

Storage requirements. Our 200K-frame state graphs correspond to about 5 GB of data. Unlike our prototype, practical implementations of complex data-driven games would likely rely on cloud-based animation storage and streaming. We used off-the-shelf video compression for simplicity, but our video corpus contains enormous redundancy across clips that standard video compression methods do not exploit. Deduplicating similar video sequences (perhaps by linear dimension reduction, or by applying frame prediction between clips) could potentially yield vast savings.

Applicability to existing games. Some types of open-world games encourage exploration and the discovery of new experience and content. However, many categories of games do encourage stereotyped player behavior, and even games that prioritize novelty will likely feature substantial overlap in player behaviors that our method can exploit.

9 Conclusion

This paper presents a first step towards *self-refining games* whose dynamics continuously improve based on player analytics. We observe that game objectives cause players to explore only a small fraction of the entire state space, making data-driven simulation feasible even for complex dynamical systems. We adapt the data-driven simulation method of Kim et al. [2013] to liquids, and replace the precomputation phase with a continuous process that concentrates state sampling in the subset of the dynamics that players really explore.

We compare three strategies to sample the game dynamics and show that using real player data (SR-CROWD) significantly outperforms both a more simplistic player model (SR-HEURISTIC) and a baseline model without player data (BASELINE). Interestingly, even our best player model significantly mispredicts player actions (Fig. 9), suggesting that further improvements are possible. Nevertheless, our results strongly indicate that player data can be successfully exploited to capture very complex dynamical systems.

Our method is well suited to mobile platforms with limited control precision and computational capacity. Player-driven state sampling enables us to deliver high quality rendered content in realtime with bounded simulation error. In addition to improving existing games, these ideas could enable a new class of cloud-based games where designers no longer have to worry about simulating and rendering the world in fractions of a second.

The ideas presented in this paper suggest several interesting questions and generalizations. How much player data is required to sufficiently sample a space? How does adding states affect the difficulty and the strategy of the game? How can we adapt our sampling approach to applications beyond games? Further research could yield more powerful techniques to composite precomputed dynamics models like ours with other virtual elements, create more flexible models through decomposition, decouple rendering from simulation, and address other limitations (§8). More generally, we hope that player-driven state sampling provides practitioners with a powerful new tool to create compelling and immersive virtual worlds.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. IIS-0915462 and IIS-0953985, and by generous gifts from Google, Intel, NVIDIA, and Pixar. We would like to thank Doyub Kim for his SPH simulator and the anonymous reviewers for their valuable comments.

References

BARBIČ, J., AND JAMES, D. 2005. Real-time subspace integration for St. Venant-Kirchhoff deformable models. *ACM Trans. Graph. 24*, 3 (July), 982–990.

BARBIČ, J., AND POPOVIĆ, J. 2008. Real-time control of physically based simulations using gentle forces. *ACM Trans. Graph. 27*, 5 (Dec.), 163:1–163:10.

CHENTANEZ, N., AND MÜLLER, M. 2010. Real-time simulation of large bodies of water with small scale details. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, 197–206.

CHENTANEZ, N., AND MÜLLER, M. 2011. Real-time Eulerian water simulation using a restricted tall cell grid. *ACM Trans. Graph. 30*, 4 (July), 82:1–82:10.

CINEMATRONICS, 1983. Dragon's Lair. [Arcade].

CLAYPOOL, M., AND CLAYPOOL, K. 2010. Latency can kill: Precision and deadline in online games. In *Proceedings of the First ACM Multimedia Systems Conference*.

COOPER, S., HERTZMANN, A., AND POPOVIĆ, Z. 2007. Active learning for real-time motion controllers. *ACM Trans. Graph. 26*, 3 (July).

COOPER, S., KHATIB, F., TREUILLE, A., BARBERO, J., LEE, J., BEENEN, M., LEAVER-FAY, A., BAKER, D., AND POPOVIĆ, Z. 2010. Predicting protein structures with a multiplayer online game. *Nature 466* (August).

CRANE, K., LLAMAS, I., AND TARIQ, S. 2007. *Real Time Simulation and Rendering of 3D Fluids*. Addison-Wesley, ch. 30.

EITZ, M., HAYS, J., AND ALEXA, M. 2012. How do humans sketch objects? *ACM Trans. Graph. 31*, 4 (July), 44:1–44:10.

EL-NASR, M. S. 2007. Interaction, narrative, and drama: Creating an adaptive interactive narrative using performance arts theories. *Interaction Studies 8*, 2 (June), 209–240.

FELLER, W. 1968. *An Introduction to Probability Theory and Its Applications*. Wiley.

GUAN, P., REISS, L.AND HIRSHBERG, D., WEISS, A., AND BLACK, M. J. 2012. DRAPE: DRessing Any PErson. *ACM Trans. Graph. 31*, 4 (July), 35:1–35:10.

GUPTA, M., AND NARASIMHAN, S. G. 2007. Legendre fluids: A unified framework for analytic reduced space modeling and rendering of participating media. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, 17–25.

HOULETTE, R. 2003. Player modeling for adaptive games. In *AI Game Programming Wisdom 2*, S. Rabin, Ed. Charles River Media.

JAKOB, W., 2010. Mitsuba renderer. http://www.mitsuba-renderer.org.

JAMES, D. L., AND FATAHALIAN, K. 2003. Precomputing interactive dynamic deformable scenes. Tech. Rep. CMU-RI-TR-03-33, Carnegie Mellon University Robotics Institute.

KAVAN, L., GERSZEWSKI, D., BARGTEIL, A. W., AND SLOAN, P.-P. 2011. Physics-inspired upsampling for cloth simulation in games. *ACM Trans. Graph. 30*, 4 (July), 93:1–93:10.

KIM, T., AND DELANEY, J. 2013. Subspace fluid re-simulation. *ACM Trans. Graph. 32*, 4 (July), 62:1–62:9.

KIM, T., AND JAMES, D. L. 2009. Skipping steps in deformable simulation with online model reduction. *ACM Trans. Graph. 28*, 5 (Dec.), 123:1–123:9.

KIM, D., KOH, W., NARAIN, R., FATAHALIAN, K., TREUILLE, A., AND O'BRIEN, J. F. 2013. Near-exhaustive precomputation of secondary cloth effects. *ACM Trans. Graph. 32*, 4 (July), 87:1–7.

KITTUR, A., CHI, E. H., AND SUH, B. 2008. Crowdsourcing user studies with Mechanical Turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, 453–456.

LEE, J., KLADWANG, W., LEE, M., CANTU, D., AZIZYAN, M., KIM, H., LIMPAECHER, A., YOON, S., TREUILLE, A., DAS, R., AND ETERNA PARTICIPANTS. 2014. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences*. 2014. (Preprint).

LIMPAECHER, A., FELTMAN, N., TREUILLE, A., AND COHEN, M. 2013. Real-time drawing assistance through crowdsourcing. *ACM Trans. Graph. 32*, 4 (July), 54:1–54:8.

MACKLIN, M., AND MÜLLER, M. 2013. Position based fluids. *ACM Trans. Graph. 32*, 4 (July), 104:1–104:12.

MCCANN, J., AND POLLARD, N. 2007. Responsive characters from motion fragments. *ACM Trans. Graph. 26*, 3 (July).

MICROSOFT, 2013. Drivatar website. http://research.microsoft.com/en-us/projects/drivatar.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1999. The PageRank citation ranking: bringing order to the Web. Technical Report 1999-66, Stanford InfoLab, November.

SCHÖDL, A., SZELISKI, R., SALESIN, D. H., AND ESSA, I. 2000. Video textures. In *Proceedings of SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, 489–498.

SMITH, A. M., LEWIS, C., HULLETT, K., SMITH, G., AND SULLIVAN, A. 2011. An inclusive taxonomy of player modeling. Tech. Rep. UCSC-SOE-11-13, University of California, Santa Cruz.

SOLENTHALER, B., AND PAJAROLA, R. 2009. Predictive-corrective incompressible SPH. *ACM Trans. Graph. 28*, 3 (July), 40:1–40:6.

STANTON, M., SHENG, Y., WICKE, M., PERAZZI, F., YUEN, A., NARASIMHAN, S., AND TREUILLE, A. 2013. Non-polynomial Galerkin projection on deforming meshes. *ACM Trans. Graph. 32*, 4 (July), 86:1–86:14.

ŠT'AVA, O., BENEŠ, B., BRISBIN, M., AND KŘIVÁNEK, J. 2008. Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, 201–210.

THUE, D., BULITKO, V., SPETCH, M., AND WASYLISHEN, E. 2007. Interactive storytelling: A player modelling approach. In *The Third Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE '07.

THUREY, N., MÜLLER-FISCHER, M., SCHIRM, S., AND GROSS, M. 2007. Real-time breaking waves for shallow water simulations. In *Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, PG '07, 39–46.

TREUILLE, A., LEWIS, A., AND POPOVIĆ, Z. 2006. Model reduction for real-time fluids. *ACM Trans. Graph. 25*, 3 (July), 826–834.

VON AHN, L., AND DABBISH, L. 2004. Labeling images with a computer game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, 319–326.

VON AHN, L., LIU, R., AND BLUM, M. 2006. Peekaboom: a game for locating objects in images. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, 55–64.

VON AHN, L., MAURER, B., MCMILLEN, C., ABRAHAM, D., AND BLUM, M. 2008. reCAPTCHA: Human-based character recognition via web security measures. *Science 321*, 5895 (August), 1465–1468.

WICKE, M., STANTON, M., AND TREUILLE, A. 2009. Modular bases for fluid dynamics. *ACM Trans. Graph. 28*, 3 (July), 39:1–39:8.

ZHU, Y., AND BRIDSON, R. 2005. Animating sand as a fluid. *ACM Trans. Graph. 24*, 3 (July), 965–972.

ZOOK, A., LEE-URBAN, S., DRINKWATER, M. R., AND RIEDL, M. O. 2012. Skill-based mission generation: A data-driven temporal player modeling approach. In *Proceedings of the 7th International Conference on the Foundations of Digital Games*, FDG '12.

ZOOK, A., FRUCHTER, E., AND RIEDL, M. O. 2014. Automatic playtesting for game parameter tuning via active learning. In *Proceedings of the 9th International Conference on the Foundations of Digital Games*, FDG '14.