Spacetime Constraints for Gameplay

Seth Cooper[1]

[1]Northeastern University, USA

Lightning Talk Abstract

This lightning talk will present the use of *spacetime constraints* [1] for modeling gameplay. Spacetime constraints are used in computer animation to specify what a character should do, and then solve a spatio-temporal constraint problem for a physically-plausible animation. In games, constraint-based methods have generally focused on tile-based levels and movement or discrete properties (e.g. [2, 3]). To support broader applications, we propose the use of spacetime constraint frameworks to model more continuous gameplay.

We have been exploring applications of spacetime constraints using a basic platformer game as a testbed. At a high level, the player state, inputs, and level layout are expressed as solver variables, and the game's update function as constraints over these variables. There are two main groupings of variables:

Timesteps — Represents all time-varying game state at a given timestep; one set of these is needed for each timestep used. This includes status (playing, won, lost); desired position, resolved position, and velocity; contact with ground, left wall, and right wall; and up, down, left, and right buttons pressed. For collision handling, two positions are used: desired position, where the player would end up based only on velocity, and resolved position, where they actually end up based on collisions. Contacts are used for display, jumping, and walljumping.

Axis-aligned bounding boxes — Used for static level layout; only one set is needed regardless of number of timesteps. These including blocks (shown in gray), goals (green), and hazards (orange). Boxes have variables for if they are active and their location and size.

Variables for all needed timesteps and boxes are allocated upfront. Step update constraints are added between adjacent timesteps. There are too many details to go into here, but they consist of constraints such as:

$$desiredPosX_{t+1} == resolvedPosX_t + velX_t;$$

$$resolveCollisionBlockLeftX_{t+1,b} \implies (resolvedPosX_{t+1} == blockLeftX_b - playerSize) \wedge (velX_{t+1} == 0);$$

Additional constraints can be added as desired. We use the Python Z3 solver [4], using the bit-blast and SAT heuristics with a fixed-point bit-vector representation. We explored these applications (examples in Figure 1):

Interactive Stepping — Using two timesteps and constraining box locations, the solver can be used to play interactively. The state and inputs are constrained in the first timestep, and the updated state is solved for in the second timestep.

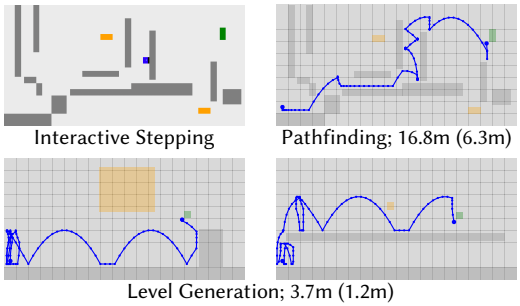

Figure 1: Selected images of different applications. Times are mean (and stdev) of 5 runs. Player is a blue square in top-left; blue lines show paths.

Pathfinding — Creating an array of timesteps and constraining box locations, the starting location, and reaching the goal by the last timestep (among other details), will solve for a path through the level. The solution includes per-timestep button presses.

Level Generation — Similar to above, but boxes are only constrained to be in the level and not overlap (if active), and the goal to be distant from the start. This will generate a level, along with the solution path (showing the level must ge solvable) and button presses.

The method does have drawbacks. Currently it is quite slow. Generated levels are simple, likely requiring more constraints to get interesting levels. It is necessary to specify sizes (i.e. maximum number of timesteps and boxes) upfront.

We are interested in exploring if there are any opportunities for interactive gameplay, content generation, or testing and debugging when the "engine" of a game is a spacetime constraint solver, and seeing if the apporach can extend to other gameplay elements, such as enemies, moving platforms, etc. We have also looked into "tank" controls for rotating, Asteroids-like player movement.

References

[1] A. Witkin, M. Kass, Spacetime constraints, SIGGRAPH 22 (1988) 159–168.
[2] A. M. Smith, M. J. Nelson, M. Mateas, LUDOCORE: A logical game engine for modeling videogames, in: Computational Intelligence and Games, 2010, pp. 91–98.
[3] M. J. Nelson, A. M. Smith, ASP with applications to mazes and levels, in: Procedural Content Generation in Games, 2016, pp. 143–157.
[4] L. de Moura, N. Bjørner, Z3: An efficient SMT solver, in: Tools and Algorithms for the Construction and Analysis of Systems, 2008, pp. 337–340.