

# cs224 Assignment 2: Plausible Animation

Out: Jan. 30th

Due: Wed, Feb. 5th, 11:59pm

## 1 Overview

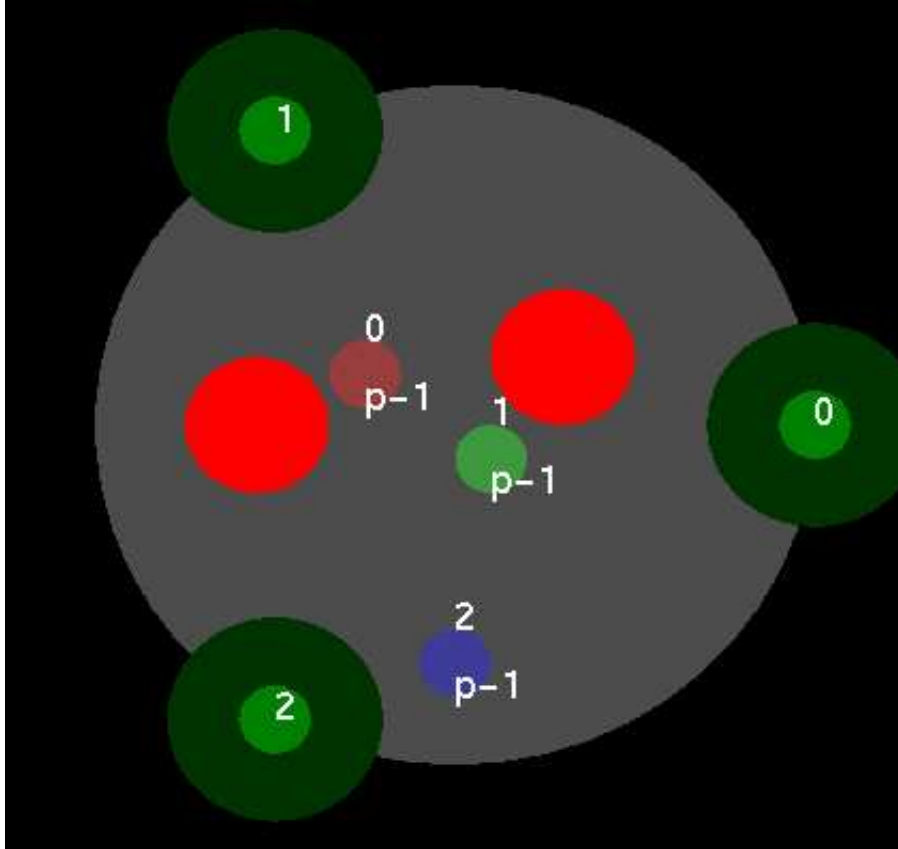
The first class lectures and the Matlab sampling assignment provided you with a basic grasp upon Metropolis sampling. As you will see, this technique has found application in many subfields within the larger umbrella of computer graphics. In this assignment we will specifically study its utility in generating physically plausible animations. Later on, in the Metropolis Light Transport assignment, we will see how it provides a computationally feasible solution to the dizzyingly complicated problem of global illumination.

In this assignment we will use a Markov chain Monte Carlo (MCMC) / Metropolis algorithm to sample multiple animations that satisfy constraints. We are heavily relying on Cheney and Forsyth’s “Sampling Plausible Solutions to Multi-body Constraint Problems” Siggraph’00 paper (available under [course/cs224/papers/plausible\\_sampling.pdf](#)). Being familiar with the paper is a prerequisite for this assignment, and this assignment handout.

## 2 Ramming Speed!



Consider a simple colliding-pool-ball simulator. We have a 2D world consisting of a round billiard table in which there are round “bumpers” placed at various locations. There are  $n$  “pockets” in the billiard table, centered on the outer circle of the table and distributed at even intervals (see figure for an example with two central bumpers and three pockets). There are several “pool balls” on the table, which move without friction.



The rule for collisions is easy: if a pool ball collides with a bumper, the edge of the table, or another ball, it bounces off with an angle of reflection equal to the angle of incidence (although you will modify this rule slightly to provide “plausible animations” through plausible variations). The physical simulation chooses final velocities such that energy and momentum is conserved for all bodies involved with the collision.<sup>1</sup> This simulation should look fairly realistic, as – modulo friction – it’s what you’d expect pool balls to do at the GCB or elsewhere: perfectly elastic collisions.

Given such a simulator and a fixed initial setup for all the pool balls (position and velocity), the rules above ensure that whenever you run the simulation you will obtain the same animation sequence. Rather boring after a while, eh? But, by altering slightly the “angle of reflection = angle of incidence” rule, for example, you may obtain slightly different variations of the animation. One can think of all the possible animation sequences thus generated as a collection of “samples”.

Now, here comes the big idea: can you smartly alter the animation sequence so that it satisfies some goal? For example, imagine an animation sequence where each specific ball ends up in a specific pocket, thus clearing the pool table. Cheney’s paper claims you can obtain several such animations via the MCMC algorithm.

### 3 Requirements

We are giving you a simulator and a viewer for the pool table simulation. You must do the following:

---

<sup>1</sup>In the case of bounces with the edge of the table or a bumper, the table or bumper is considered to have infinite mass. Masses throughout the simulation are stored as inverses, so we just say that the table and bumpers have an inverse mass of zero

- Choose some goal for the simulation. We have encouraged you to use the final pocket for each ball as a goal, but you should add more. Incorporate any new goals into the file format. (Total number of collisions globally, total number of collisions per-ball, et cetera)
- Devise some method  $f(\bar{x})$  to evaluate a given animation  $\bar{x}$ . The [somewhat subjective] composition of this highly non-linear and home-brewed function will have an enormous effect on the quality of your experimental results. You not only want to make sure that  $f(\bar{x})$  will be large when objectives are met, but that  $f(\bar{x})$  will also be somewhat large when objectives are “close” to being met. Otherwise McMC is unlikely to converge quickly. Remember that McMC should converge to the highest valued domain values of your function, regardless of its difficult/discontinuous nature; McMC is what this assignment is really about.
- Devise two mutation strategies and compute the tentative transition function  $T(\bar{y}|\bar{x})$  for each.
- Use McMC to try to sample your function  $f(\bar{x})$ .
- Set up two experiments where you use a distinct mutation strategy (or combinations of strategies) with one scene file. We should be able to generate 1000 samples in your Markov chain, and the animations should gradually improve (with some exceptions, of course). (See the documentation for `m_showAllConstraintSatisfyingAnimations`) With a well chosen scene file and mutation strategy, we should be able to see an evolution towards animations that both meet the goals you specified and that look physically plausible. We should be able to run these experiments (1 or 2) from the command line like so:
 

```
$ plausanim 1 <optional .input file>
$ plausanim 2 <optional .input file>
```

 If the optional `.input file` is not specified, you should use a default that you have tested and vouch for as an illustration of your McMC implementation.
  - Remember that McMC correctness counts. Don’t hack things up such that the theory goes out the window: pick mutation strategies and distributions that you will be able to implement with theoretical rigor.
- Provide a suite of test files that work with your [modified] parser. Some should be “easier” for the simulator than others – provide a range.
- Provide documentation about your ideas and thought processes. Specifically, describe the goals you can set with your implementation, and what we should do to see your implementation try to satisfy them. Also provide any other instructions that you think might be useful.
- Though it’s not specifically a *Requirement* (with a capital “R”), you are expected to modify the skeleton code you get to accomplish your goals. You will probably want to add or change code in the physical simulator and to the file parser, for instance.

## 4 Getting Started

1. If you haven’t read the Cheney paper, print it out and absorb it.
2. Grab the skeleton code from `/course/cs224/asgn/plausanim`
3. Read and understand this document.

4. Augment the skeleton code with an implementation of McMC suitable for the task at hand: a pool-table simulation using a discrete event simulator.

## 5 Support Code

We hope you will use the support code to hit the ground running and not get bogged down in the details of the physical simulator. You should have plenty of time to experiment with McMC. If anything is disastrously confusing, please make a post to the newsgroup and we'll try to illuminate the matter.

The support code does the following for you:

- Parses in a primitive file format that describes the layout of objects on a unit-circular pool table
- Provides a discrete event simulator.<sup>2</sup>
- Offers a visualizer to display and interact with a completed [or in-completed] animation.

Here are brief descriptions of the various classes you'll need to know about:

- **SimElt:** The basic superclass for all table elements is the SimElt. The board itself, the bumpers, the pockets, and the balls are all subclasses of the SimElt. Note that all SimElts extend the Decorable class/interface.
- **Animation:** The ordered vector of discrete events (Collisions, in this case) is maintained by the Animation. The Animation also owns the pointers to the SimElts on the board.
- **Collision:** Each event in the table simulation is modelled as a Collision. Each Collision encapsulates a timestamp, two unique SimElt identifiers, and expected velocities at the point of collision. At least one of these will always be a moving ball. Collision is a subclass of Decorable.
- **Decorable:** This interface allows subclasses to tag themselves using a simple string/double map. This can be useful when assigning probabilities to various events and objects.
- **Viewer:** The Viewer takes an Animation and lets the user interact with it graphically. Subclasses must implement the evolveAnimation() method, with is declared pure virtual. This is the top-level hook into your code.
- **Vector2d:** The Vector2d is a primitive class to store 2-vectors. There's nothing fancy, but it's useful. Feel free to add other methods and operators if you wish.
- **Everything Else:** The other files and classes are of less importance and visibility in this assignment. Mostly, they are support for the glut visualizer and console renderer.<sup>3</sup>

There are a few functions of particular prominence.

---

<sup>2</sup>A "discrete event simulator" accurately and efficiently simulates an event-based phenomena by jumping from one relevant moment to the next, and not by iterative (and costly) integration. In our system, all positions are linearly interpolated from collision to collision at render time. Your code will need to do work at the collisions, adding plausible variation to various quantities manipulated therein.

<sup>3</sup>Kudos to Pete Demoreuille, former cs224hta, for the always-useful glut console code

- `StudentViewer::evolveAnimation()`: This is called whenever the user presses 'e'. Your McMC code should probably be called underneath this method somewhere.
  - If `m_showAllConstraintSatisfyingAnimations` is set to true, you should stop whenever you hit an animation that meets the constraints you have set up.
  - If `m_showAllConstraintSatisfyingAnimations` is set to false, you should show every  $n$ th animation in your Markov chain, whether it satisfies your constraints or not. This should allow you [and us] to see how well your McMC algorithm evolves the animation.
- `StudentAnimation::areConstraintsSatisfied()`: This will return true or false if the animation in question does or does not satisfy the constraints specified in the file format. This will only behave properly if the ideal/final pocket decorations are set correctly for all balls in the animation. You must reset these if you re-simulate an animation.
- `StudentAnimation::copyIntoStudentAnimation()`: This will do a deep copy of a `StudentAnimation`. You will want to use this in your McMC implementation.
- `StudentAnimation::collide(Collision *)`: The collision physics are computed in `collide`. This method calls upon `playWithNormal()`, which you may want to alter.
- `StudentAnimation::playWithNormal(Collision *curcol, Vector2d &normal)`: This method allows control over the normals at each collision. This is probably the simplest and most effective way to alter the physics without violating the simulation's "plausibility".
- `SimElt::addPathSegment(double time, const Vector2d &pos, const Vector2d &vel, bool ontable=true)`: This method adds another linear path segment to the life of a `SimElt`. In practice, this is only called on balls when they bounce off other objects.
- `Viewer::key_down(int key, int x, int y)`: This is where you should add your own key-bindings.
- `console_printf_color(bool clear, float r, float g, float b, float a, const char *fmt, ...)`: This will allow you to display fading, scrolling text to the gl console. Set `clear` to false if/when you decide to use this.

## 6 The Viewer

The viewer keeps a single reference to an animation it draws. If you want to draw a new animation, you must copy your new animation into the viewer's animation.<sup>4</sup>

Key:	Binding:
s	Toggle the behavior of <code>evolveAnimation()</code> . See above.
e	Call <code>evolveAnimation()</code>
p	Toggle playback
1-9	Select the animation timestep when playing is disabled. 1 is the smallest timestep, 9 is the largest
0	Set the simulation time to the animation epoch ( $t=0$ )
→	Advance the animation display by one timestep
←	Decrement the animation display by one timestep

---

<sup>4</sup>Code to do this is included in the skeleton's "StudentViewer.cpp" file.

## 7 File Format

The file format is as follows:

- The first line contains the number of pockets on the board. They are distributed around the unit circle at even intervals, with the 0'th pocket sitting at  $\theta = 0$ . The board is always the unit circle.
- The second line specifies the number of bumpers,  $b$ .
- The next  $b$  lines describe the bumpers: `x y radius`
- The next line specifies the number of balls,  $n$ .
- The next  $n$  lines describe the balls: `startx starty startvelx startvely idealpocket`. The “idealpocket” is the pocket this ball must eventually end up in by the end of the animation. The pockets are 0-indexed. If “idealpocket=-1”, there is no constraint on the balls final resting place.

Here is an example file with 5 pockets, 4 bumpers, and 3 balls:

```
5
4
-.5 0 0.04
.5 0 0.06
0 .5 0.05
0 -.5 0.16
3
-0.5 0.5 0 0 0
0.5 0.5 0 2 -1
-0.5 -0.5 -1 1 4
```

You should augment the file format (and the `Animation::initialize()` method) if you want to, say, limit the number of collisions allowed for any given file. You are expected to change the support code to fit your needs, do not assume that you will get through the assignment without adding a few fields here or there.

## 8 Hints

- If something seems unclear, it probably is. Please post [non-revealing] questions to the news-group if or when they come up.
- Do not use the demo as a measure of what we expect. The owner of the support code spent a lot of energy building the simulator. Do better (MCMC-wise) than the demo – it often searches quite poorly in systems with relatively lenient constraints. This may change over the next week, we'll let you know if it does.
- Note that the demo does not support the `plausanim <i> <filename>` specification: it's a one-trick pony for the time being, but does operate on any file.

- You should not only display animations that satisfy the constraints. A good MCMC algorithm will not assign 0 probability to samples that are close-but-not-quite; as a result, you will have animations in your Markov chain that come close but do not quite accomplish your goal. Do show them. (Again, don't look to the demo on this one)

## 9 Handing In

To hand in your work, type

```
/course/cs224/bin/cs224_handin plausanim
```

from your working directory. Include a plausanim.txt file describing the goals you set for yourself and what you did to get them.