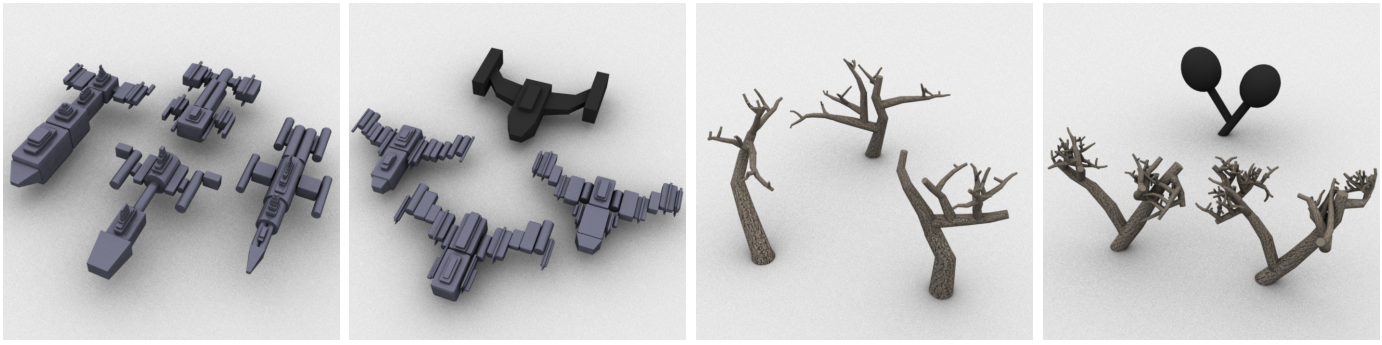
Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo

Daniel Ritchie* Stanford University
Ben Mildenhall* Stanford University
Noah D. Goodman* Stanford University
Pat Hanrahan* Stanford University



Forward Sampling | SOSMC-Controlled Sampling | Forward Sampling | SOSMC-Controlled Sampling

Figure 1: *Controlling the output of highly-variable procedural modeling programs using our Stochastically-Ordered Sequential Monte Carlo algorithm. Here, the controls encourage volumetric similarity to a target shape (shown in black).*

Abstract

We present a method for controlling the output of procedural modeling programs using Sequential Monte Carlo (SMC). Previous probabilistic methods for controlling procedural models use Markov Chain Monte Carlo (MCMC), which receives control feedback only for completely-generated models. In contrast, SMC receives feedback incrementally on incomplete models, allowing it to reallocate computational resources and converge quickly. To handle the many possible sequentializations of a structured, recursive procedural modeling program, we develop and prove the correctness of a new SMC variant, Stochastically-Ordered Sequential Monte Carlo (SOSMC). We implement SOSMC for general-purpose programs using a new programming primitive: the stochastic future. Finally, we show that SOSMC reliably generates high-quality outputs for a variety of programs and control scoring functions. For small computational budgets, SOSMC's outputs often score nearly twice as high as those of MCMC or normal SMC.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems;

Keywords: Procedural Modeling, Directable Randomness, Probabilistic Programming, Sequential Monte Carlo

*e-mail: {dritchie, bmild, ngoodman, hanrahan}@stanford.edu

1 Introduction

Procedural modeling has long been used in computer graphics to generate varied, detailed content with minimal human effort. Procedural models for trees, buildings, cities, and decorative patterns enrich the virtual worlds of movies and games [Měch and Prusinkiewicz 1996; Müller et al. 2006; Wong et al. 1998]. Ambitious new projects aim to produce fully-procedural, galactic-scale environments for players to explore [Procedural Reality 2014; Hello Games 2014].

This expressive power comes at a cost: procedural models often use complex, recursive control logic, resulting in emergent behavior which is difficult to direct. As a result, technical artists often must tweak parameters and massage initial conditions to achieve a desired look. This time and effort may defeat the purpose of using procedural modeling in the first place.

Fortunately, recent years have seen advances in the use of probabilistic inference techniques to control procedural models [Talton et al. 2011; Stava et al. 2014; Yeh et al. 2012]. Viewing a procedural model as sampling from a probability distribution allows for the application of Bayesian inference techniques: the prior is the procedural model itself, and the likelihood is some high-level control expessed as a scoring function.

This previous work relies on Markov Chain Monte Carlo (MCMC), but other Bayesian posterior sampling algorithms are available: another popular choice is Sequential Monte Carlo (SMC). SMC uses a set of samples, or *particles*, to represent a distribution that changes over time as new evidence is observed. As the distribution changes, SMC shifts more particles (and thus more of its computational budget) to higher-probability regions of the state space. For probabilistic models that fit this pattern of 'evidence arriving over time,' such as modeling the location of a mobile robot, SMC is often the method of choice: the incremental evidence it receives provides feedback early and often, allowing it to converge quickly [Doucet et al. 2001]. In contrast, MCMC receives feedback only after running through the entire model.

Can we use Sequential Monte Carlo to control procedural models? Procedural models are typically hierarchical and recursive—we need to cast them instead as sequential processes, where control can be imposed incrementally over time. Our insight is that representing procedural models with *probabilistic programs* makes this possible. A probabilistic program makes random choices, and doing inference amounts to reasoning about the space of possible executions under some constraint [Goodman et al. 2008]. For procedural models, random choices are decisions about the structure and shape of the generated model, and the constraint is the control scoring function. Critically, these programs have sequential semantics: they execute in a series of discrete time steps. Control can be imposed incrementally by evaluating a scoring function on the incomplete model at each step, providing an estimate as to how well the algorithm is doing thus far.

However, there are multiple ways to sequentialize a structured procedural modeling program—and as we will show, SMC does not always perform well using the depth-first ordering given by most modern, stack-based programming languages. It is typically not clear *a priori* what the best ordering(s) will be for a given program and control scoring function: in the absence of any special knowledge, a good strategy might be to execute the program in *random* order.

Following this insight, we introduce a new variant of SMC, *Stochastically-Ordered Sequential Monte Carlo (SOSMC)*, in which each particle executes the program in an independent, random order. We also prove that this algorithm is a correct, asymptotically-unbiased sampler for the posterior distribution defined by the constrained program. To implement SOSMC for procedural models expressed as general-purpose probabilistic programs, we also introduce a new programming primitive, the *stochastic future*, whose use requires minimal modification to the original program. We then show that SOSMC can handle a range of procedural models and controls explored in the literature, and that it generates better-scoring samples under tight time budgets than either normal SMC or Metropolis-Hastings (MH). For small computational budgets, SOSMC's outputs often score nearly twice as high as those of normal SMC or MH.

In summary, our main contributions are:

1. The SOSMC algorithm and a proof of its correctness.
2. An implementation of SOSMC in a practical, general-purpose programming language using stochastic futures.
3. An evaluation of SOSMC's expressiveness and performance.

We give a high-level overview of our main insights and approach in Section 3, then we formally describe the SOSMC algorithm in Section 4 and our prototype implementation in Section 5. In Section 6, we evaluate the algorithm's performance on a variety of procedural models with constraints and compare to other sampling methods.

2 Related Work

Controlled Procedural Modeling Multiple existing projects aim to control procedural models through probabilistic inference. One uses reversible-jump MCMC to direct the output of stochastic context free grammars [Talton et al. 2011]. Another uses similar MCMC techniques to guide L-system-based trees toward a target polygonal model [Stava et al. 2014]. Others develop new trans-dimensional MCMC methods to solve complex layout problems [Yeh et al. 2012], use gradient-based MCMC to guide random structures toward physical stability [Ritchie et al. 2015], or use MCMC to make parameterized models of urban environments satisfy desired critera [Vanegas et al. 2012]. These all use MCMC as their core control algorithm; in contrast, we focus on Sequential Monte Carlo for its potential performance benefits.

There have also been several non-probabilistic approaches to directing procedural models. Environmentally-sensitive L-systems and Open L-systems allow communication between a procedural model and its environment [Prusinkiewicz et al. 1994; Měch and Prusinkiewicz 1996]. Another approach decomposes the modeling domain into geometric guides to which the procedural model should adhere [Beneš et al. 2011]. These approaches affect the model as it evolves. Our approach can be thought of as generalizing this type of control to the probabilistic inference setting.

Sequential Monte Carlo Sequential Monte Carlo has a long history, beginning with the simulation of self-avoiding polymer chains [Hammersley and Morton 1954; Rosenbluth and Rosenbluth 1955]. A critical point came with the introduction of a resampling step, allowing the reallocation of particles according to their probability [Gordon et al. 1993; Stewart and McCarty 1992]. The resulting algorithm, called the *bootstrap filter*, was designed for linear time-series processes. We extend it for structured processes by linearizing the process and treating the linearization order as additional set of random variables. It can be shown that, as the number of SMC particles approaches infinity, their distribution approaches the target posterior density [Smith and Gelfand 1992; Gordon et al. 1993]. We prove that this distribution is invariant under linearization order, thus re-ordering does not change program semantics.

SMC in Computer Graphics Sequential Monte Carlo has found applications in computer graphics already. It has been applied to Monte Carlo integration for physically-based rendering, in particular rendering with participating media [Fan 2006; Pegoraro et al. 2008]. It has also been used to control virtual characters responding to dynamic environments [Hmlinen et al. 2014]. These applications have straightforward sequential interpretations: propagation of light along a path through space, or the motion of a character over time. In contrast, we focus on structured procedural models, which have many possible sequentializations.

SMC belongs to the family of *population-based methods*, which evolve a population of samples toward some desired goal. This general approach has also been used for 3D shape design [Xu et al. 2012]. This system maintains a complete set of shapes at all times, whereas ours works with incomplete shapes defined by partial program executions.

SMC for Probabilistic Programs Sequential Monte Carlo has also previously been applied to probabilistic programs. The Anglican language implements several SMC methods, including sophisticated SMC/MCMC hybrids [Wood et al. 2014]. Probabilistic C uses OS multiprocessing primitives to construct efficient, parallel implementations of these same algorithms [Paige and Wood 2014]. It is also possible to implement these algorithms using a continuation-passing-style compiler [Goodman and Stuhlmüller 2014]. These systems are restricted to handling a fixed number of time steps—the common case in statistical inference, where each step corresponds to a data point. In contrast, we are concerned with scenarios that have a variable number of steps, as this situation arises often with structured, recursive procedural models.

3 Approach

In this paper, we focus on programs that generate models through hierarchical, recursive accumulation of geometric primitives into an implicit global state. To illustrate our approach, we use an example

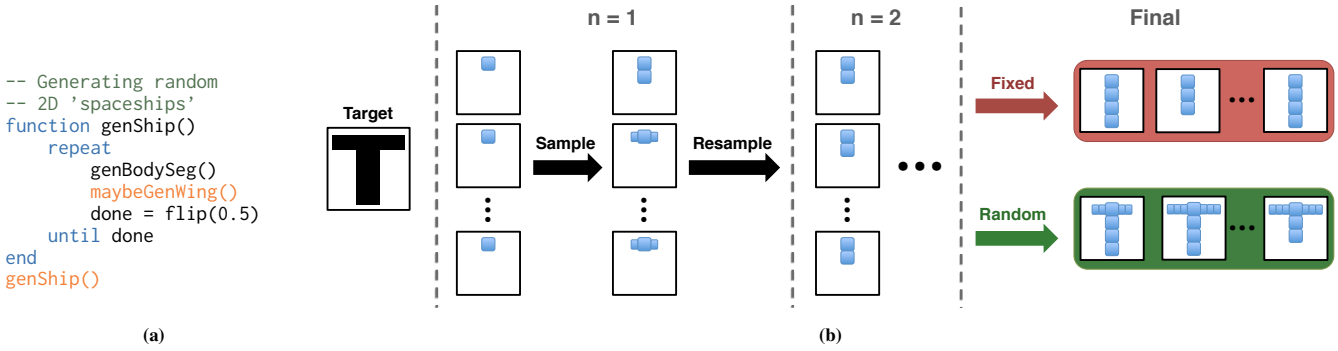


Figure 2: (a) *A program that generates simple random spaceships. Orange-highlighted function calls can be executed in any order with respect to one another.* (b) *SMC resampling favors higher-scoring states, so particles that fill in the body first will dominate. Under fixed ordering, particles skip wing generation altogether, whereas random ordering can defer wing generation until after body generation.*

Algorithm 1 SMC for procedural modeling programs

procedure SMC(*program*, *scorefn*, N)
  $P \leftarrow N$ new particles (instances of *program*)
  $W \leftarrow N$ real-valued weights
  while some particle $p \in P$ has not terminated **do**
    // Sample
    for all unterminated particles $p \in P$ **do**
      Run p until it generates a new geometric primitive
    // Score
    for $i = 1$ to N **do**
      $W(i) \leftarrow$ *scorefn*$(P(i))$
    NORMALIZE(W)
    // Resample
    $P \leftarrow$ WEIGHTEDSAMPLEN(P, W, N)

program that generates random simplified spaceships out of blocks (Figure 2a). The program generates the ship body by placing a random number of contiguous blocks and may randomly grow wings, also made of a random number of blocks, from any body segment. For brevity, we do not show pseudocode for the wing-creation function `maybeGenWing`—its structure is similar to that of `genShip`. We will use SMC to sample from this program under a scoring function that encourages similarity to a target shape.

SMC runs N copies of the program, called *particles*, (conceptually) in parallel. Particles execute until they arrive at a barrier synchronization point—this is the *sampling* phase. In our procedural modeling programs, barriers occur when programs generate a new geometric primitive. SMC computes the score of each particle and then randomly samples N particles in proportion to their scores: high-scoring particles are sampled more often, and low-scoring ones are sampled less often, or not at all. This is the *resampling* phase, and these new particles become the input for the next sampling phase. Resampling ensures that the algorithm concentrates particles (and thus its computational budget) in high-scoring regions of the state space. Essentially, SMC operates like a stochastic version of beam search [Bisiani 1987]. Algorithm 1 shows high-level pseudocode for running SMC on procedural modeling programs.

The first column of Figure 2b shows a hypothetical set of particles that have passed the first barrier—that is, they have placed one primitive, which in this case must be a body segment. At the next barrier (second column), some particles will randomly start growing wings from the first body segment, while others will instead proceed with the next body segment. Because body segments are larger than wing segments, placing a body segment brings the model closer to the target more quickly than placing a wing segment. Thus, the resampling phase will favor particles that place body segments over those that place wing segments—body-segment particles will dominate the next round (third column).

Consider the calls to `genShip` and `maybeGenWing`, highlighted in orange in Figure 2a. These calls generate independent components of the model and could in principle interleave their execution in any order with respect to one another. However, most programming languages will execute them in a fixed, depth-first order, which causes a problem in this example: all of the second-round particles decided not to generate wings on the first body segment, and SMC has no mechanism to reverse that decision. The best possible result from this point on are ships with bodies that match the target, but no wings (Figure 2b, red box). We could try to eliminate this problem by only resampling after body segment generation, but this would leave wing generation without any guidance from resampling, requiring it to match the target by pure chance. And even if this fix worked, it would be specific to this program—we seek general-purpose solutions that work for any procedural model.

Now suppose each particle executes the calls to `genShip` and `maybeGenWing` in a random order. This means that some second-round particles likely deferred execution of `maybeGenWing` and instead continued executing `genShip`—they have yet to decide if they will generate wings on the first body segment. By deferring this decision, SMC can generate results that have both body and wings that match the target (Figure 2b, green box). Sequential Monte Carlo is a form of importance sampling, and here execution order randomization helps it sample the most important objects first.

These execution-order-sensitive situations do not occur in the linear time-series models for which SMC was developed, but they frequently arise in structured procedural modeling. It is clear that some orderings are better than others, but it is *not* always clear which orderings those are. Even if known, it is cumbersome to explicitly express specific orderings in the program text. And finally, the best orderings depend upon the score function being used—it is unreasonable to expect the programmer to restructure her code for each new control imposed on a model.

This is the motivation behind Stochastically-Ordered Sequential Monte Carlo: since we cannnot know what execution orderings are good *a priori*, we randomize them, in the hope that randomization will discover good orderings on average. After formally describing SOSMC and its implementation in Sections 4 and 5, we show that randomization does lead to reliably better results, both qualitatively and quantitatively (Section 6).

4 SOSMC

Having outlined our approach intuitively, we now formally define the probability distribution sampled by the SOSMC algorithm. SMC algorithms are specified as sampling from a sequence of distributions p_1, p_2, ...; the final distribution p_N is often the one of interest. These distributions are usually defined over a growing set of variables x, e.g. $p_1(x_1)$, $p_2(x_1, x_2)$, and so on. These variables typically represent states which evolve over time.

Defining such a sequence of distributions for SOSMC is more complicated, as general procedural modeling programs follow a structured execution that does not conform to a single, linear time-series interpretation. Thus, we augment our state space of variables x to include execution ordering choice variables o in addition to the program's own random choice variables r.

We define the intermediate distribution p_n to be the distribution over all execution traces which generate n or fewer geometric primitives. Let $\mathbf{x}_n$ be the sequence of all random choices made up to primitive n, where $\mathbf{x}_0$ is empty. As subsets of this sequence, let $\mathbf{r}_n$ denote the procedural model's random choices, and let $\mathbf{o}_n$ denote the ordering choices. We will sometimes refer to $\mathbf{r}$ as a *trace* through the procedural model program. The intermediate distribution p_n can then be defined recursively as

$$\begin{aligned} p_n(\mathbf{x}_n) &= p_{n-1}(\mathbf{x}_{n-1}) \cdot p(\mathbf{x}_n|\mathbf{x}_{n-1}) \\ &= p_{n-1}(\mathbf{x}_{n-1}) \cdot \prod_{i=1}^{|\mathbf{x}_n \setminus \mathbf{x}_{n-1}|} p(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) \end{aligned}$$

where $x_{n,j:k}$ are the jth to kth random variables generated up to primitive n. The form of the per-variable conditional probability $p(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1})$ depends on the type of the variable $x_{n,i}$. If it is one of the procedural model's random choices, then the conditional probability is a function of the variable's parents in the program's dataflow graph and depends on the primitive distribution from which the variable is drawn (e.g. uniform, Gaussian):

$$p_r(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = p(x_{n,i}|\text{par}(x_{n,i}))$$

If $x_{n,i}$ is an ordering choice, then the conditional probability is defined by an *ordering policy* π. This policy determines how to select the next subcomputation to continue when the currently-executing subcomputation finishes, or when a particle synchronization barrier is reached (i.e. when a geometric primitive is generated).

We are concerned with two ordering policies. The first is the *deterministic* policy:

$$\pi_D(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = \begin{cases} 1 & \text{if } x_{n,i} = N(x_{n,1:(i-1)}, \mathbf{x}_{n-1}) \\ 0 & \text{otherwise} \end{cases}$$

where $N(x_{n,1:(i-1)}, \mathbf{x}_{n-1})$ is the number of subcomputations that could be continued at this point. This policy chooses the last option, equivalent to popping the top of a stack, with 100% probability. This behavior corresponds to running a program with depth-first execution ordering—normal SMC, in other words.

The second ordering policy of interest is the *stochastic* policy:

$$\pi_S(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = \frac{1}{N(x_{n,1:(i-1)}, \mathbf{x}_{n-1})}$$

which uniformly at random chooses a subcomputation to continue. This behavior corresponds to running a program with randomized execution order—the full SOSMC algorithm.

Thus far, we have only defined the *prior* distribution specified by the program itself. We already know how to sample from this distribution: run the program forward. Sampling only becomes challenging when we include a *likelihood* term that shapes the distribution. Our likelihood term is given by a user-provided score function $s(\cdot)$. Critically, this score function must be defined for partial execution traces $\mathbf{r}_n$, not just complete execution traces. The total, unnormalized posterior density at step n is

$$F_n(\mathbf{x}_n) = s(\mathbf{r}_n) \cdot p_n(\mathbf{x}_n)$$

The full, normalized probability distribution from which SOSMC samples at step n is

$$P_n^\pi(\mathbf{x}_n) = \frac{F_n(\mathbf{x}_n)}{Z_n^\pi} \tag{1}$$

where Z_n^π is the partition function which normalizes the distribution and depends on the ordering policy π. The final distribution in this sequence is P_N^π: the distribution over complete runs of the program. As the number of SMC particles approaches infinity, their distribution approaches the target posterior density [Smith and Gelfand 1992; Gordon et al. 1993]. Thus, SOSMC is an asymptotically-unbiased sampler for P_N^π.

In the Appendix, we show that the marginal distribution on generated models is the same under the stochastic ordering policy π_S as under the deterministic policy π_D. In other words, if we consider only the final state and not the order in which it was generated, then SOSMC draws samples from the desired distribution. The proof proceeds by marginalizing out the ordering choices $\mathbf{o}_N$ and showing that the two policies generate equivalent sets of complete traces $\mathbf{r}_N$. Finally, the N subscript indicates that our programs always terminate after a finite number of steps. The proof in the Appendix operates in this setting, but it also discusses programs that almost always terminate (i.e. terminate with probability one).

5 Implementation Using Stochastic Futures

To implement execution order randomization, we need a mechanism for interleaving the execution of different function calls with respect to one another. This requirement suggests looking at concurrent programming primitives. We settled on *futures*, a lightweight concurrency primitive that operates at the function call level [Halstead 1985]. Futures were originally designed for fine-grained parallelism, but we use them for a different interpretation of concurrency: sequential, interleaved programming. When called, a future may or may not begin executing, but it must finish executing when the program requests its return value. One common programming interface for futures allows for their creation by wrapping a function call with `future.create` and requesting their values by calling a `force` function. The interface to *stochastic* futures includes two more features:

- `future.switch()`: Switch control to and resume executing some other (random) active future. Our SOSMC implementation calls this function after every resampling step, allowing resampled particles to take different paths through the program as they advance.
- `future.finishall()`: Finish all active futures. Our programs generate geometry by appending to an implicit global model state, so most futures do not have a return value (e.g. the highlighted lines in Figure 2a). Our SOSMC implementation calls this function at the end of every program to force all such futures to finish.

To achieve the best performance with SOSMC, a procedural modeling program should use a stochastic future wherever it makes a

Algorithm 2 Implementing stochastic futures with coroutines

```
q ← { }                                   // A global queue of active futures
curr ← nil                                // The currently-running future
procedure SWITCH()
    COYIELD()
procedure FORCE(future)
    // Suspend the forcing future until this future is finished
    q ← q \ {curr}
    future.waiters ← future.waiters ∪ {curr}
    return COYIELD()
procedure FINISHALL()
    // Randomly continue futures until all are finished
    while ¬ EMPTY(q) do
        f ← UNIFORMDRAW(q)
        CONTINUE(f)
procedure CONTINUE(future)
    curr ← future
    retvals ← CORESUME(future.co, future.args)
    future.args ← { }
    if COFINISHED(future.co) then
        // Reactivate any suspended futures that were waiting
        // for this one to finish
        for all w ∈ future.waiters do
            w.args ← retvals
            q ← q ∪ {w}
        q ← q \ {future}
```

branching decision predicated on a random choice. In our implementation, we insert these futures manually, since these situations are easy to identify in practice and typically occur near natural function call boundaries (e.g. `maybeGenWing` in Figure 2a). It should also be possible to automatically transform programs into this form using source-to-source compilation guided by simple static analysis.

Note that since function calls may execute in an arbitrary order, the program must be thread safe: any accesses to shared data can be reordered without changing program behavior. In our implementation, the only shared data structure is the global model state, and adding geometry to this state is an associative operation.

Implementing stochastic futures requires the ability to arbitrarily switch between different in-progress computations. Higher-level concurrency primitives are often implemented atop lower-level ones, such as threads. For stochastic futures, *coroutines* are a natural choice of implementation primitive. Coroutines are a generalization of subroutines that can suspend their execution, yield control to another coroutine, and then resume later. They were designed for sequential concurrency in the form of cooperative multitasking [de Moura and Ierusalimschy 2004]. Algorithm 2 outlines an impementation of `switch`, `force`, and `finishall` in terms of asymmetric coroutines. Calling `finishall` initiates a loop that drives the random execution of futures, while `switch` and `force` determine when control returns to this loop.

We implement a prototype of SOSMC in Lua, with performance-critical components such as mesh voxelization and intersection implemented as high-performance extensions in Terra [DeVito et al. 2013]. Following Algorithm 2, we implement stochastic futures using Lua's native coroutines. To perform weighted particle resampling, we use the well-known *systematic* resampling scheme for its simplicity and practical variance reduction properties; we also found *residual resampling* to work well [Douc and Cappe 2005]. The source code for our implementation can be found here: `https://github.com/dritchie/procmod`. For the comparisons to Metropolis Hastings in Section 6, we also implement MH for probabilistic programs in Lua [Wingate et al. 2011]. When the program's dataflow graph is tree-structured, this algorithm is nearly identical to the MH algorithm for context-free grammars described in previous work [Talton et al. 2011].

Both our prototype SMC and MH implementations must, on each iteration, replay the program trace from its beginning, though generated geometry and derived quantities are cached and not recomputed. This gives them a quadratic time complexity in the depth of the program and is a known drawback of 'lightweight' probabilistic programming implementation techniques [Wingate et al. 2011]. The asymptotic similarity of the two implementations makes it fair to compare their relative running times, which we do in Section 6. In practice, SMC actually suffers *more* from trace replay overhead: in our experiments, up to 80% of system runtime for deep, complex models, compared to 10% for MH. As we will show, SOSMC regularly outperforms MH despite this handicap and stands to improve significantly with a more efficient implementation.

SMC for probabilistic programs can in fact be implemented without this overhead. Rather than replaying traces, particles could suspend and then resume at each sample/resample step. Resampling then requires copying suspended particles, which can be implemented efficiently with a construct like POSIX fork [Paige and Wood 2014]. In a purely functional language, suspended particles can also be represented with continuations [Goodman and Stuhlmüller 2014].

Finally, while our prototype implementation is serial, SMC is very straightforward to parallelize, which further enhances its performance potential. Particles can be evolved independently in parallel in the sampling phase and then gathered for the resampling phase using barrier synchronization.

6 Evaluation

We now demonstrate the ability of SOSMC to quickly and reliably generate high-quality procedural modeling samples. As test cases, we have chosen a variety of programs and controls that span a range of useful features, many of which have been explored previously in the literature [Talton et al. 2011]. We show that SOSMC can draw useful samples from these programs and controls, and that it generates higher-scoring samples than SMC or MH given small computational budgets. In all examples, we impose an additional score function which prevents geometry self-intersections by assigning a zero score to such configurations.

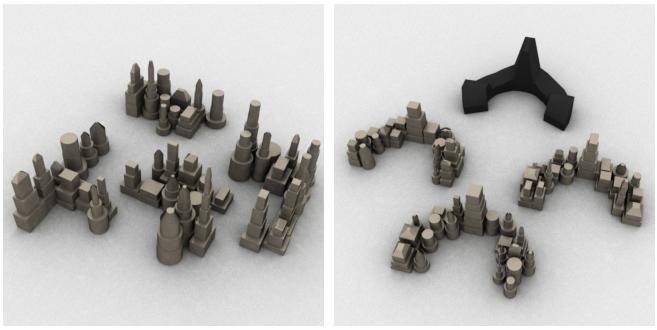
6.1 Volume Matching

It can be useful to control the overall 3D shape of a model via a rough geometric proxy. We implement this control volumetrically. If $V_{\textbf{target}}$ is a target binary voxel grid defined over domain $\mathcal{D}$, and $V_{\mathbf{r}}$ is the voxelization of the model described by execution trace $\mathbf{r}$ onto $\mathcal{D}$, then the *volume matching* score function $s_{\textbf{vmatch}}$ is

$$s_{\textbf{vmatch}}(\mathbf{r}) = \mathcal{N}(\text{sim}(V_{\mathbf{r}}, V_{\textbf{target}}), 1, \sigma) \cdot \mathcal{N}(\varepsilon_{\textbf{out}}(\mathbf{r}), 0, \sigma)$$

$$\text{sim}(V_1, V_2) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \mathbb{1}\{V_1(x) = V_2(x)\}$$

where $\text{sim}(V_1, V_2)$ returns a [0,1] similarity score for two voxel grids. $\varepsilon_{\textbf{out}}(\mathbf{r})$ returns the maximum amount to which the model defined by $\mathbf{r}$ extends outside $\mathcal{D}$ along any dimension. The first normal term in $s_{\textbf{vmatch}}(\mathbf{r})$ encourages similarity to the target volume. The second term penalizes growing beyond $\mathcal{D}$, where the target volume is not defined. We use a 2% error tolerance in all of our experiments ($\sigma = 0.02$) unless otherwise specified.



Forward Sampling SOSMC-Controlled Sampling

Figure 3: *SOSMC sampling from a random building complex model with volume matching applied.*

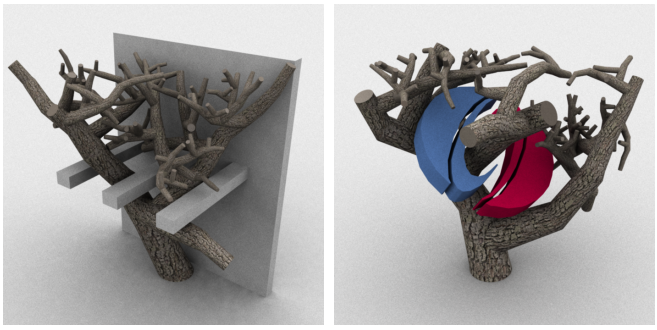


Figure 4: *Using the object avoidance scoring function to make gnarly trees grow around obstacles.*

Figure 1 shows some examples of spaceships and trees sampled according to this score function using SOSMC. Figure 3 applies the same score function to encourage a building complex to take on a crescent-like shape.

6.2 Object Avoidance

Volume matching allows an artist to specify what regions of space a model should occupy; it can also be valuable to specify the space a model should *not* occupy. For this control, the user provides a set of objects with which the model should avoid contact. We rasterize these objects onto a binary voxel grid $V_{\textbf{avoid}}$. The *object avoidance* score function $s_{\textbf{avoid}}$ is then

$$s_{\textbf{avoid}}(\mathbf{r}) = \prod_{x \in \mathcal{D}} \mathbb{1}\{V_{\mathbf{r}}(x) \uparrow V_{\textbf{avoid}}(x)\}$$

where $\uparrow$ is logical NAND. This function imposes a hard constraint: it returns 0 if $V_{\mathbf{r}}$ and $V_{\textbf{avoid}}$ have any mutually filled cells and 1 otherwise.

Figure 4 shows two examples of using object avoidance to generate trees that avoid obstacles. On the left, the tree avoids a wall with three protruding ledges; on the right, it grows through and around the SIGGRAPH logo. These examples also use a weaker version of the volume matching score function ($\sigma = 0.05$) to encourage the trees to grow to a tall, full shape.

6.3 Image Matching

It is also useful to specify projective properties of a model, such as how it looks from a particular viewpoint or when lit from a particular angle. We implement this type of control through image-based comparisons. If $I_{\textbf{target}}$ is a target binary image defined over domain $\mathcal{D}$, and $I_{\mathbf{r}}$ is a rendering of the model described by trace $\mathbf{r}$ onto $\mathcal{D}$, then the *image matching* score function $s_{\textbf{imatch}}$ is

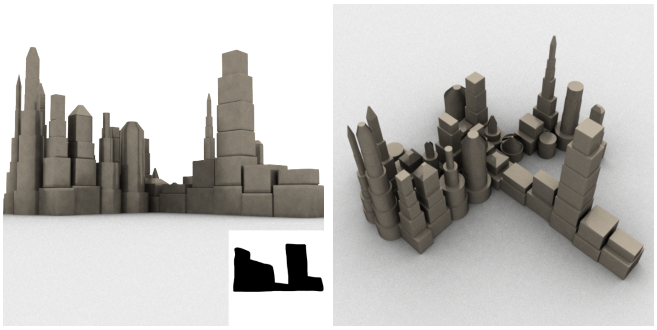
$$s_{\textbf{imatch}}(\mathbf{r}) = \mathcal{N}(\text{sim}(I_{\mathbf{r}}, I_{\textbf{target}}), 1, \sigma)$$
$$\text{sim}(I_1, I_2) = \frac{\sum_{x \in \mathcal{D}} W(x) \cdot \mathbb{1}\{I_1(x) = I_2(x)\}}{\sum_{x \in \mathcal{D}} W(x)}$$

where W is a 'weight image' defined over $\mathcal{D}$. The weight image allows users to draw strokes over parts of the image domain where strict matching is more or less important. For the results shown in this paper, W is uniform unless explicitly shown. As with volume matching, σ is 0.02 unless otherwise specified.

Figure 5 shows a use of the image matching scoring function to enforce a target silhouette for a building complex when viewed from a particular angle. Note that the generated model is still free to exhibit random structure when viewed from other angles.

In Figure 6, we use image matching to control the profile of a spaceship. The generated models bear strong similarity to the target image when viewed from the front but are otherwise unconstrained, revealing diverse structure when viewed from other angles.

Figure 7 shows another use of image matching: controlling the shadows cast by toy blocks strewn about a floor. Here, we decrease the score error tolerance by an order of magnitude ($\sigma = 0.002$), use



Front View Top View

Figure 5: *The image matching scoring function is used to control the appearance of a building complex from a particular viewpoint.* (Left): *The model as viewed from the target viewpoint.* (Right): *The model viewed from above.*

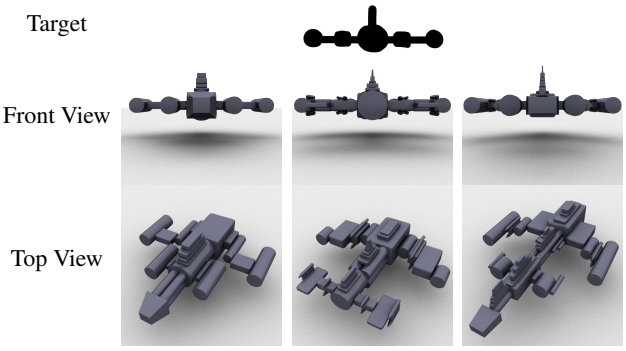
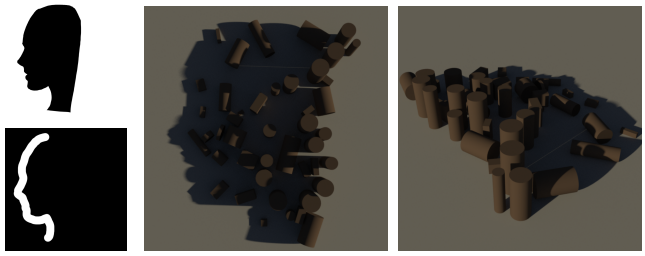


Figure 6: *Using image matching to control the appearance of a spaceship's front profile. The SOSMC-sampled results closely match the target when viewed head on but exhibit a variety of structures when viewed from other angles.*



Target/Weight Top View Oblique View

Figure 7: *Using the image matching scoring function to control the shape of cast shadows in a scene with toy blocks scattered on a floor. Face silhouette image derived from a template created by Milliande Printables (*`http://www.milliande-printables.com/face-silhouette-woman-stencil.html`*).*

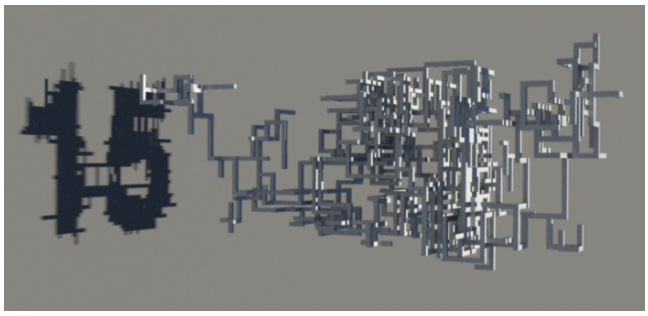


Figure 8: *Using image matching to control the shadows cast by a network of pipes.*

a weight image that places 10 times more weight on the outline of the target face image, and increase the particle count ($N = 1500$). These changes help SOSMC to match the fine details along the shadow silhouette edge. Again, the blocks in this example appear randomly arranged when viewed from other angles.

In Figure 8, we use image matching to shape the shadow cast by a network of rectangular pipes. We lower the score error tolerance to $\sigma = 0.0005$ to encourage SOSMC to fill in the shadow as completely as possible while avoiding extrusions beyond the desired silhouette. For our implementation, this is more practical than increasing particle count due to the model's extreme depth.

6.4 Quantitative Evaluation

Table 1 shows timing statistics for the examples presented in this section. The second-to-last column shows the number of particles used by SOSMC; we find that 300 particles is sufficient to generate high-quality results in most cases. The last column reports the time taken to generate the example; for figures that show multiple output models, the time reported is the average time to generate them. All timing data was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.8.5. Times for simpler models, such as the spaceship, are already fast enough (a few seconds) to be used in interactive settings. As noted in Section 5, generation times for more complex models can be significantly reduced by eliminating trace replay overhead or through parallelization.

We can also compare how well SOSMC, SMC, and MH generate high-scoring samples under different computational budgets. We are particularly interested in their behavior in low-budget scenarios. As test cases, we use the spaceship, building complex, and tree

Program	*Control*	*Figure*	N	*Time* (s)
Spaceship	$s_{\textbf{vmatch}}$	1	300	3.09
Gnarly Trees	$s_{\textbf{vmatch}}$	1	300	598.34
Building Complex	$s_{\textbf{vmatch}}$	3	300	24.14
Gnarly Trees	$s_{\textbf{avoid}} \cdot s_{\textbf{vmatch}}$	4	100	164.25
Building Complex	$s_{\textbf{imatch}}$	5	300	38.44
Spaceship	$s_{\textbf{imatch}}$	6	300	7.33
Toy Blocks	$s_{\textbf{imatch}}$	7	1500	135.91
Pipes	$s_{\textbf{imatch}}$	8	100	675.81

Table 1: *Timing data for all procedural modeling examples shown in this paper. N is the number of particles used by SOSMC.*

programs under volume constraints. These programs all exhibit recursive structure, but of a different nature: the spaceship program spawns recursive paths (wings, etc.) from a single recursive spine (the body), whereas the building complex and tree programs generate components in a multiply-branching, tree-recursive style.

We find that MH requires additional tuning to achieve peak performance. Specifically, it generates better results with a score function tempered down to a 0.5% error tolerance ($\sigma = 0.005$). We also experimented with parallel tempering but found it not to perform better than normal MH when run for the same amount of time on sequential hardware. If run on parallel hardware—as in e.g. Talton et al. [2011]—it could perform better, but for fair comparison, we would also have to run SMC in parallel. SMC could then process more particles in the same amount of time, improving its performance as well.

In our comparison experiment, we run SMC and SOSMC for particle counts ranging from 10 to 1000. At each particle count, we also run MH, giving it as much time as an average SOSMC run takes to complete at that particle count. We run each algorithm 10 times, take the highest score for each run, and record the mean and variance of those high scores. Figure 9 shows the results of this experiment. On the left, we plot mean highest score against increasing computational budget; line thickness is proportional to variance in highest score. Our implementation computes all quantities in log-probability space, so the scores shown are log scores.

For the spaceship example, SOSMC starts with higher scores than either SMC or MH, which both require a significant amount of computation to reach the same score level. SOSMC also achieves consistently low variance in scores (evidenced by the thin orange lines in Figure 9, left), suggesting that it reliably generates high-scoring results on every run. SMC suffers from the order-sensitivity problems discussed in Section 3 but appears to overcome them when given enough particles. MH fares slightly better than SMC in terms of score, and its outputs are also qualitatively more diverse, featuring more interesting variations.

In the building complex example, SMC again suffers from order sensitivity. While it can generate good results, it often fails to generate both sides of the target crescent curve (Figure 9b, right), leading to high variance in scores (Figure 9b, left). MH fares better than SMC and does eventually match SOSMC's scores at high budgets. At low budgets, however, SOSMC generates good volume matches, whereas MH does not have enough time to reliably do so (Figure 9b, right). MH requires twice as much computation as SOSMC to consistently score above -10, the threshold above which results appear consistently 'good' for this example.

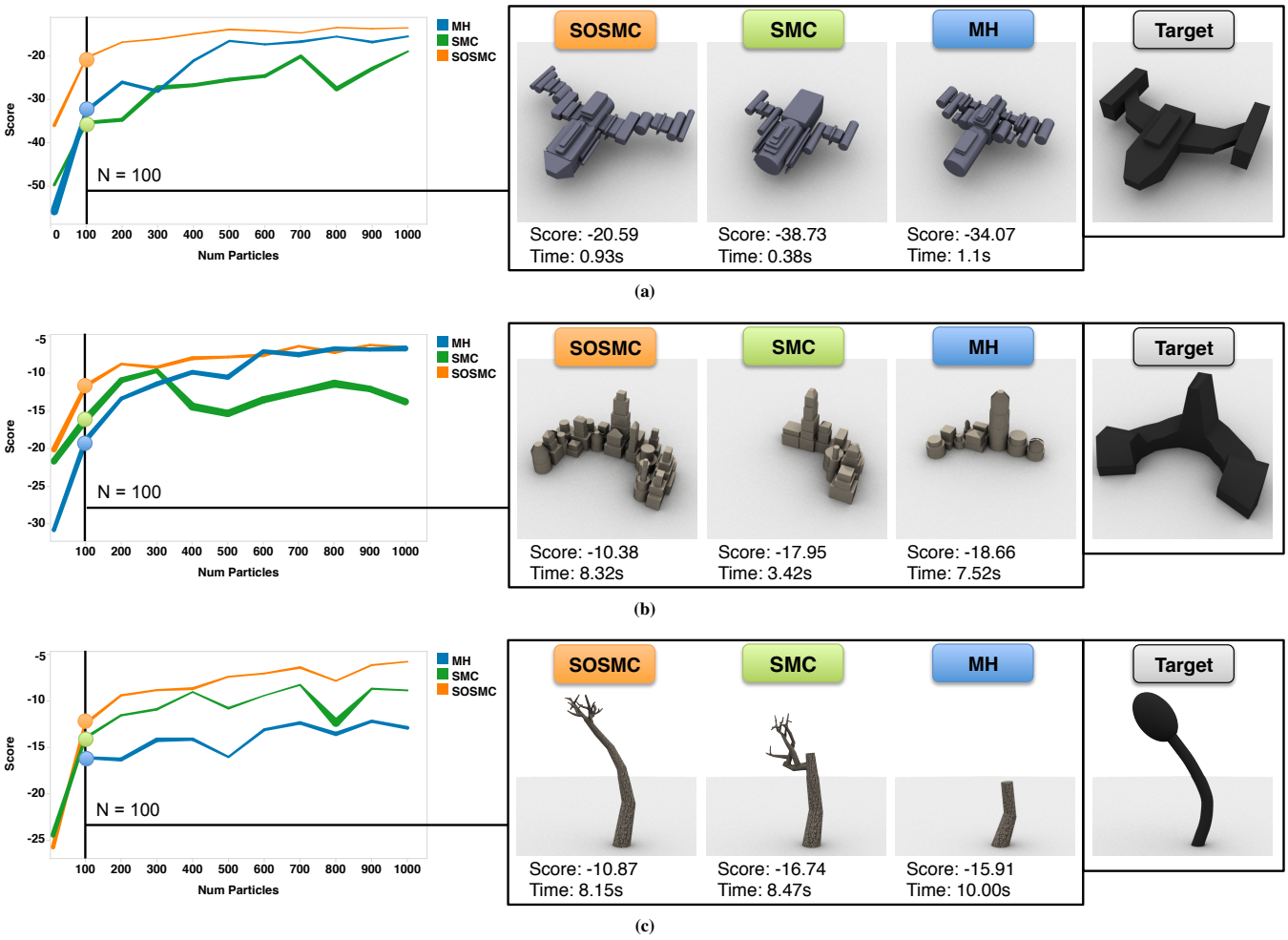


Figure 9: *A comparison of SOSMC, SMC, and MH in generating high-scoring outputs with limited computation time.* (Left) *Maximum score achieved by each method, averaged over 10 runs, as computational budget increases. Line thickness is proportional to variance in high scores over those runs. SMC and SOSMC use the same number of particles; MH runs for as long as SOSMC takes to run on average.* (Right) *Representative samples generated by each method given a computational budget of 100 particles (or equivalent average running time, for MH). SOSMC consistently outperforms both SMC and MH in reliably generating high-quality samples at small budgets.*

For the tree example, SMC's performance is close to SOSMC's, since the target shape has linear structure with branching only at the end. However, order-sensitivity is still an issue, as SMC sometimes generates models that use a large branch where continuing the trunk would be more natural (Figure 9c, right). MH also performs well overall on this example, but there is a persistent gap between its performance and that of SOSMC. MH's proposals—which randomly re-generate subtrees—can fail to discover the long structure of the target shape, especially at low budgets (Figure 9c, right).

Finally, as discussed in Section 5, our SMC implementations suffer significantly worse trace replay overhead than our MH implementation. We expect SOSMC to further outperform MH in the above comparisons as this overhead is eliminated.

7 Discussion

This paper introduced SMC to the task of controlled procedural modeling. We developed the SOSMC algorithm and the stochastic future to handle the multiple possible sequentializations of a procedural modeling program. We demonstrated SOSMC's ability to generate high-quality results for a variety of programs and controls, and we showed that it reliably generates better results under small computational budgets than both depth-first SMC and MH.

7.1 Limitations

SOSMC will not always succeed for all possible programs and score functions. SMC is known to be susceptible to 'garden paths,' or execution traces that look promising for much of their runtime only to become undesirable later on [Levy et al. 2009]. In settings where such paths exist, SOSMC could conceivably perform worse than depth-first SMC, as it may randomly discover garden paths that depth-first SMC cannot follow. For such problems, the ability to revise past decisions is critical, so MCMC or hybrid SMC/MCMC approaches work better [Andrieu et al. 2010].

SMC also needs random choices to be interleaved with evidence (i.e. geometry generation) to work well. If too many random choices are made up-front, the program 'overcommits' itself and proceeds like simple forward sampling. Fortunately, most hierarchical, recursive procedural models can be written in interleaved

style. Simple data flow analysis could be used to push random choices as close as possible to their dependent geometry, if the program is not already written in this way.

In addition, SMC can suffer from the 'sample impoverishment' problem: repeated resampling tends to kill off all but one or a few particle execution histories, resulting in a final set of particles whose early execution histories are identical. For procedural modeling programs, this behavior manifests in many near-duplicates in the final set of sampled output models. Ideally, SMC would deliver as many unique samples as it has particles, and there exist a variety of impoverishment-fighting techniques that could help realize this goal [Gilks and Berzuini 2001; Lindsten et al. 2014]. MCMC algorithms suffer from a similar problem in the form of 'mode lock,' wherein the MCMC chain becomes stuck in a small, localized region of the state space.

7.2 Scalability

The examples presented in this paper are relatively simple, using from dozens up to a few hundred primitives, but we believe that SOSMC should scale well to models of increasing complexity. In terms of depth complexity (i.e. how many primitives the program generates), an implementation that avoids trace replay, such as a continuation-based implementaiton, should be able to maintain nearly-constant work per SMC timestep. Some scoring functions, such as intersection testing, could still become more expensive as depth complexity increases, however.

In terms of breadth complexity (i.e. the program's approximate overall branching factor), a high branching factor results in more possible execution orderings, which could require more particles to explore. The results presented in this paper suggest that SOSMC can work well up to branching factor 4 (the Building Complex program) with a reasonable number of particles, but future work should more thoroughly explore this question.

7.3 Future Work

The scoring function applied to incomplete models can be viewed as a type of *heuristic*, as in search algorithms. For the types of controls we consider in this paper, we have seen that using just the raw score of the partially-generated shape can bias SMC toward placing large primitives first. Future work could develop—or perhaps derive algorithms for automatically learning—heuristics that take into account the incomplete model's possible future, as well.

This paper explores procedural models that generate their outputs through repeated addition of primitive shapes, but this is not the only procedural modeling paradigm. Some models evolve a shape over time according to a simulation, such as erosion. Others refine or subdivide a shape, as in recursive fractal terrain generation. These types of models could also be sequentialized and sampled with SMC; understanding the resulting behavior consitutes important future work.

We would like to enable interactive control of procedural models, and the SMC family of algorithms has useful properties for this scenario. One algorithm processes particles asynchronously, which could allow a user to start with a low particle count, receive feedback immediately, and then gradually increase particle count to continuously improve results [Paige et al. 2014]. Another SMC variant injects complete particles from one run of SMC into a new run—this mechanism could allow a user to bias sampler output toward one or a few results of particular interest [Andrieu et al. 2010].

Our presentation of SOSMC considers a stochastic policy π_S that chooses computations to continue uniformly at random, but there may exist better, non-uniform policies for particular programs or score functions. In particular, it may be possible to *learn* such policies, improving them the more often the inference system is used and effectively amortizing the cost of inference over time [Gershman and Goodman 2014].

In the Appendix, we prove that SOSMC is correct, but it may also have provable performance benefits. Given the empirical results presented in Section 6, it seems possible that execution order randomization may improve worst-case performance in a manner similar to multiple importance sampling [Veach and Guibas 1995]. Investigating this possibility is an important avenue for future work.

Finally, SOSMC can also be useful outside the realm of procedural modeling, as any application that involves guided sampling from a structured, non-linear random process could benefit from execution order randomization. Physically-based rendering is one example, where SOSMC could help sample high-energy 'trees' of light paths that involve multiply-branching recursive bounces. SOSMC could also help Bayesian vision-as-inverse-graphics systems, which generate random 3D scenes as hypotheses for the underlying structure of a 2D image [Kulkarni et al. 2014].

Acknowledgments

The authors would like to thank Andreas Stuhlmüller and Ling Yang for helpful discussions, as well as Josh Parnell for procedural modeling advice. Katherine Breeden and Mark Meyer provided useful feedback on earlier drafts of this paper. This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

ANDRIEU, C., DOUCET, A., AND HOLENSTEIN, R. 2010. Particle Markov Chain Monte Carlo Methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 72*, 3.

BENEŠ, B., ŠAVA, O., MĚCH, R., AND MILLER, G. 2011. Guided Procedural Modeling. In *Proc. Eurographics 2011*.

BISIANI, R. 1987. Beam Search. In *Encyclopedia of Artificial Intelligence*, S. Shapiro, Ed.

DE MOURA, A. L., AND IERUSALIMSCHY, R. 2004. Revisiting Coroutines. Tech. rep.

DEVITO, Z., HEGARTY, J., AIKEN, A., HANRAHAN, P., AND VITEK, J. 2013. Terra: A Multi-stage Language for High-performance Computing. In *Proc. PLDI 2013*.

DOUC, R., AND CAPPE, O. 2005. Comparison of Resampling Schemes for Particle Filtering. In *Proc. ISPA 2005*.

DOUCET, A., DE FREITAS, N., AND GORDON, N., Eds. 2001. *Sequential Monte Carlo Methods in Practice*. Springer.

FAN, S. 2006. *Sequential Monte Carlo Methods for Physically Based Rendering*. PhD thesis.

GERSHMAN, S., AND GOODMAN, N. D. 2014. Amortized Inference in Probabilistic Reasoning. In *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*.

GILKS, W. R., AND BERZUINI, C. 2001. Following a moving target—Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology) 63*, 1.

GOODMAN, N. D., AND STUHLMÜLLER, A., 2014. The Design and Implementation of Probabilistic Programming Languages. Retrieved 2014/12/15 from http://dippl.org.

GOODMAN, N. D., MANSINGHKA, V. K., ROY, D. M., BONAWITZ, K., AND TENENBAUM, J. B. 2008. Church: a language for generative models. In *Proc. of UAI 2008*.

GORDON, N., SALMOND, D., AND SMITH, A. 1993. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F 140*, 2.

HALSTEAD, JR., R. H. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst. 7*, 4.

HAMMERSLEY, J. M., AND MORTON, K. W. 1954. Poor Man's Monte Carlo. *Journal of the Royal Statistical Society. Series B (Methodological) 16*, 1.

HELLO GAMES, 2014. No Man's Sky. Retrieved 2014/12/18 from http://www.no-mans-sky.com/.

HMLINEN, P., ERIKSSON, S., TANSKANEN, E., KYRKI, V., AND LEHTINEN, J. 2014. Online Motion Synthesis Using Sequential Monte Carlo. In *Proc. SIGGRAPH 2014*.

KULKARNI, T. D., MANSINGHKA, V. K., KOHLI, P., AND TENENBAUM, J. B. 2014. Inverse Graphics with Probabilistic CAD Models. *CoRR*.

LEVY, R. P., REALI, F., AND GRIFFITHS, T. L. 2009. Modeling the effects of memory on human online sentence processing with particle filters. In *In Proc. NIPS 2009*.

LINDSTEN, F., JORDAN, M. I., AND SCHÖN, T. B. 2014. Particle Gibbs with Ancestor Sampling. *J. Mach. Learn. Res. 15*, 1.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND VAN GOOL, L. 2006. Procedural Modeling of Buildings. In *In Proc. SIGGRAPH 2006*.

MĚCH, R., AND PRUSINKIEWICZ, P. 1996. Visual Models of Plants Interacting with Their Environment. In *Proc. SIGGRAPH 1996*.

PAIGE, B., AND WOOD, F. 2014. A Compilation Target for Probabilistic Programming Languages. In *Proc. ICML 2014*.

PAIGE, B., WOOD, F., DOUCET, A., AND TEH, Y. 2014. Asynchronous Anytime Sequential Monte Carlo. In *Proc. NIPS 2014*.

PEGORARO, V., WALD, I., AND PARKER, S. G. 2008. Sequential Monte Carlo Adaptation in Low-Anisotropy Participating Media. *Computer Graphics Forum 27*, 4.

PROCEDURAL REALITY, 2014. Limit Theory. Retrieved 2014/12/18 from http://ltheory.com.

PRUSINKIEWICZ, P., JAMES, M., AND MĚCH, R. 1994. Synthetic Topiary. In *Proc. SIGGRAPH 1994*.

RITCHIE, D., LIN, S., GOODMAN, N. D., AND HANRAHAN, P. 2015. Generating Design Suggestions under Tight Constraints with Gradient-based Probabilistic Programming. In *Proc. Eurographics 2015*.

ROSENBLUTH, M. N., AND ROSENBLUTH, A. W. 1955. Monte Carlo Calculation of the Average Extension of Molecular Chains. *The Journal of Chemical Physics 23*, 2.

SMITH, A. F. M., AND GELFAND, A. E. 1992. Bayesian Statistics without Tears: A Sampling-Resampling Perspective. *The American Statistician 46*, 2.

STAVA, O., PIRK, S., KRATT, J., CHEN, B., MCH, R., DEUSSEN, O., AND BENES, B. 2014. Inverse Procedural Modelling of Trees. *Computer Graphics Forum 33*, 6.

STEWART, L., AND MCCARTY, JR., P. 1992. Use of Bayesian belief networks to fuse continuous and discrete information for target recognition, tracking, and situation assessment. *Proc. SPIE*.

TALTON, J. O., LOU, Y., LESSER, S., DUKE, J., MĚCH, R., AND KOLTUN, V. 2011. Metropolis Procedural Modeling. *ACM Trans. Graph. 30*, 2.

VANEGAS, C. A., GARCIA-DORADO, I., ALIAGA, D. G., BENES, B., AND WADDELL, P. 2012. Inverse Design of Urban Procedural Models. In *Proc. SIGGRAPH Asia 2012*.

VEACH, E., AND GUIBAS, L. J. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proc. SIGGRAPH 1995*, SIGGRAPH '95.

WINGATE, D., STUHLMÜLLER, A., AND GOODMAN, N. D. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proc. AISTATS 2011*.

WONG, M. T., ZONGKER, D. E., AND SALESIN, D. H. 1998. Computer-generated Floral Ornament. In *In Proc. SIGGRAPH 1998*.

WOOD, F., VAN DE MEENT, J. W., AND MANSINGHKA, V. 2014. A New Approach to Probabilistic Programming Inference. In *Proc. AISTATS 2014*.

XU, K., ZHANG, H., COHEN-OR, D., AND CHEN, B. 2012. Fit and diverse: Set evolution for inspiring 3d shape galleries. In *Proc. SIGGRAPH 2012*, SIGGRAPH '12.

YEH, Y.-T., YANG, L., WATSON, M., GOODMAN, N. D., AND HANRAHAN, P. 2012. Synthesizing Open Worlds with Constraints Using Locally Annealed Reversible Jump MCMC. In *Proc. SIGGRAPH 2012*.

Appendix: SOSMC Proof of Correctness

We aim to show that the marginal distribution over models generated by SOSMC is the same as the marginal distribution over models generated by depth-first SMC.

We will use the fact that a random choice variable r in a trace $\mathbf{r}_n$ can be uniquely addressed by its position in the function call tree of the trace [Wingate et al. 2011]. We call this address addr(r).

Definition 1. Two variables r^1 and r^2 are equivalent ($r^1 \equiv r^2$) if their addresses and values are the same. That is, addr(r^1) = addr(r^2) and $r^1 = r^2$.

Definition 2. Two traces $\mathbf{r}_n^1$ and $\mathbf{r}_n^2$ are equivalent ($\mathbf{r}_n^1 \equiv \mathbf{r}_n^2$) if they contain equivalent variables. That is,

- $\forall r_{n,i}^1, \exists r_{n,j}^2$ such that $r_{n,i}^1 \equiv r_{n,j}^2$.
- $\forall r_{n,j}^2, \exists r_{n,i}^1$ such that $r_{n,j}^2 \equiv r_{n,i}^1$.

In particular, we assume that equivalent traces are considered equivalent by the scoring function $s(\cdot)$—that is, the score assigned to a trace does not depend upon the order in which it was generated.

Assumption 1. If $\mathbf{r}_n^1 \equiv \mathbf{r}_n^2$, then $s(\mathbf{r}_n^1) = s(\mathbf{r}_n^2)$. *

Together, these definitions allow us to group traces into equivalence classes $\mathbf{X}_n$, where all $\mathbf{x}_n = \{\mathbf{r}_n, \mathbf{o}_n\} \in \mathbf{X}_n$ generate the same partial model. Formally, our goal is to show that $P_N^{\pi_D}(\mathbf{X}_N) = P_N^{\pi_S}(\mathbf{X}_N)$. We start with defining the unnormalized density of an equivalence class by marginalizing out all the orderings that generate it. Let $\hat{\mathbf{r}}_n$ be any trace from equivalence class $\mathbf{X}_n$. Then:

$$
\begin{aligned}
F_n(\mathbf{X}_n) &= \sum_{\mathbf{x}_n \in \mathbf{X}_n} F_n(\mathbf{x}_n) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \cdot p_n(\mathbf{x}_n) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\mathbf{x}_m \setminus \mathbf{x}_{m-1}|} p(x_{m,i} | x_{m,1:(i-1)}, \mathbf{x}_{m-1}) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\mathbf{r}_m \setminus \mathbf{r}_{m-1}|} p(r_{i,m} | \text{par}(r_{i,m})) \\
&\qquad \prod_{m=1}^{n} \prod_{j=1}^{|\mathbf{o}_m \setminus \mathbf{o}_{m-1}|} \pi(o_{j,m} | \mathbf{x}_m) \\
&= s(\hat{\mathbf{r}}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\hat{\mathbf{r}}_m \setminus \hat{\mathbf{r}}_{m-1}|} p(\hat{r}_{i,m} | \text{par}(\hat{r}_{i,m})) \\
&\quad \left(\sum_{\mathbf{x}_n \in \mathbf{X}_n} \prod_{m=1}^{n} \prod_{j=1}^{|\mathbf{o}_m \setminus \mathbf{o}_{m-1}|} \pi(o_{j,m} | \mathbf{x}_m) \right) \\
&= s(\hat{\mathbf{r}}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\hat{\mathbf{r}}_m \setminus \hat{\mathbf{r}}_{m-1}|} p(\hat{r}_{i,m} | \text{par}(\hat{r}_{i,m})
\end{aligned}
$$

We can move the $s(\mathbf{r}_n) \cdots$ terms outside the summation because all $\mathbf{r}_n$ are equivalent (Definition 2, Assumption 1) and because the remaining terms—the ordering probabilities—form a discrete probability distribution whose elements sum to one.

By Equation 1, it remains to show that $Z_N^{\pi_D} = Z_N^{\pi_S}$. By the definition of partition function,

$$
Z_N^{\pi} = \int_{\mathcal{X}^{\pi}} F_N(\mathbf{x}) \mathrm{d}\mathbf{x} = \int_{\Omega^{\pi}} F_N(\mathbf{X}) \mathrm{d}\mathbf{X}
$$

where $\mathcal{X}^{\pi}$ is the set of all complete traces that can be generated under ordering policy π and Ω^{π} is the set of all equivalence classes (from here on, we omit the N subscript for brevity). Thus it suffices to show that $\Omega^{\pi_S} = \Omega^{\pi_D}$.

Lemma 1. *$\Omega^{\pi_D} = \Omega^{\pi_S}$, which means*

1. *$\forall \mathbf{x}^D \in \mathcal{X}^{\pi_D}$, $\exists \mathbf{x}^S \in \mathcal{X}^{\pi_S}$ such that $\mathbf{r}^D \equiv \mathbf{r}^S$.*
2. *$\forall \mathbf{x}^S \in \mathcal{X}^{\pi_S}$, $\exists \mathbf{x}^D \in \mathcal{X}^{\pi_D}$ such that $\mathbf{r}^S \equiv \mathbf{r}^D$.*

Proof.

1. $\forall \mathbf{x}^D \in \mathcal{X}^{\pi_D}$, $\mathbf{x}^D \in \mathcal{X}^{\pi_S}$, since the fixed ordering generated by π_D can be generated by π_S with nonzero probability.
2. $\forall \mathbf{x}^S \in \mathcal{X}^{\pi_S}$, create an empty trace $\mathbf{r}^D$ and walk the function call tree of $\mathbf{r}^S$ in depth-first order. When encountering a variable r with location in the call tree given by addr(r), insert that variable into $\mathbf{r}^D$. This process results in a valid trace in $\mathcal{X}^{\pi_D}$ which is equivalent to $\mathbf{r}^S$. □

We have proven that $P_N^{\pi_D}(\mathbf{X}_N) = P_N^{\pi_S}(\mathbf{X}_N)$ for programs that always terminate after at most N steps. Procedural models that explicitly limit recursion depth or that stop when geometric features become too small fit this description. Without such checks, procedural models only *almost* always terminate after a finite number of steps, i.e. termination probability approaches one as the number of steps approaches infinity. The same analysis should hold in this case as well, as probabilistic programs that terminate with probability one have well-defined marginal distributions over execution traces [Goodman et al. 2008]. The proof would require a limit argument on N for approximating finite programs.

*Efficient, incrementalized implementations that use intermediate results of $s(\mathbf{r}_{n-1})$ to compute $s(\mathbf{r}_n)$ must guarantee this property.