Declarative Processing for Computer Games

Walker White
Cornell University
wmwhite@cs.cornell.edu

Benjamin Sowell
Cornell University
sowell@cs.cornell.edu

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Alan Demers
Cornell University
ademers@cs.cornell.edu

Abstract

Most game developers think of databases as nothing more than a persistence solution. However, database research is concerned with the wider problem of *declarative processing*. In this paper we demonstrate how declarative processing can be applied to computer games. We introduce the *state-effect* pattern, a design pattern that allows game developers to design parts of their game declaratively. We present SGL, a special scripting language which supports this design pattern and which can be compiled to a declarative language like SQL. We show how database techniques can process this design pattern in a way that improves performance by an order of magnitude or more. Finally, we discuss some design decisions that developers must make in order to adopt this pattern effectively.

CR Categories: D.3.2 [Programming Languages]: Language Classifications—[Specialized application languages]

Keywords: Games, Scripting, Databases

1 Introduction

Scalability is a fundamental problem in the development of computer games. Players constantly demand more — be it more polygons, more physics particles, more AI behaviors, or, in the case of massively multiplayer online games (MMOs), more players.

Scalability has long been a focus of the database community. However, the game development industry has done little to exploit database research in this area. Most developers think of databases as a persistence solution, designed to store and read game objects from disk. As such, almost all database usage in games design has been within the development of MMOs.

However, the advantages of databases are not limited to persistence. It is true that commercial databases focus on persistence, but that is because this is what their customers have demanded. Database research has dealt with the much larger issue of processing *declarative languages*. Declarative languages, such as SQL, transform one collection of data to another without specifying the exact side effects in-between. Because the query optimizer is free to process the transformation however it wants, operations written declaratively can be highly optimized and parallelized, often with orders of magnitude improvement in performance.

It would be both difficult and unnatural to write all of a game in a declarative language. However, there are large subsystems of computer games that are amenable to being processed declaratively. For example, game developers have designed algorithms to parallelize physics simulations with GPUs [Nguyen 2007]. Much of this work involves custom pipelines that are very similar to techniques developed decades ago by the database community. However, instead of a general pipeline model that can be reused over and over again, game developers come up with a distinct solution every time they encounter a new algorithm. With a more principled approach, the time spent engineering this pipeline could be freed up for more time improving game play.

Declarative processing opportunities are not just limited to physics algorithms. The authors have published work demonstrating how declarative processing can improve the computational performance of scripted character AI by an order of magnitude [White et al. 2007]. We believe there are many other possibilities for declarative processing in games. By providing game developers with tools to design declaratively, we can help them overcome many of the difficulties in game scalability.

The aim of this paper is to introduce our work on declarative processing to the game design community. In doing so, we explore the design decisions necessary to adequately leverage this processing model. In this paper, we make the following contributions:

- In Section 2 we show how declarative processing can be applied to games using a novel design pattern, which we call the *state-effect pattern*.
- In Section 3 we present SGL, a custom scripting language for the state-effect pattern. SGL allows designers to benefit from database-style optimizations without any knowledge of SQL or other query languages.
- In Section 4 we provide an overview of the optimization techniques that a query processor can apply to the state-effect pattern. These techniques can result in orders of magnitude improvement in performance.
- In Section 5 we identify and discuss some of the challenges that designers may face in adopting the state-effect pattern.

In Section 6 we give an overview of related work. Finally, we conclude with some directions for future work in Section 7.

2 The Declarative Processing Model

One of the main ways in which declarative processing achieves performance gains is by intentionally limiting *expressiveness*. In other words, some declarative languages are not as expressive as traditional programming languages, but they can be more efficiently optimized.

A canonical example of this is the relational algebra, a popular declarative language implemented by SQL. The relational algebra does not support arbitrary iteration: it does not have `for` or `while` loops and does not support recursion. The only type of iteration that it does support is the `for-each` loop, which iterates through the elements in a list. This specialized type of iteration — called a "join" in the database literature — has been heavily researched and highly optimized. As we show throughout this paper, this weaker form of iteration is still capable of expressing a wide range of complex and interest behavior. Furthermore, restricting ourselves to this form of iteration can result in substantial performance benefits.

To provide games with these performance benefits, we need to identify and separate joins from arbitrary iteration in the game logic. In general, this is a very difficult problem. A better solution is to identify design patterns that are adaptable to our optimizations and to provide tools for processing these patterns. In this section we present a basic design pattern that allows for extensive declarative optimizations. In addition, we illustrate this pattern with several examples showing its wide range of applicability.

2.1 The State-Effect Design Pattern

We call our design pattern the *state-effect pattern*. In practice, this pattern is applied to part or all of the game simulation loop (i.e. the part of the game that processes object behavior and updates the game state). In this pattern, we separate the attributes of the game objects into two disjoint types: states and effects. Naively, states are values that change only at the end of the simulation time step, but which remain constant during the main computational phase. Effects, on the other hand, are ephemeral values that can change within the simulation loop as the object interacts with other objects in the game. In addition to separating states and effects, we put restrictions on how these two types of object attributes can interact with one another.

Before we outline the pattern in detail, we first consider an example that is already familiar to most game developers: a particle system. In its simplest form, the attributes of an object of a particle system are mass, volume, position, velocity and (possibly) acceleration. In addition, during a single simulation time step, each particle has a force attribute which represents the interactions with all of the other objects in the particle system. The simulation loop computes this force attribute, and uses it to update the other particle attributes in the following way:

(1) It sums up all of the forces acting upon each particle.

(2) From these forces, it computes the new acceleration and/or velocity for each particle.

(3) It tentatively moves the particles each to their new position according to the acceleration and/or velocity.

(4) Finally, it searches for any collisions that occurred during the process, and backs them out as necessary.

Ignoring for now the collision detection and resolution phase in step (4), we can easily separate this process into states and effects. The force attribute is an effect that is computed from interactions with other objects. It can be computed using a join, as we just loop over all the other game objects with a `for-each` and sum the results together. The other attributes — position, velocity, acceleration — are only updated at the end of the loop and therefore may be classified as state. Furthermore, state values are updated using only a simple mathematical calculation from the force attribute and the existing state values; no further iteration is required.

A powerful feature of this design is that, because the state is not updated until the end of the simulation loop, the force calculations can all be isolated from one another. We can evaluate the game objects in any order, or even in parallel, and still get the same answer. This feature has been heavily utilized by game engineers to parallelize physics computations on modern GPUs [Nguyen 2007]. The ability to do order-independent processing is the crucial property that allows us to optimize this calculation with database techniques.

Given the particle system as our motivation, we now outline the state-effect pattern in more detail. It consists of the following parts:

- **Separation of object attributes into states and effects.**
  State attributes remain unchanged for the majority of a simulation time step, only changing at the end. Effect attributes, on the other hand, support intermediate computation. They can change within the simulation loop, but are "reset" at the beginning of every time step. In those cases where we want an attribute to be both a state and an effect, we make a separate attribute for each role.

  In the particle system example, force is an effect while all other attributes are state.

- **Rules for combining effects within a single time step.**
  Intuitively, effects represent actions that a game object can perform either on itself or on other objects. These effects can include actions from intelligent agents, such as the result of a magic spell, or actions from inanimate objects, such as the force exerted by one particle on another. As all actions in a single time step are applied simultaneously, we need order-independent rules for combining these effects to produce a single net effect.

  In our particle system example, summation is the rule for combining individual force effects. This rule is independent of order since summation is associative and commutative.

- **(Query Phase) Specification of object interactions.**
  For each game object, we need instructions specifying what effects it either generates or incurs. If we were to use a scripting language to implement our pattern, these instructions would be scripts attached to the individual game objects. These instructions have one major restriction; the query phase may not have any form of iteration other than a join (e.g. a `for-each` loop).

  In our particle system example, the query phase consists of the mathematical formulae computing the force between pairs of particles.

- **(Post-processing Phase) Specification for updating state.**
  At the end of the simulation loop, we need instructions for how to compute the new state from the effects that we have. For efficiency, we would prefer that the instructions be simple calculations done in straight-line code. However, to make it easier to integrate this pattern into the rest of the game, we place no restrictions on this part of the pattern; it may even include instructions that cannot be computed declaratively.

  In our particle system example, steps (2) and (3) are examples of simple post-processing which updates the state from the effect. Furthermore, since there are no restrictions on what we do in the post-processing phase, we can include step (4), which may involve unsafe iteration.

The names "query phase" and "post-processing phase" are chosen to reflect the ways in which we process these steps. Suppose that each of our game objects were a row in a database table. Because of our restrictions on the query phase, we could process it as a single database query that returns a new table of effects on each of these objects. The post-processing phase then uses a standard programming model to update these objects using the values in this effect table. As we show in Section 4, the benefits of using traditional database techniques to speed up the query phase can be enormous.

The primary difficulty with this design pattern is that we have to be careful with what we put in the post-processing step; it can easily become a "catch-all" step for parts that are hard to separate. There is little advantage for our pattern in the post-processing phase, as it is handled normally without declarative optimizations. Hence, if we put too much in this phase, the cost of this phase can dominate the optimizations described in Section 4. We discuss this issue in more detail in Section 5.2.

2.2 An Illustrative Example

While particle systems is a natural example of the state-effect pattern, the pattern is by no means restricted to physics applications. For a more sophisticated example, we consider the case of scripted character AI. We apply the state-effect pattern to design a real-time strategy game in which every unit is individually scripted.

First we define the state and effects. While our units have many attributes, the important ones for our example are as follows:

- **State**: player, unit type, health, x position, y position
- **Effects**: damage, healing, x velocity, y velocity,

We use sum as our rule to combine the damage effects. That way, if the unit is attacked by multiple opponents, all the damage effects stack on on another. Similarly, we use sum as the combination rule for the velocity components. On the other hand, we use a different rule for healing. which we model after the healing auras from *Warcraft III*. These auras do not stack with one another; at each time step, a unit receives healing only from the active aura of the greatest power. Hence our combination rule for healing is maximum, not sum. This gameplay feature illustrates why we separated damage and healing into two separate effects, even though they are both used to update the health state.

Next we specify the query phase. To keep our example simple, we limit ourselves to two unit types, healers and archers. We assign them the following simple behavior as part of the query phase:

- If the unit is an archer, search for the weakest enemy unit (e.g. the one with the least health) within firing range.
  - If no such unit is found, set the velocity effects to move towards some predefined location (e.g. the enemy base, the mouse click location, etc.).
  - If a unit is found, assign some fixed amount of damage to the enemy unit.
- If the unit is a healer, apply a fixed amount of healing to all units in range of the healing aura.

Finally our post-processing stage updates the game state in the obvious way. That is, we add the velocity to the position to get the new position. Furthermore, we subtract the total damage from the health, but add the healing.

Because we have used the state-effect pattern, we can express the query phase as a database query. Figure 1 illustrates how to represent our simple example as a nested query in SQL (for this example, `DIST()` and `IN_RANGE()` are predefined functions introduced to simplify the query).

This query demonstrates a serious issue with our design pattern: the query is unreadable to anyone without extensive SQL experience. Moreover, the SQL query expresses the behavior for all types of units, making it difficult for the game developer to program individual character behavior. In order to use this design pattern effectively, game designers will need tools to specify character and object behavior individually, but then compile all this behavior into a single declarative expression for optimization.

```
SELECT U.PLAYER, U.TYPE, U.HEALTH, U.X, U.Y,
       SUM(E.DAMAGE), MAX(E.HEAL), SUM(E.VX), SUM(E.VY)
FROM Units U,
     (SELECT T.KEY, 0 as DAMAGE, HAMOUNT AS HEAL,
             0 as VX, 0 as VY
      FROM Units T, Units H
      WHERE H.TYPE="Healer" AND H.PLAYER = T.PLAYER
            AND IN_RANGE(T,H)
      UNION
      SELECT T.KEY, 0 as DAMAGE, 0 as HEAL,
             (GOAL_X-T.X)/DIST(T.X,GOAL_X) AS VX,
             (GOAL_Y-T.Y)/DIST(T.Y,GOAL_Y) AS VY
      FROM Units T, Units A
      WHERE NOT EXISTS (SELECT *
                        FROM Units T
                        WHERE IN_RANGE(T,A)
                        )
      UNION
      SELECT MIN(T.KEY), DAMOUNT as DAMAGE,
             0 as HEAL, 0 as VX, 0 as VY
      FROM Units T, Units A
      WHERE IN_RANGE(T,A) AND A.TYPE = "Archer"
            AND A.PLAYER <> T.PLAYER
            AND T.HEALTH = ANY(SELECT MIN(S.HEALTH)
                                FROM Units S
                                WHERE IN_RANGE(S,A) AND
                                      A.PLAYER != S.PLAYER
                               )
     ) as E
WHERE U.KEY = E.KEY
GROUPBY U.KEY
```

Figure 1: *SQL Expression for the RTS Query Phase*

3 The SGL Scripting Language

The simplest type of tool that we can provide to game developers is a scripting language. Developers have a long history of creating highly specialized and restricted scripting languages. Sometimes these restrictions are imposed to make processing efficient, such as Microsoft's rule specification language in *Age of Kings* [Ensemble Studios 2000]. Other times limitations are imposed in order to keep designers from introducing unsafe behavior: Cryptic Studios removed iteration from one of the scripting languages for *City of Heroes/City of Villains* after designers repeatedly introduced infinite loops into the game [Posniewski 2007]. In our case, the scripting language is introduced to help the game engine recognize uses of the state-effect pattern and optimize them accordingly.

We call our scripting language the Scalable Games Language (SGL), because of its ability to process large numbers of game objects. SGL consist of two types of files — data files and script files. The data files define the data types ("classes") in our language, while the script files define the behavior for a given data type during the query phase.

Every game object accessed by a script must have an associated data file in order to make its attributes accessible. The list of attributes need only include those that are accessed by the scripting language; game objects may have other attributes that are used by the game engine proper, but are not visible to the scripting language. Figure 2 shows an example of a data file for the RTS example in Section 2.2. The format is similar to a C++ class. The data type comprises several fields, which are explicitly separated into states and effects. While later versions of SGL will support a wider array of field types, currently a field may either be a number, another data type specified by a data file, or an (unordered) set of values (e.g. Set$\langle$Unit$\rangle$).

For state fields, the data file specifies initial values for new objects allocated by the script. Effect fields, on the other hand, specify only a combination function. Currently SGL supports only built-in combination rules: sum, minimum, maximum, and average for numbers, and union for set types. Object types are combined by *priority functions*, which take a set of objects and select a unique element from the set according to specified rules.

The most basic priority function is `priority`; this function uses an opaque priority value built into the system. Custom priorities are introduced using the functions `argmin` and `argmax`. These func-

```
class Unit {
state:
  number player = 0;
  number type = 0;
  number x = 0;
  number y = 0;
  number health = 0;
  Unit target = null;
effects:
  number vx : avg;
  number vy : avg;
  number damage : sum;
  number healing : max;
  Unit acquired : priority;
update:
  x = x + vx;
  y = y + vy;
  health = health - damage;
  target = (acquired != null ? acquired : target);
}
```

Figure 2: *Sample Datatype Declaration*

tions take an expression ⟨EXP⟩ and a set of objects. The expression is evaluated on each object in the set, and the function returns the one with the minimum (or maximum) value. For example, we could use `argmin` to chose the enemy unit that is closest to our character; we simply provide the expression computing distance to the function `argmin`. In those cases where there is more than one minimum (or maximum), these functions use the built-in priority value to break ties.

In addition to attributes, the data file also specifies update rules. The update rules are expressions defined in terms of the effect and state fields. They are used to define the new value of each state field at the end of the post-processing stage. For example, in Figure 2, the update rules use the velocity effects to adjust the position state. Therefore SGL can handle very simple post-processing instructions, like steps (2) and (3) in our particle system example.

For more complicated post-processing operations, we rely on the integration between SGL and the game engine. The game engine invokes SGL by calling a function which processes a single step of the query phase, followed by an application of the update rules. The game engine is free to do whatever operations it wants to the game objects before invoking the script again. Thus we can safely ignore more complex post-processing examples such as step (4) in our particle system example.

The script files are structured to look like an imperative programming language, even though they are processed declaratively. This is intended to make the scripting language more accessible to designers who have no experience with declarative languages. Each script file is associated with a data type, and all of the fields of that data type may be accessed in that script without a dot. In addition, fields of other objects are accessed using the traditional "dot selection" found in most object-oriented languages. For example, in a script associated with a unit type, "x" is the x position of the unit executing the script, while "target.x" is the x position of the current combat target of the script executor.

The scripting language has many features common to scripting languages, such as `if-else` conditionals. Furthermore, it has expressions for manipulating numbers, objects, and sets of objects. For brevity, we focus on the most important parts of the script.

The primary purpose of the script is to assign values to the effect fields. A script may assign a value to an effect field of the object executing the script, or to an effect field of another object. In order to process the script as efficiently as possible, there is no guarantee on the order in which the values are assigned to the effect fields;

```
let (number dist = (x-target.x)*(x-target.x)+
                   (y-target.y)*(y-target.y)) in {
   if (dist < ATTACK_RANGE) {
      // If in range, attack.
      target.damage <- DMG_AMOUNT;
   } else {
      // Else move closer (use unit velocity)
      vx <- (target.x-x)/dist;
      vx <- (target.y-y)/dist;
   }
}
```

Figure 3: *Assigning Effect Values*

if more than one value is assigned to an effect value, then these values are combined using the rules specified in the data file. Additionally, in order to prevent hazardous side effects such as reading effects assigned out of order, effect fields may never be read — they are write-only. In essence, our effect fields work like the aggregate variables found in Sawzall, Google's highly parallelizable data processing language [Pike et al. 2005]. Indeed, our notation for writing to an effect field, "<-", is the same as Sawzall's.

SGL provides local variables to store and access the results of intermediate computation. However, as in a functional language, these variables may never be reassigned after they are defined. As long as there is no iteration in the script, this does not actually place any restrictions on the designer. Furthermore, since we handle iteration specially, we believe this is an acceptable trade-off in language expressiveness to make processing simpler.

To introduce an intermediate variable in a SGL, we use the syntax

```
let ⟨TYPE⟩⟨identifer⟩ = ⟨EXP⟩ in { ⟨BLOCK⟩ }
```

The scope of this variable is the code block surrounded by the braces. Figure 3 shows the use of a `let` declaration to represent a unit choosing between two actions, depending on the distance between it and its combat target. SGL supports multiple variable assignments in a single `let` statement; they must all be part of a comma-separated list.

In addition to `let` statements, SGL supports a limited form of iteration. Again, in order to improve our ability to optimize the execution of a script, we wish to avoid variables that can be both read and written arbitrarily. Therefore, our design of this iteration loop itself follows the state-effect pattern. We call our iteration `accum`-loops, and they have the syntax

```
accum ⟨TYPE⟩⟨identifier⟩₁ with ⟨COMBINATOR⟩
      over ⟨TYPE⟩⟨identifier⟩₂ from ⟨EXP⟩ {
  ⟨BLOCK⟩₁
} in {
  ⟨BLOCK⟩₂
}
```

Naively, this loop uses the first code block, ⟨BLOCK⟩$_1$, to iterate over the elements in the set ⟨EXP⟩, and then makes the results of that iteration available to the second code block, ⟨BLOCK⟩$_2$. Within the first code block, the variable ⟨identifier⟩$_1$ is treated as an effect field. We make no guarantees on the order in which the accumulation loop is processed; the elements in ⟨EXP⟩ can be processed in any order, or even in parallel. Therefore, within ⟨BLOCK⟩$_1$, the variable ⟨identifier⟩$_1$ may never be read, and values are assigned to it using the same "<-" operator as effect fields.

At the completion of ⟨BLOCK⟩$_1$, the `accum`-loop combines all of the values assigned to ⟨identifier⟩$_1$ using the combination function ⟨COMBINATOR⟩. This combinator function can be any of the functions we used for effect fields. Once these values are combined, they may be read from the variable in the second code block, ⟨BLOCK⟩$_2$. As with a `let` statement, the variable ⟨identifier⟩$_1$ may never be reassigned in ⟨BLOCK⟩$_2$; it is read-only.

```
// Compute the minimum health of a possible target
accum number healthvalue with min
      over Unit u from UNIT {
   let (number dist = (x-u.x)*(x-u.x)+
                      (y-u.y)*(y-u.y)) in {
      // Only select enemies in range
      if (u.player != player && dist < ARCHER_RANGE) {
          healthvalue <- u.health;
      }
} in {
   // Find the unit with that health.
   accum Unit weakest with priority
         over Unit w from UNIT {
      let (number dist = (x-w.x)*(x-w.x)+
                         (y-w.y)*(y-w.y)) in {
         // Be sure to select with same rules
         if (w.player != player && dist < ARCHER_RANGE
             && w.health = healthvalue) {
             weakest <- w;
         }
      }
   } in {
      if (weakest != null) {
         acquired <- weakest;
      }
   }
}
```

Figure 4: *Accum-Loop Specification of Target Acquisition*

Figure 4 illustrates the use of two `accum`-loops to specify the target acquisition behavior of our archers in Section 2.2. The first `accum`-loop finds the minimum health of an acceptable target in range, and the second searches for a target with that health. This is not the most efficient way to express this particular behavior — a more efficient way would use the `argmin` priority function — but it is a simple example showing how `accum`-loops can be coordinated to produce interesting behavior. Furthermore, a typical query optimizer can recognize that these two `accum`-loops have similar structure and combine them automatically.

All of our language structures have been designed so that they can be compiled into the relational algebra, the declarative language used by SQL. As a result, the SGL compiler can gather all of the script files and compile them into a single database query. As we show in the next section, this allows us to improve script performance by orders of magnitude. Thus while SGL does have some very unusual language features — such as the `accum`-loops — the performance enhancements make their adoption worthwhile.

4 Advantages of Declarative Processing

The advantage of representing the query phase by a database query is that we can apply query optimization technology to process it efficiently. While no commercial query optimizer is targeted at precisely the kind of workload found in computer games, there is a substantial body of research on in-memory databases [Boncz and Kersten 1995; Bohannon et al. 1997], and more recent research has focused on optimizing queries over streams of data [Carney et al. 2002; Motwani et al. 2003; Demers et al. 2007]. When combined with traditional database query optimization techniques, this work can be directly applied to game processing.

In this section we explain a few ways in which known database technology can be used to process the state-effect design pattern efficiently. These techniques are not new; indeed, one of the strengths of the state-effect pattern is that we have so many existing optimization techniques to choose from. We present them here just to give the reader a sense of the advantages of adopting this pattern.

4.1 Aggregate Indexing

A well-known difficulty encountered in the design of computer games is the "n^2-problem". This occurs when one game object needs to iterate over most or all of the other game objects in order to determine its behavior. For example, suppose we want to design a game character that runs in fear if it encounters a large number of marching skeletons. The probability that the character runs is determined by the number of skeletons it sees, so the character needs to count skeletons. In a naive implementation, the game engine would take each character and enumerate all the game objects visible to that character, incrementing a counter for each skeleton encountered. If a character can see most of the other game objects, this process requires $\Omega(n)$ steps. Thus, in a situation where there are $\Omega(n)$ skeleton-phobic characters, each of which can see most of the other game objects, the total processing time is $\Omega(n^2)$. This cannot be remedied by computing the number of skeletons only once, as different characters may have different fields of vision resulting in different skeleton counts.

Examples like this arise frequently in practice. Military simulations, in particular, often encounter cases in which "everyone sees everyone else" [Kruszewski and van Lent 2007]. Database technology can help us here. In the state-effect pattern, enumeration is done using `accum`-loops, and at the end of an `accum`-loop the effect results are combined into a single value using an *aggregate function* like sum or max. Databases have developed aggregate processing techniques that are often much faster than the naive $\Omega(n)$ per query. One common solution is the *aggregate index*, which effectively pre-computes the aggregate on subsets of the data in such a way that the result for any query can be computed by combining a small number of the pre-computed aggregates. In many instances, an aggregate index can reduce the cost of our `accum`-loop computation from $\Omega(n^2)$ to $O(n \log n)$.

A good query optimizer can usually infer when to use an aggregate index by static analysis of a database query. Thus, the game designer does not need to worry about building and maintaining a custom index for each new application; the query optimizer handles it automatically.

In an early prototype of SGL, the authors developed a battle simulation in a simple RTS that demonstrated the advantages of aggregate indexing [White et al. 2007]. The simulation was designed to have very complex behavior, with each unit interacting with large numbers of other units. For example, armies of archers continually polled the locations of both enemy units and allied knights, so that they were always able to hide behind the knights for cover. In addition, healing units distributed themselves among allied units so that their healing auras had maximum effectiveness. A large number of aggregate indices were required to implement this behavior. For exact details, we refer the reader to our original paper.

Timing results for this simulation are shown in Figure 5. This graph plots the number of simulated units against total processing time for 500 simulation steps, comparing the naive algorithm to the use of aggregate indexing.

Figure 5 clearly demonstrates the value of aggregate indexing. The overhead of building and maintaining all the aggregate indices is not noticeable even when the number of units is small; and the indices enable the system to scale to an order of magnitude more units using a reasonable simulation time step of 100 ms.

4.2 Dynamic Optimization

Aggregate indexing is not always the most efficient way to process iteration. If it is not true that "everyone sees everyone else," it may

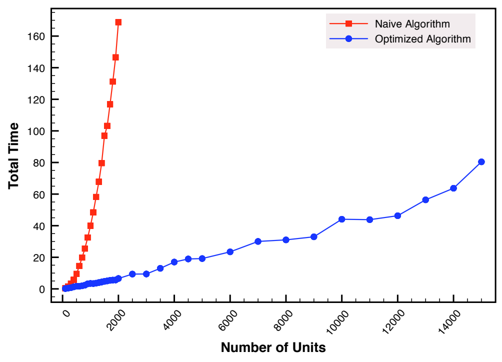

Figure 5: *Comparison of Naive Processing versus Indexing*

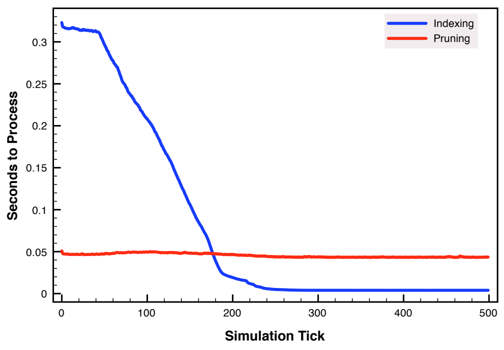

Figure 6: *Comparison of Aggregate Indexing versus Pruning*

be better just to prune the search space and iterate over the pruned objects normally. This solution is commonly used in games, and it often yields excellent results.

The problem is that the choice between aggregate indexing and simple pruning depends on the current state of the game. For example, consider a game in which we want to distribute healers for maximum effectiveness. A simple algorithm assigns to each healer a "healing priority." At each time-step, each healer searches its visible range for wounded allies whose health is below this priority. With uniformly distributed priority values, this algorithm drives healers to congregate in areas with the most wounded. Obviously, if there are few wounded units, we can process this behavior very efficiently by just pruning out the healthy allies. However, in a pitched battle with large numbers of wounded, we need a different technique (like aggregate indexing) to avoid the n^2 problem.

Figure 6 shows the results of simulating the healer behavior described above over several hundred time steps. This simulation uses the same prototype as the previous aggregate index simulation [White et al. 2007], but we have isolated the performance to the healer behavior only. The graph compares the cost of building and maintaining an aggregate index to that of simple pruning.

The simulation begins at the end of major battle. All allied units are wounded, and so pruning is not very effective. Over time, the wounded allies are gradually healed until eventually no more wounded remain. As the number of wounded decreases, the cost of pruning decreases, overtaking the cost of the aggregate index and eventually outperforming it by another order of magnitude.

Ideally, we would like to keep statistics that enable us to predict which of pruning or aggregate indexing will be more efficient on the current game state, so we can switch between these query plans as necessary. This technique is called *dynamic query optimization*. There has been substantial research on dynamic query optimization, particularly on streaming data [Zhu et al. 2004]. In a simulation environment, we can theoretically achieve the performance of the optimal query plan at each time step with a constant (multiplicative) overhead. In practice we can often do much better than this.

4.3 Other Optimizations

While our early work on SGL has focused on index optimization, there are other other known optimization techniques that should apply to the state-effect pattern. Two that are of particular interest for games are pipelining and parallelization. While we have not applied these techniques specifically to SGL, there has been some recent work on adapting them to in-memory systems.

Pipelining in database processing works much the same way that it does in computer graphics. A database query plan consists of a number of simple operators connected together in a graph. With pipelined evaluation, rather than materializing the entire output relation of each operator and storing it in memory, we instead generate output values incrementally, "on demand." This can greatly reduce the amount of memory required for query processing: instead of allocating memory for all intermediate results, we provide only small input and output buffers for each operator. Large memory blocks are required only for aggregate indices and for certain "blocking" operators (such as sorting). This is particularly useful for effect fields, which exist only during the query phase and update rules, and thus need to be materialized only in the input and output buffers of the pipeline.

Game designers sometimes develop custom streaming algorithms such as pathfinding [O'Brien and Stout 2007; Dunki 2008] or GPU physics simulations [Nguyen 2007]. In our approach, pipelining comes for free when we compile the query phase into a declarative format. Thus the game developer can focus on designing the behavior without having to construct custom pipelines.

Another class of optimizations relevant to games is parallelization. Because database queries have no side effects (they take streams of data as input and produce new streams of data as output) they are embarrassingly parallel [Dewitt et al. 1990; Stonebraker et al. 1988]. Indeed, there has even been recent work on processing database queries with GPUs [Govindaraju et al. 2004]. In the case of SGL, it is obvious how to parallelize our query phase. As effect variables can never be read, we can isolate the objects in the query phase from one another and process them separately. We can also parallelize any `accum`-loop for exactly the same reason.

5 Game Design Issues

While the state-effect pattern provides several performance advantages, it is unusual and does require that the game designer structure the game logic appropriately. While our formalization of the state-effect pattern is still very new, we have already identified two major design challenges from our initial prototypes.

5.1 Designing Effects

The most obvious challenged is the design of effects. Very often, games update the state of an object in the middle of the simulation loop. In order to take advantage of the state-effect pattern, the game developer must design effects that appropriately delay the update until the end of the simulation loop.

Fortunately, game designers already do work with effects. They are particularly common in RPGs; these games have paralyzation effects, poison effects, performance enhancing effects, and so on. Designing a single effect is usually straightforward. The game designer introduces a field for the effect, and specifies rules for handling those cases when the effect is produced more than once in a single time step. While this may result in a large number of fields, the pipelining process mentioned in Section 4.3 can minimize the amount of memory necessary to process them.

A greater challenge is managing the interplay between different effects. For game balance purposes we may not want certain effects to stack with one another. For example, we may want a character to receive a strength enhancement effect or a speed enhancement effect, but not both. One way to implement this would be to model strength and speed enhancements as separate effect fields, but write update rules so that only one of them is used at a time. However, this becomes unwieldy as we introduce more incompatible effects.

An alternate solution is to use an object to represent a performance enhancement effect. Then, in order to choose the single effect that takes effect, we use a priority function, — such as `argmin` or `argmax` — to choose among them. For example, in modeling performance enhancement, we may want an object with three fields: the enhancement type, the enhancement strength, and the rank of that enhancement, where enhancements of higher rank are give higher priority over those of lower rank. When we assign an enhancement to a character via an effect field, we use `argmax` to select the enhancement of greatest rank.

Another interesting example is the use of effects for object acquisition. In Section 3 our unit type had a field to represent its current combat target, and we used an object effect field with a priority function to update this value. Another example would be the use of an object effect to assign loot to characters after a successful boss kill. Loot assignment presents an interesting challenge, since we want to ensure that only one character gets the object. Typically this is not an issues, as the game processes each player separately in turn. However, when we use the state-effect pattern, players may processed simultaneously, so we cannot check whether another player has received the loot in the same time step.

The solution to this problem is to understand what our semantic constraints are. While players may have multiple items of loot, each loot may have only one player. Because the uniqueness constraint applies to the loot and not the player, the correct design in this case is to make the owner an effect field in the loot object. That way the loot has a priority function for choosing its unique owner in the case of ties. Once the owner has been determined, the object can be added the character's inventory either by the update rules (SGL supports expressions for manipulating sets of objects) or in a subsequent iteration of the update loop.

With some practice, our current formulation of the state-effect pattern can express a wide range of effect interactions. As we continue to explore the uses of this pattern, we may find ways to extend it to other types of effect interactions. This is an interesting area for future work.

5.2 Integrating the State-Effect Pattern

Another challenge with the state-effect pattern is in integrating it with other parts of the game engine. First of all, we must identify what parts, if any, of the game engine are amenable to the state-effect pattern. Some subsystems are very easy to model in this framework, while others cannot be modeled at all. The most significant limitation is that we cannot use this pattern search over the transitive closure of a graph. This means that it cannot support pathfinding algorithms like $A^\star$ which crawl over a terrain graph. On the other hand, steering algorithms [Reynolds 1999] are similar to force calculations in a particle systems and are quite amenable to the state-effect pattern.

The first step in the design process is to identify all instances of iteration, such as `for`-loops or `while`-loops. Any iteration which can be expressed as a `for-each` loop, or unrolled as a constant number of iterations, is a candidate for the query phase. However, we must also examine the variable assignments within each iteration loop. These assignments must all be expressible as effect variables; otherwise they will be lost when the iteration is done. Obviously we can do this when the iteration is simple aggregation (e.g. summing, averaging, building up sets of objects). With greater proficiency in effect design, we are capable of doing more.

For cases such as pathfinding, which cannot be modeled by this pattern, we still may be able to improve performance through more sophisticated use of the state-effect pattern. The state-effect pattern assumes that it is part of the main simulation loop. However, there is nothing preventing us from embedding the state-effect pattern in other iteration loops as well. In SGL, a call from the game engine only performs a single query phase and update, allowing us to position it where ever we want. Thus we can apply the state-effect pattern to the *internals* of these algorithms, and then iterate over this pattern externally in the game engine.

However, this type of design may require that we reorder the application of iteration. For example, pathfinding iterates over all of the characters searching for a path, and, for each character, uses $A^\star$ to perform a search of the terrain graph. The outside iteration is acceptable, while the inner iteration is not. We can extend our state-effect pattern to pathfinding if we swap the order of iterations. In this case, we use the query phase to pipeline the processing of all of the characters at once, and iterate over the steps in the $A^\star$ algorithm outside in the post-processing phase. It is not clear whether this gives any performance benefit over existing high-performance algorithms, but it does illustrate the basic principle.

An alternative to this extension of the state-effect pattern, is to simply separate troublesome calculations and integrate them into the post-processing phase. Once again, pathfinding is an excellent example, as most game engines implement it in a separate subsystem which communicates the set of computed waypoints to the steering algorithms [O'Brien and Stout 2007]. As SGL supports set fields, we can easily pass this information from the pathfinding system to the steering algorithms in the query phase. Furthermore, as it is a separate subsystem, we can either process the pathfinding during the post-processing phase, or asynchronously in a separate thread.

The right design choice in each of these instances depends upon the application. While we should always be reluctant to move work out of the declarative query phase, sometimes it is the right thing to do. Again, as we develop more proficiency with the state-effect pattern, we will begin to learn more about the appropriate design choices in this case.

6 Related Work

There has been some work in the academic community on special purpose scripting languages for games, such as the ScriptEase language [McNaughton et al. 2004]. However, much of this work has focused on tools that make design accessible to inexperienced programmers. The work in this paper is different in that our scripting language is developed for the sole purpose of processing complex game behavior efficiently. While there are other special purpose languages designed for improving game performance, such as Simbionic [Fu et al. 2003] or Naughty Dog's GOAL [Liebgold 2008], our approach is the first one to leverage database processing techniques.

Some of the work presented in this paper, particular the optimizations in Section 4, have been presented previously in the context of early SGL prototypes [White et al. 2007]. These prototypes supported only one type of game object and could not perform iteration without some working knowledge of SQL. The purpose of this paper has been present a much more advanced version of SGL that does not require SQL. In addition, our aim has been to make this material accessible to a non-database audience, as well as to discuss some of the issues that arise when attempting to use our design pattern for developing games.

7 Conclusions

The state-effect pattern is a very powerful technique for efficiently scaling the interactions between game objects. It allows game designers to use a design pattern already familiar to them, albeit informally, to leverage decades of research in declarative processing and optimization. It removes the burden of designing custom pipelines and optimization, as this is all handled by the compiler, and allows the developer to focus more on the design process.

In addition to the obvious research opportunities in game-specific database processing, the state-effect pattern offers many interesting research opportunities in design. SGL is still a fairly simple scripting language because the authors have focused their research on formalizing the semantics and optimizing the query phase [White et al. 2007; Albright et al. 2008]. Higher level design tools would make it easier to visualize the design process and leverage this design pattern. The design of effects and their combination rules are interesting even by themselves. As we showed in Section 5.1, we can use them to enforce semantic constraints like "each loot object has one owner". Since cheats in MMOs are often some violation of semantic constraints, we would like to develop tools that allow game designers to specify other types of constraints.

Declarative processing in games presents an excellent opportunity for database researchers and the game designers to collaborate with one another. We believe that continued work in this area will greatly enrich both communities, and allow us to create more complex and immersive worlds.

Acknowledgements

This work is supported by the National Science Foundation under Grant IIS-0725260, the Air Force under Grant FA9550-07-1-0437, and a grant from Microsoft Corporation. Any opinions, findings, conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of the sponsors.

References

ALBRIGHT, R., DEMERS, A., GEHRKE, J., GUPTA, N., LEE, H., KEILTY, R., SADOWSKI, G., SOWELL, B., AND WHITE, W. 2008. SGL: A scalable language for data-driven games (demonstration paper). In *Proc. SIGMOD*.

BOHANNON, P., LIEUWEN, D., RASTOGI, R., SILBERSCHATZ, A., SESHADRI, S., AND SUDARSHAN, S. 1997. The architecture of the Dalí main-memory storage manager. *Multimedia Tools Appl 4*, 2, 115–151.

BONCZ, P., AND KERSTEN, M. 1995. Monet: An impressionist sketch of an advanced database system. In *Proc. BIWIT*.

CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. 2002. Monitoring streams — a new class of data management applications. In *Proc. VLDB*.

DEMERS, A., GEHRKE, J., PANDA, B., RIEDEWALD, M., SHARMA, V., AND WHITE, W. 2007. Cayuga: A general purpose event monitoring system. In *CIDR*, 412–422.

DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H., AND RASMUSSEN, R. 1990. The gamma database machine project. *IEEE Trans. on Knowledge and Data Engineering 2*, 1, 44–62.

DUNKI, Q. 2008. Streaming open world pathfinding. In *Proc. GDC*.

ENSEMBLE STUDIOS. 2000. *Computer Player Strategy Builder Guide, AI Expert Documentation for Age of Empires II: The Age of Kings*. Ensemble Studios.

FU, D., HOULETTE, R., AND JENSEN, R. 2003. A visual environment for rapid behavior definition. In *Proc. Conference on Behavior Representation in Modeling and Simulation*.

GOVINDARAJU, N., LLOYD, B., WANG, W., LIN, M., AND MANOCHA, D. 2004. Fast computation of database operations using graphics processors. In *Proc. SIGMOD*.

KRUSZEWSKI, P., AND VAN LENT, M. 2007. Not just for combat training: Using game technology in non-kinetic urban simulations. In *Proc. Serious Game Summit, GDC*.

LIEBGOLD, D. 2008. Adventures in data compilation and scripting for uncharted: Drake's fortune. In *Proc. GDC*.

MCNAUGHTON, M., CUTUMISU, M., SZAFRON, D., SCHAEFFER, J., REDFORD, J., AND PARKER, D. 2004. ScriptEase: Generative design patterns for computer role-playing games. In *Proc. ACE*.

MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G. S., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*.

NGUYEN, H., Ed. 2007. *GPU Gems*, vol. 3. Addison-Wesley.

O'BRIEN, J., AND STOUT, B. 2007. Embodied agents in dynamic worlds. In *Proc. GDC*.

PIKE, R., DOWARD, S., GRIESEMER, R., AND QUINLAN, S. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming Journal 13*, 4, 227–204.

POSNIEWSKI, S. 2007. Massively modernized online: MMO technologies for next-gen and beyond. In *Proc. Austin GDC*.

REYNOLDS, C. 1999. Steering behaviors for autonomous characters. In *Proc. GDC*.

STONEBRAKER, M., KATZ, R., PATTERSON, D., AND OUSTERHOUT, J. 1988. The design of XPRS. In *Proc. VLDB*.

WHITE, W., DEMERS, A., KOCH, C., GEHRKE, J., AND RAJAGOPALAN, R. 2007. Scaling games to epic proportions. In *Proc. SIGMOD*, 31–42.

ZHU, Y., RUNDENSTEINER, E., AND HEINEMAN, G. 2004. Dynamic plan migration for continuous queries over data streams. In *Proc. SIGMOD*.