Scaling Games to Epic Proportions

Walker White
Cornell University
Ithaca, NY 14853, USA
wmwhite@cs.cornell.edu

Alan Demers
Cornell University
Ithaca, NY 14853, USA
ademers@cs.cornell.edu

Christoph Koch*
Saarland University
Saarbrücken, Germany
koch@infosys.uni-sb.de

Johannes Gehrke†
Cornell University
Ithaca, NY 14853, USA
johannes@cs.cornell.edu

Rajmohan Rajagopalan
Cornell University
Ithaca, NY 14853, USA
mohan@cs.cornell.edu

ABSTRACT

We introduce scalability for computer games as the next frontier for techniques from data management. A very important aspect of computer games is the artificial intelligence (AI) of non-player characters. To create interesting AI in games today, developers or players have to create complex, dynamic behavior for a very small number of characters, but neither the game engines nor the style of AI programming enables intelligent behavior that scales to a very large number of non-player characters.

In this paper we make a first step towards truly scalable AI in computer games by modeling game AI as a data management problem. We present a highly expressive scripting language SGL that provides game designers and players with a data-driven AI scheme for customizing behavior for individual non-player characters. We use sophisticated query processing and indexing techniques to efficiently execute large numbers of SGL scripts, thus providing a framework for games with a truly epic number of non-player characters. Experiments show the efficacy of our solutions.

Categories and Subject Descriptors

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

General Terms

Languages, Processing

Keywords

Games, Scripting, Aggregates, Indexing

1. INTRODUCTION

Computer games are becoming the next frontier for social interaction between humans. The Entertainment Software Association estimates that computer and video game software sales in 2005 were $7.0 billion dollars [3]. While graphics have always motivated the growth of the game industry, we believe that the database community also has the opportunity to make significant contributions to this field.

A computer game is a virtual environment where players interact with digital objects or each other for entertainment. One of the keys to developing rich playing experiences is the creation of complex and interesting artificial intelligence (AI). In game development, AI has a slightly different meaning than it does in the academic context. Game AI is the system that controls the behavior of non-player characters (NPCs) — entities created by the game designer and controlled by the computer. While this system may use classic AI algorithms, game AI includes all routines that control behavior, be they intelligent or not.

Broadly speaking, there are two approaches to improving game AI. The first is to create complicated, detailed, dynamic behavior for a few particularly important NPCs, like the player's archnemesis or sidekick. This approach is ideal for games that do not have many NPCs in need of interesting behavior. For this type of behavior, classic AI is relevant, and has been employed to various degrees in existing games [14]. Expert systems have been used for choosing plays in sports games; natural language processing has been used for character interaction in *Façade* [17]; machine learning has been used for creature behavior in *Black & White* [12].

However, these techniques are often too computationally expensive or labor intensive to be practical for more than a handful of NPCs. Increasingly, having just a few intriguing NPCs is insufficient for many categories of games. Strategy games, massively multiplayer online games, and open world games all frequently require large numbers of interesting characters. Hence, the second approach to game AI is to enable interesting but relatively simple behavior for a large number of NPCs. For example, character behavior may be controlled by a simple finite state machine. In the aggregate, even simple game AI can lead to complex emergent behavior, so populating a game world with many NPCs can create compelling gameplay. However, there is a trade-off between having complex NPCs and having many NPCs. When the game demands too many NPCs, developers may have no choice but to employ simple game AI. But if the AI is too simple, the game will exhibit predictable uniformity. This trade-off is not addressed by the classic areas of research in artificial intelligence.

A further complication in creating large numbers of NPCs is the actual design of the AI for each NPC. Even if the processing power is available, creating AI is very labor intensive. To solve the problem of content creation, developers employ the use of *data-driven AI*. In this paradigm, the AI system is heavily parameterized by data files stored outside the code. In the simplest case, these parameters may be numerical values affecting transitions in state machines. However, more generally, they are scripts that are read and processed by the game's AI engine. This approach works for de-

*Work done while visiting Cornell University.

†Supported by a Sloan Foundation Fellowship.

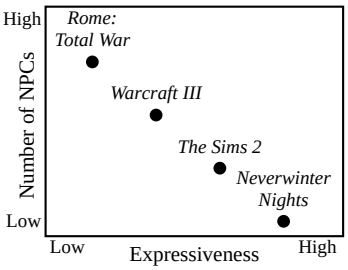

Figure 1: Expressiveness versus Number of NPCs

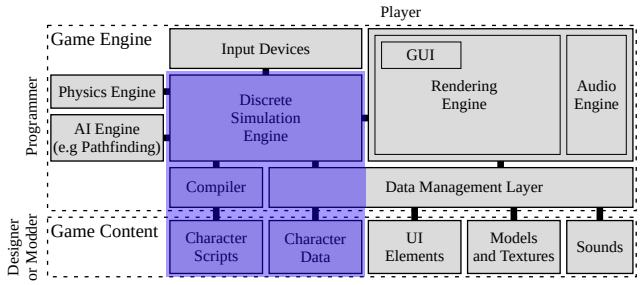

Figure 2: Data-Driven Game System Architecture

signing large numbers of NPCs because these scripts are simple but flexible enough to be adapted to many kinds of characters. In addition, a data-driven AI scheme offloads much of the burden of creating AI from the programmers to the game designers, allowing the game AI to be modified rapidly without recompiling.

The ability to produce interesting data-driven AI depends on the expressive power of these scripts. The scripting languages used in games vary widely, and are often customized to meet the specific needs of a game. Generally, the more expressive the scripting language, the smaller the number of NPCs that can be processed at any given time. Figure 1 illustrates this trade-off in existing data-driven games. *Neverwinter Nights*, with its versatile Aurora [18] scripting engine, supports intricate behaviors but only for a handful of units. *The Sims 2*, with a more restrictive system for its characters, can support a few dozen Sims pursuing their lives' objectives [24]. *Warcraft III* can support a couple hundred units, but it only allows relatively simple battle decisions for each unit [2]. Finally, *Rome: Total War* supports thousands of soldiers, but its system is extremely limited, as large groups of soldiers must have identical behavior [23]. If the expressiveness of *Neverwinter Nights* were possible in a game on the scale of *Rome: Total War*, this would provide new opportunities for gameplay not currently possible.

1.1 Scaling Data-Driven AI

Our goal is to create a data-driven AI system that is both highly expressive and capable of supporting large numbers of NPCs. As the number of NPCs with distinct behaviors grows, the maximal complexity of those NPCs must decrease to maintain performance. However, when large numbers of NPCs are making individual decisions, they may be acting on distinct but very similar sets of information. By treating game AI as a data management problem, we can leverage this fact to dramatically boost performance. In particular, we have developed a new functional scripting language, SGL (Scalable Games Language), that allows us to analyze scripts written by users and to use query rewrite techniques from the database community to factor out expensive function evaluation that is common to a large number of scripts. We use sophisticated query processing techniques to pre-compute the results of these expensive functions and we use indexing techniques to quickly access them within the scripts. This novel type of multi-query optimization in game AI significantly improves performance of the execution of the scripts.

The language itself is functionally quite similar to those used in existing games, and accessible to game designers. The AI system is designed to fit with typical game architecture without disrupting the usual structure of the other systems. Fundamentally, it is simply an optimization and generalization of data-driven AI schemes already employed by developers, and integrating it should be natural for both programmers and designers. As a result, designers are able to add more and more intelligent NPCs to their games while retaining the same development processes.

Contributions and Outline of the Paper

In Section 2, we start by describing the basic architecture for data-driven games, and identifying those subsystems for which data management is most important. After this introduction, we make the following contributions in this paper.

- In Section 3, we introduce a specific data management problem that must be solved in order to scale games to large numbers of interesting NPCs. We also present a case study using real-time strategy (RTS) games to analyze the effectiveness of our approach.
- In Section 4, we describe SGL, a novel and expressive scripting language for game AI. We demonstrate that the semantics of SGL allows us to process non-player characters set-at-a-time instead of individually.
- In Section 5, we show how to optimize SGL with rewrite rules and specific query plans. We also present several index structures to efficiently compute a large class of aggregate functions used in games.
- In Section 6 we present an experimental evaluation demonstrating the effectiveness of our query optimization and query processing techniques.

We discuss related work in Section 7 and conclude in Section 8.

2. DATA-DRIVEN GAMES

Loosely defined, a data-driven game is any game that separates the game content from the game code [10]. This design has several advantages. It allows the game studio to separate development between the programmers and the game designers, two groups with important but not necessarily overlapping skills.

Historically, games have long had some form of separation between content and code. Media such as character models, textures, or sounds are often kept in data files separate from the game engine. However, recently, the trend has been to move as much game content as possible out of the engine. The data used to define the characters or story-line is increasingly being stored in XML files [25]. Modern design even separates logic specific to the game play from the code, through the use of scripting languages; these scripting languages may either be custom tailored to the game engine, or a standard language like stackless Python [11, 8].

2.1 Architecture of Data-Driven Games

While data-driven games may have different architectures depending on their genre, they all have roughly the same design. Figure 2 represents the architecture of a typical data-driven game. Three different groups of actors interact with this system. The largest group of actors are the game players. They primarily interact with the game through the input and display devices.

The next group are the game programmers, who have designed the "game engine". The engine is not specific to any one game, and

can be reused for other games. In some cases, like Epic Games' *Unreal Engine 3* [13], the engine may even be licensed to other companies for development. The engine consists of several different generic components common to all games. The rendering and audio engines comprise the media experience of the game. The physics engine is a library of algorithms that simulate physical effects like gravity and collisions. The AI engine is a library of algorithms for solving classical AI problems like pathfinding or natural language processing. All of these are connected together through the discrete simulation engine. This part of the game controls actions of the characters and objects, instructing the rendering and audio engines how to generate output. The discrete simulation engine takes cues from the physics and AI engines, but it is largely directed by the content of the game.

The game content is created by the game designers. The designers are responsible for creating the game world. This includes a lot of the artistic elements like character models and sounds. However, it also includes any game specific logic. The character objects are stored in data files outside of the game engine. The behavior of these character objects is defined by the character scripts. These scripts are read by either a compiler or an interpreter, and processed by the discrete simulation engine.

This separation is particularly important for game AI, as character behavior must be constantly adjusted during game testing for reasons of "game balance" (i.e. ensuring that there is no single optimal strategy, so that game play does not become monotonous). For example, seven years after its first publication, Blizzard continues to update *Starcraft* with balance changes based upon observations of games played on their BattleNet server [16].

This separation is also important to players, as they can also interact with the content as game "modders". A modder is player who modifies a commercially released game to create a game variant. The new and challenging experiences provided by modders can often extend the lifetime of a game. For example, both *Starcraft* and the subsequent 2002 strategy game *Warcraft III*[1] store their game data and AI as scripts in MPQ files, a proprietary compression format that is similar to ZIP. The AMAI project [2] has tools for extracting the scripts from these MPQs and replacing them with new scripts to improve the combat AI in *Warcraft III*.

Games like *Neverwinter Nights* and *Second-Life* show that user-created content is a vibrant, growing aspect of the gaming world [21]. Therefore, we believe that the boundary between players and modders is closing, as more and more games embrace the idea of user-created content.

While the entire data management layer is an interesting area of research for the database community, our primary focus is the discrete simulation engine; before games can render large numbers of characters, they first must process their behavior. *The Sims 2* is an example of a game whose performance is determined primarily by the simulation bottleneck. A character in a room with a large number of objects can slow the game down perceptibly, even if the screen is not rendering the room; this is because the game is querying each of the objects in the room to determine which one currently satisfies the character's needs. This performance problem is so significant that the console version of *The Sims 2* introduced a "feng shui meter" as a gameplay element to keep a player from adding too many objects to a room [1].

2.2 The Discrete Simulation Engine

Almost all computer games are architected so that the AI engine processes its objects in clock ticks [20]. In *turn-based* games, these ticks are controlled by player input; the game will not proceed to the next tick until the player ends his or her turn. In *real-time* games, these ticks are controlled entirely by the game, and progress proportional to the frame-rate of the graphics engine.

Each clock tick, the simulation engine processes the actions of one or more characters. Each character can perform at most one action per tick, but since we want the number of NPCs to be determined by the data and not the game engine, our architecture should allow more than one unit to act per clock tick. A particular action may span more than an a single clock tick, as the game takes time to render the action. However, this is modeled by performing the action in a single tick, and assigning the character a "cooldown" period until it can act again. As a result, some characters may be inactive during a clock tick, as they are still in the cooldown period from their last action. Our model will assume that those characters just perform an empty action. Therefore, on each clock tick, we process exactly one action for every character in the game. As we show in Section 5, characters performing the empty action are eliminated by a selection operation, and so this assumption will not have any adverse effect on performance, given the appropriate index structures.

Each action, in turn, may produce several *effects*. An effect is simply an update to the data which defines an object. For example, movement is an action that has a single effect – it alters the position of that unit. On the other hand, mortar-fire in a combat game is an action that may affect several units, damaging every NPC in its blast radius.

At each clock tick, the simulation engine reads the data, determines the actions of each of the characters, and determines the effects of this actions, and then updates the game data for the next tick. It is traditional practice in game design that when multiple characters act during a clock tick, they act simultaneously. This keeps the engine from having to read the data more than once during a clock tick, as no action can depend on the action of another character in the same tick. It also allows us to cleanly separate each clock tick into three stages:

- A query stage, where we read the contents of the game data.
- A decision stage, where we choose the actions of each NPC based on the data read.
- An update stage, where we update the game data according to the effect of these actions.

Since the actions are all updating the game data simultaneously, we need a transaction model for how these updates are processed. In games, this is relatively easy, because effects typically increment or decrement numerical values in the character data. For example, damage decrements an NPC's health value, while healing increments it. Games additionally separate effects into *stackable* and *nonstackable*. In stackable effects, like damage, all of the effects for that tick are cumulative. For nonstackable effects, only one effect of that kind can apply – typically the most beneficial (or disadvantageous, depending on the context). In *Warcraft III*, witch doctors can create healing wards that heal all units in a certain range; this is a nonstackable effect as a unit in range of two wards is only healed once.

This design in games makes the update phase straightforward. We just combine the effects of all the actions, using sum for stackable effects and $\max$ for the nonstackable ones, and increment or decrement the data values accordingly. In a few instances, an effect may set some character data to an absolute value. For example, a freeze spell may set a character's speed to 0. In these instances, the effect is given a priority. Thus they are nonstackable effects determined by maximum priority.

[1] *Warcraft III* was the inspiration for the famous massively multiplayer online game *World of Warcraft*, but is not the same game.

3. INCREASING EXPRESSIVENESS IN REAL-TIME STRATEGY GAMES

One of the challenges with trying to increase expressiveness in game AI is that it must have a perceptible (positive) effect on the gameplay. While we believe that our approach will apply to all simulation games (like *The Sims 2*), as demonstrated by the number of units in *Rome: Total War*, real-time strategy (RTS) games are the ideal genre to scale to large numbers of characters[2]. In these games, a player does not control a single character, but instead controls armies of characters, which are called *units*. The player controls units by selecting them and issuing commands, which they then execute. However, the way in which a unit executes a command is controlled by the game AI. For example, if a human player instructs a character to attack a specific enemy unit, the game AI may first instruct it to attack other nearby enemy units just so that it can maneuver into range. Most of the gameplay consists of issuing a command to a unit, and then scrolling to another portion of the map to command other units, while the first unit executes its orders. Thus these games can scale by orders of magnitude without advances in rendering technology.

Because of this gameplay, RTS games should ideally have scripts defining the behavior of each individual. A player wants a unit to execute its command correctly without further instruction; that way the player can issue commands to large numbers of units, effectively controlling massive armies. However, unit behavior in RTS games is relatively primitive; they are typically modeled as simple finite state machines [22]. As a result a player must directly control the units if there is to be any coordination between units. For example, a standard tactic in strategy games is to have archers stay behind armored troops in order to protect them; if the armored troops move, the archers need to move as well to retain their cover. Even achieving this relatively simple level of coordination requires the human player to neglect all other troops and repeatedly issue instructions to these two.

The problem is that processing individual AI scripts can be *very expensive* as each unit is typically processed separately. Game AI is a main efficiency bottleneck in such games. Suppose the game designer wants a certain type of unit to run in fear from a large number of marching skeletons. If the number of skeletal troops is on the order of n, the total number of units, then it takes $O(n)$ to count the number of skeletons. Furthermore, if all the units can see the skeletons, then each unit performs an $O(n)$ count aggregate, for a total time of $O(n^2)$ to process all of the units.

3.1 Processing Units as a Group

The typical solution to this problem in RTS games is to handle all coordination in *centralized AI scripts*. In centralized AI, a script controls the actions of a large number of units. For example, each computer player in *Warcraft III* has two invisible commanders to control all the units: one for attacking, and one for defense. Centralized AI controls units by querying the environment, and then issuing a simple command to each unit. This solves the problem in our skeleton example since the centralized script can count the number of skeletal troops in $O(n)$ time and issue the "run away" instruction to each unit again in $O(n)$ time.

However, centralized AI has three major problems. Because one script controls all of the units of a faction, it is difficult to write scripts that control more than one geographic cluster of units at a time. The limitation of *Warcraft III* to two commanders means the computer is unable to defend and fight a multiple-front war at the same time; human players use limitations like this to their advantage. Another problem is that it is difficult to separate individual behavior from herd behavior. When the centralized AI script sees the skeletal warrior, it issues the run away command to all units. Thus the units flee uniformly, ignoring issues such as which units can see the skeletons. Changing the centralized script to account for this makes the script harder to design and read.

Most importantly, however, centralized AI is really only designed to run the computer player. It is of no help to the human player because he or she controls individual units, and not a central commander. Therefore, sophisticated individual AI scripts would be a massive improvement to RTS games.

Note that centralized AI is a crude form of set-at-a-time processing, explicitly implemented by the game designer. The designer knows that all of the units will compute the same aggregate and places this in the centralized script. However, it should not be necessary for the game designer to do this explicitly. If we construct a scripting language that allows us to use sophisticated rewrite rules to group calculations together, then we can do this in the script compiler.

The primary difficulty in designing such a query language is iteration; if the language only has conditionals, we can easily convert our language to a declarative language like SQL and optimize it accordingly. Fortunately, an analysis of the scripts in RTS games like *Warcraft III* [2] reveals that iteration is only used in the following contexts.

- Computing an aggregate value about a set of units or the local environment. Examples include summing up the strength of visible units, or finding the weakest unit in range.
- Applying an update to a set of units or the environment.
- Processing an array whose size is fixed and determined at compile time (e.g. an array representing the "strength" of each troop type in *Warcraft III*).
- Reimplementing functionality that exists already in the game, but is not open to modders (e.g. the pathfinding algorithms in the AMAI file `common.eai` [2]).

The first two cases can easily be handled by a declarative language. The third case is also either an aggregate computation or an update to the array, and can be processed similarly. The final case is simply a matter of opening up more of the API to the scripting language, which is an orthogonal problem. Therefore, we can get the most important functionality of these scripting languages with a purely functional language with aggregate functions on sets. At each step, the AI script performs a declarative query on the environment and uses the result to perform an update. We define this language explicitly in Section 4.

To provide true individualized behavior, it is not enough that our optimization pull out common aggregate expressions. For example, the units counting the number of skeletons may not be able to see exactly the same number of skeletons. However, if the units are clustered together – as they normally are in combat – the skeletons they see should overlap. To take advantage of this overlap, we would like to construct indices that efficiently compute the number of skeletons for each visible region, and process each script as a look-up in this index. However, the type of index that we make depends heavily on the type of aggregate and our query plan. We investigate this further in Section 5.

3.2 Case Study: A Battle Simulation

RTS games have non-combat aspects to them such as economics and building. However, in these games these aspects are highly-abstracted and do not feature large numbers of individuals. Therefore they are relatively easy to process. Hence, we will evaluate our approach by focusing on the combat simulation of an RTS.

[2]While massively multiplayer online games have more characters, relatively few interact with each other at any moment.

Our battle simulation is structured like that of the typical RTS. The state of each unit consists of at least three values: the x and y position of the unit, and its health. Health is modeled as an integer; when it is reduced to 0, the unit is dead and is removed. There are only three types of actions: a unit can either move (to change its x and y value), damage an enemy unit (reducing its health), or heal a friendly unit (restoring its health). Which of these actions are available depends on the type of the unit.

- **Knights**: These units can only move and attack. They are armored, and hence take less damage from the attacks of others. They also do the most damage in their attacks. However, they can only attack units that are in arm's reach.
- **Archers**: These units can only move and attack. Unlike knights, they are not armored, so they take more damage from the attacks of others. Their arrows also do less damage than the swords of the knights. However, they have a much larger range in which they damage an enemy unit.
- **Healers**: These units can only move and heal. Like archers, they are not armored, and so take more damage from the attacks of others. They heal units by casting a "healing aura" that restores health to all friendly units within the circle of this aura. The health of a unit can never be restored beyond the initial health of the unit. Healing auras are nonstackable, so a unit can only be healed once per clock tick.

For modeling specifics such as determining damage, the effects of armor, and so on, we use the game mechanics in the pen-and-paper d20 system[27]. This system is the foundation for all computer game combat simulations, and thus is a reasonable model. This system has the added advantage that its rules are not designed according to the limitations of computer games. In *Warcraft III*, a typical unit can only see an area capable of holding 100 other units. Therefore, processing a query like "count the number of skeletal units" is really just $O(1)$ with a large constant. On the other hand, visibility in the d20 system allows characters to see and make judgments about areas containing up to 25,000 other units. Thus these mechanics allow for interesting scaling to large numbers of units.

In our case study, we want our scripting language to support interesting coordination between units. For example, we want the archers to use the knights as cover. To do this, the scripts compute the centroids of the enemy, the knights, and the archers, and moves the archers so that these three points are in a line with the knights in the center. As another example, we want the knights to close ranks to keep the enemies from going through. To do this the knights compute their approximate density by computing the standard deviation of all the troop positions, and then counting the number of troops in two standard deviations. If they are too spread out, they move towards their centroid.

In general, our scripting language will support a much larger class of aggregates than these examples. However, they are enough to exhibit interesting behavior not found in current RTS games. Furthermore, they will serve as useful examples when we define our language in the next section.

4. THE SGL LANGUAGE

Our game data is abstractly modeled as a relation E. We assume that this table is a *multiset*; it need not have keys. Each row in the table represents a unit or object, and contains information such as the unit's health, speed, attack damage, and so on. It may also include data representing messages from other units of the system, like the pathfinding subsystem, or the time remaining in the unit's cooldown period.

The language SGL (Scalable Gaming Language) is a scripting language for specifying individual unit behavior. An SGL script represents a single action for a single unit. Informally, an SGL script is a function that, at each clock tick, takes the environment E and returns a new environment table E_u. However, since there are several individuals acting, we need to be able to combine the environments E_u to produce the final environment at the end of our clock tick.

We do this by separating the schema of E into attributes representing the state of the each unit and the attributes representing the effects on the unit. For example, one possible schema for our battle simulation is

```
E(key,player,posx,posy,health,cooldown,
  weaponused,movevect_x,movevect_y,          (1)
  damage,inaura)
```

The attributes `key` ... `cooldown` in (1) represent the state of the unit. These attributes cannot be modified directly by an SGL script. The remaining (auxiliary) attributes represent the effects applied to the unit, such as how far the unit will move, or the strength of the nearest healing aura. These are the values altered by an SGL script; we combine these values together to calculate the final effect on each unit using the rules outlined in Section 2.2.

Only once we have combined all of the individual environments E_u together into a single environment do we actually apply the effects and change the state of the units. This is done by a post-processing step outside of the SGL scripts, and is considered as part of the game mechanics.

EXAMPLE 4.1. For the schema in (1) the post-processing step consists of performing the following SQL query to get the new environment.

```
SELECT u.key, u.player,
    u.posx + u.movevect_x * norm AS posx,
    u.posy + u.movevect_y * norm AS posy,
    u.health - u.damage + u.inaura AS health,
    u.cooldown - 1
      + u.weaponused*_TIME_RELOAD AS cooldown,
    0 AS weaponused,
    0 AS movevect_x, 0 AS movevect_y,
    0 AS damage, 0 AS inaura
FROM E u
WHERE u.health > 0;  # remove the dead
```

where `norm` is a shortcut for `_WALK_DIST_PER_TICK / sqrt(u.movevect_x^2 + u.movevect_y^2)`. For example, at the end of the tick, we take the total damage done to a unit and subtract it from the health (as well as restore the amount provided by the healing aura). It is also at this point that we remove units with 0 health from the table.

We spend the rest of this section formalizing this processing model so that we can optimize it in Section 5.

4.1 Syntax of SGL

Informally, SGL scripts consist of SQL together with conditionals (`if-then-else` statements), `let`-statements to (temporarily) add new attributes to the current unit, and a special keyword `perform` for invoking other scripts or applying built-in actions. A `perform` statement specifies an update to the environment. To help with readability, the programmer can decompose a script into several functions.

Because individual unit behavior must be tailored to the unit, each AI script has access to the current unit tuple `u` (which holds its own state) from the environment. Furthermore, it has a function $Random$ for generating random values. To get a random number, the script provides a number as a seed. For any number i, $Random(i)$ will always return the same number within a single clock tick, but not necessarily between clock ticks.

```
main(u) {
  (let c = CountEnemiesInRange(u,u.range))
  (let away_vector = (u.posx, u.posy) -
           CentroidOfEnemyUnits(u, u.range)) {
    if (c > u.morale) then
      perform MoveInDirection(u,away_vector);
    else if (c > 0 and u.cooldown = 0) then
      (let target_key = getNearestEnemy(u).key) {
        perform FireAt(u, target_key);
  } } }
```

Figure 3: An SGL Script

In detail, the syntax of action functions is given by the grammar

```
action ::= (let attributename = term) action
         | action; action
         | if cond then action
         | if cond then action else action
         | perform actionfn_name
```

Conditions are Boolean combinations of atomic conditions. Atomic conditions are comparisons of two terms (using $=$, $<$, $\leq$, $\neq$). Terms are constructed using arithmetics over constants, attributes of the unit, random numbers, and aggregate functions.

EXAMPLE 4.2. Figure 3 shows an example of a simple script that fires an arrow if there is a unit in range, but runs away if there are too many enemies. If neither case is true, or it is waiting on the weapon cooldown, then it does nothing. Note that `CountEnemiesInRange`, `CentroidOfEnemyUnits`, and `NearestEnemy` are all aggregate functions that compute a value from E. The functions `MoveInDirection` and `FireAt`, on the other hand, are action functions and update the environment. These functions are provided as SGL built-ins, but we show how to define them explicitly in Section 4.3.

4.2 Combining Effects in Environment Tables

As we described in Section 2.2, the way in which we combine effects depends on whether they are stackable or nonstackable. Therefore, we tag the attributes of our environment E to keep track of how we combine effects on this attribute (i.e. sum for stackable effects, $\min$ or $\max$ for nonstackable effects). Formally, our environment E has schema $E(K, A_1, \ldots, A_k)$. Each attribute A_i of E is tagged as the type τ_i which is either const, $\max$, $\min$, or sum. Attributes of type "const" never change and can never be the direct subject of an effect; the type of K is always const. For example, in the schema in (1), the first line of attributes are all of type const. The attribute `inaura` has type $\max$, since healing auras are not stackable; all other attributes have type sum.

To combine output of the SGL scripts, we define a *combination operation* $\oplus$ on a relation R whose schema $R(K, A_{i_1}, \ldots, A_{i_m})$ is a subschema of that of E. We let $K, A_{i_1}, \ldots, A_{i_l}$ be precisely the const-typed attributes of R. We define $\oplus R$ as

$$\texttt{select } K, \ f_{i_1}(A_{i_1}) \texttt{ as } A_{i_1}, \ldots, f_{i_m}(A_{i_m}) \texttt{ as } A_{i_m} \\ \texttt{from } R \texttt{ group by } K, A_{i_1}, \ldots, A_{i_l};$$

where, abusing notation and identifying type τ_j with the aggregate function of the same name,

$$f_j(A_J) := \begin{cases} A_j & \ldots \quad \tau_j = \text{const} \\ \tau_j(A_j) & \ldots \quad \text{otherwise} \end{cases} \tag{2}$$

When attribute K is a key for table R, we will sometimes write $R^{\oplus}$ to highlight this; note that in this case $R = \oplus R$. We use $R \oplus S$ as a shortcut for $\oplus(R \uplus S)$, where $\uplus$ denotes the multiset union operation.

Because $\oplus$ is defined in terms of $\min$, $\max$, and sum, it is associative and commutative. Furthermore, given two environment tables E_1 and E_2,

$$\oplus(E_1 \uplus E_2) = \oplus(\oplus(E_1) \uplus E_2). \tag{3}$$

In the case $E_2 = \emptyset$, this in particular implies idempotence of the combination operator, $\oplus(\oplus(E_1)) = \oplus(E_1)$, and by applying the equivalence twice we obtain $\oplus(E_1 \uplus E_2) = \oplus(\oplus(E_1) \uplus \oplus(E_2))$. This property will be useful in generating our query plans.

EXAMPLE 4.3. If the schema in (1) is tag with types as described above, then

```
SELECT key, player, posx, posy, health, cooldown,
       max(weaponused) AS weaponused,
       sum(movevect_x) AS movevect_x,
       sum(movevect_y) AS movevect_y,
       sum(damage) as damage,
       max(inaura) as inaura
FROM E
GROUP BY key, player, posx, posy, health, cooldown
```

computes the environment $\oplus E$.

4.3 Semantics of SGL

The goal of the SGL language is to support the specification of character actions. It is intended to be expressive, but to have a simple semantics that can be easily mapped to query evaluation techniques nevertheless. For this reason it is a functional language with a somewhat imperative surface syntax. Each expression in our language is called an *action function.*

An action function is a function of signature

$$f : \text{Env} \times \text{Multiset}(\text{Env}) \times 2^{\text{Env} \times \mathbb{N} \to \mathbb{N}} \to \text{Multiset}(\text{Env})$$

for some constant c. Thus an action function takes

- a tuple from the environment table (the current unit),
- the environment itself, and
- a function that maps any pair consisting of a tuple *from the environment* and a natural number to a natural number

as input and returns an updated environment table. The function $\text{Env} \times \mathbb{N} \to \mathbb{N}$ is used to simulate random numbers inside our functional language.

The semantics of SGL action functions is given by the semantics functions $[\![\cdot]\!]$, $[\![\cdot]\!]_{\text{cond}}$, $[\![\cdot]\!]_{\text{term}}$ for action functions, conditions, and terms, respectively. We define this semantics as follows

$$\begin{aligned} [\![\texttt{(let } v \texttt{ := } t\texttt{) } f]\!]_{E,r}(u) &:= [\![f]\!]_{E,r}(u, v : [\![t]\!]_{\text{term}}(u, E, r)) \\ [\![f_1; f_2]\!]_{E,r}(u) &:= [\![f_1]\!]_{E,r}(u) \oplus [\![f_2]\!]_{E,r}(u) \\ [\![\texttt{if } \phi \texttt{ then } f_1]\!]_{E,r}(u) &:= \begin{cases} [\![f_1]\!]_{E,r}(u) & \text{if } \phi_{E,r}(u) \\ \emptyset & \text{otherwise} \end{cases} \\ [\![\texttt{perform } G]\!]_{E,r}(u) &:= [\![g]\!]_{E,r}(u) \\ [\![\texttt{perform } H]\!]_{E,r}(u) &:= h(u, E, r) \end{aligned}$$

where f_1, f_2 and g are SGL action functions, G is the name of defined function g, H is the name of built-in action function h, and v is an attribute not yet present in record u. We will consider `if` ϕ `then` f_1 `else` f_2 a shortcut for `if` ϕ `then` f_1`; if` $\neg\phi$ `then` f_2. Note that our definition of `let` means that we extend the current unit record by value of term t.

The semantics of conditions $[\![\cdot]\!]_{\text{cond}}$ commutes with the Boolean operations, and $[\![\cdot]\!]_{\text{term}}$ commutes with the usual arithmetic operations. There are only three interesting types of terms. We define an attribute access in a tuple by $[\![u.v]\!]_{\text{term}}(u, E, r) = u.v$. Our random function is defined as $[\![Random(i)]\!]_{\text{term}}(u, E, r) = r(u, i)$; Finally, for an aggregate function name `a` we assume that

```
function CountEnemiesInRange(u, range) returns
  SELECT Count(*)
  FROM   E
  WHERE  E.x >= u.posx - range
  AND    E.x <= u.posx + range
  AND    E.y >= u.posy - range
  AND    E.y <= u.posy + range
  AND    E.player <> u.player;

function CentroidOfEnemyUnits(u, range) returns
  SELECT Avg(x) AS x, Avg(y) AS y
  FROM   E
  WHERE  E.x >= u.posx - range
  AND    E.x <= u.posx + range
  AND    E.y >= u.posy - range
  AND    E.y <= u.posy + range
  AND    E.player <> u.player;
```

Figure 4: Aggregate Function Definitions

there is an external function $a : (u, E, r) \mapsto \mathbb{R}^c$ and we define $[\![\mathsf{a}]\!](u, E, r) := a(u, E, r)$.

While the built-in aggregate and action functions could be arbitrary computable functions of appropriate signature, it appears that in practice it causes no loss of generality to assume that they are expressible in SQL, more specifically of the following form.[3]

- Each built-in action function $h(u, E, r)$ is of the form

$$\begin{aligned}&\texttt{SELECT e.K, } h_1(u,e,r) \texttt{ AS A}_1\texttt{, } \ldots\\ &\qquad\qquad h_k(u,e,r) \texttt{ AS A}_k\\ &\texttt{FROM } E\ e \texttt{ WHERE } \phi(u,e,r).\end{aligned} \tag{4}$$

- Each built-in aggregate function $a(u, E, r)$ is of the form

$$\begin{aligned}&\texttt{SELECT } a_1(h_1(u,e,r))\texttt{, } \ldots\texttt{, } a_k(h_k(u,e,r))\\ &\texttt{FROM } E\ e \texttt{ WHERE } \phi(u,e,r).\end{aligned} \tag{5}$$

Here the tuples u and r are assumed to hold constants, $h_1, \ldots, h_k$ are terms over u, e, and r, and $a_1, \ldots, a_k$ are SQL aggregates.

Figure 4 shows definitions of the aggregates used in the SGL script of Figure 3 and Figure 5 defines some built-in action functions, both using the SQL fragments indicated above.

To process a complete SGL script, each script has a main action function called MAIN. Given a function $\rho : E \to E \to \mathbb{N} \to \mathbb{N}^c$, the semantics of an SGL script is

$$\text{tick}(E, \rho) := \text{main}^{\oplus}_{\rho}(E) \oplus E \tag{6}$$

where, here and in the following, $f^{\oplus}_{\rho}(E)$ is a shortcut for

$$\oplus(\biguplus\{[\![f]\!]_{E,\rho(u)}(u) \mid u \in E\}). \tag{7}$$

The function ρ contributes the random element to the evaluation of the script. Note, however, that this formalization is completely deterministic. Since below we will only discuss the computation done within a single tick, we usually omit the subscript ρ and simply write $f^{\oplus}$. Further note that $f^{\oplus}$ is a unary relational operation.

Now that we have our formal definitions, we review our processing model once more. In a single tick, the processing model first initializes the auxiliary attributes introduced by the scripts. Then it produces the environment table E_u for each script, which encodes the effects but does not apply the effects. These all combined into a single table $\text{tick}(E, \rho)$. Finally, we apply effects using a special post-processing query defined by the game mechanics as shown in Example 4.1.

[3]For example, the Warcraft III `common.ai` can be fully expressed under these restrictions.

```
function FireAt(u, target_key)
returns
   SELECT e.key,e.player,e.posx,e.posy,e.health,
          e.cooldown, 1 AS weaponused,
          e.movevect_x, e.movevect_y,
          e.damage+(_ARROW_HIT_DAMAGE - _ARMOR) *
                   (Random(e,1) mod 2) as damage,
          e.inaura
   FROM   E e
   WHERE  e.key = target_key;

function MoveInDirection(u, x, y)
returns
   SELECT e.key,e.player,e.posx,e.posy,e.health,
          e.cooldown, e.weaponused,
          x - e.posx AS movevect_x,
          y - e.posy AS movevect_y,
          e.damage, e.inaura
   FROM E e
   WHERE e.key=u.key;

function Heal(u)
returns
   SELECT e.key,e.player,e.posx,e.posy,e.health,
          e.cooldown, e.weaponused,
          e.movevect_x, e.movevect_y, e.damage,
          nonsql_max(e.inaura,_HEAL_AURA)
                 AS inaura
   FROM   E e
   WHERE  u.player = e.player
   AND    abs(u.posx - e.posx) < _HEALER_RANGE
   AND    abs(u.posy - e.posy) < _HEALER_RANGE;
```

Figure 5: Action functions implemented in SQL

5. QUERY OPTIMIZATION

In this section we address the efficient processing of SGL scripts using data management techniques. We first show how SGL scripts can be translated in a natural way into a relational algebra-like language. Then we discuss the algebraic optimization of such queries and the determination of query plans including the use of indexes. Finally, we give efficient algorithms for computing index structures for aggregate functions and area-of-effect actions which in total improve the running time of tick() from time $O(n^2)$ to $O(n \log^d n)$, where d depends on the query plan.

5.1 Bag Algebra

We use a fragment of the relational algebra on multisets (using operations projection π, selection σ, product $\times$, and multiset union $\uplus$) extended by a *combination operation* $\oplus$. The multiset algebra operations are defined by a mapping to SQL:

$$\begin{aligned}\sigma_{\phi}(R) &:= \texttt{select * from R where } \phi\texttt{;}\\ \pi_{\vec{f}(*)\,\texttt{AS}\,\vec{B}}(R) &:= \texttt{SELECT } \vec{f}(*) \texttt{ AS } \vec{B} \texttt{ FROM R;}\\ R \times S &:= \texttt{SELECT * FROM R, S;}\\ R \uplus S &:= \texttt{R UNION S;}\\ \text{agg}_{\vec{A},\,\vec{g}(\vec{B})}(R) &:= \texttt{SELECT } \vec{A}\texttt{, } \vec{g}(\vec{B})\\ &\qquad \texttt{FROM R GROUP BY } \vec{A}\texttt{;}\end{aligned}$$

Here $\vec{f}(*)\,\texttt{as}\,\vec{B}$ stands for $f_1(*)\,\texttt{as}\,B_1, \ldots, f_n(*)\,\texttt{as}\,B_n$, where the f_i are terms built using the attributes of the input relation, constants, arithmetics, and external functions, and each g_i is an SQL aggregate function (e.g. `min`, `max`, `count`, `sum`, `avg`).

The natural join $\bowtie$ is defined in analogy to relational algebra using the above multiset operations. Below, we will use algebraic expressions interchangeably with SQL queries. We will only apply the natural join on pairs of relations whose schema overlaps on exactly the attribute K, and use the notation $\bowtie_K$ to make this clear.

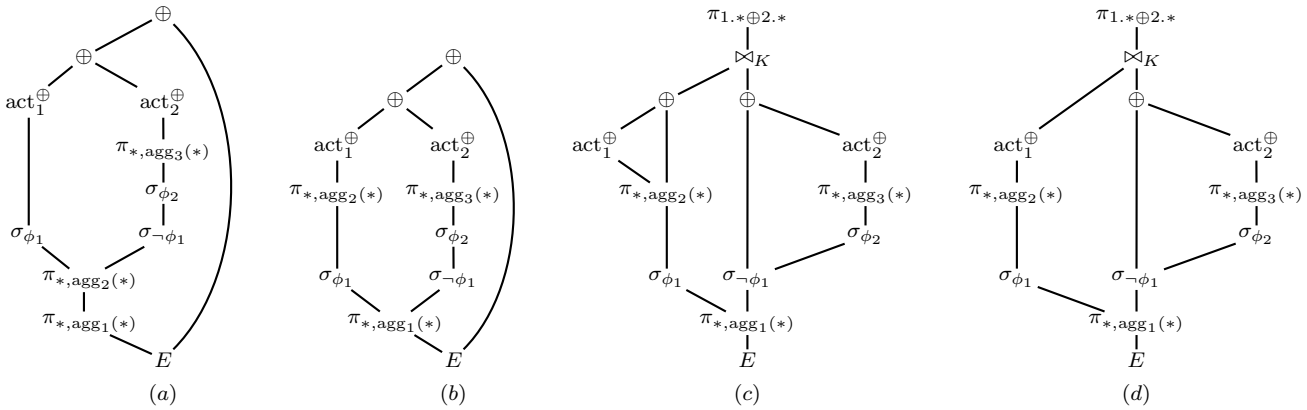

Figure 6: Example Query Plans. Example 5.1 translates (a) via (b) and (c) into (d).

$$g(\pi_{*,f(*)}(R)) \oplus R = g(R') \oplus R' \quad \text{where } R' = \pi_{*,f(*)}(R) \tag{8}$$

$$f(\sigma_{\phi}(R)) \oplus g(\sigma_{\neg\phi}(R)) \oplus R = (f(R') \oplus R') \oplus (g(R'') \oplus R'') \quad \text{where } R' = \sigma_{\phi}(R),\ R'' = \sigma_{\neg\phi}(R) \tag{9}$$

$$R_1^{\oplus} \oplus R_2^{\oplus} = \pi_{1.*\oplus 2.*}(R_1^{\oplus} \bowtie_K R_2^{\oplus}). \quad \text{where } \pi_K(R_1^{\oplus}) = \pi_K(R_2^{\oplus}) \tag{10}$$

Figure 7: Some rules involving $\oplus$. $R, R_1^{\oplus}, R_2^{\oplus}$ denotes extended environment relations.

We assume that SGL scripts are in a normal form in which aggregate functions only occur in let-statements, but in no other terms. It is easy to see that this is a normal form for SGL and that this assumption causes no loss of generality. For example, `if agg(u.health) = 3 then f` is equivalent to `(let v = agg(u.health)) if u.v = 3 then f`.

By the following translation, each SGL script can be turned into an equivalent expression in our algebra.

$$\begin{aligned} [\![f_1; f_2]\!]^{\oplus}(E) &= [\![f_1]\!]^{\oplus}(E) \oplus [\![f_2]\!]^{\oplus}(E) \\ [\![\texttt{if } \phi \texttt{ then } f]\!]^{\oplus}(E) &= [\![f]\!]^{\oplus}(\sigma_{\phi}(E)) \\ [\![(\texttt{let } \vec{A} = a)\ f]\!]^{\oplus}(E) &= [\![f]\!]^{\oplus}(\pi_{*,a(*) \text{ as } \vec{A}}(E)). \end{aligned}$$

These translations follow immediately from the SGL semantics definition. Using these equivalences, we can rewrite the overall semantics function tick() (see Eq. (6)) into an expression of our bag algebra. Note that extensions of the schema of an environment relation effected by "let" add untyped columns, which however are eliminated by the built-in action functions. These by definition always return environment relations of schema $K, A_1, \ldots, A_k$.

If the built-in action function $f_i(u, E, r)$ is in the fragment of SQL specified in Eq. (4), we can express $f_i^{\oplus}(R)$ in our algebra as

$$\oplus(\pi_{E.K,g_1,\ldots,g_k}(\sigma_{\psi_i}(R \times E))).$$

Each aggregate of the form of (5) applied to a set of units E_0 can be written in the bag algebra as

$$\pi_{u,f_a(u)}(E_0) \quad := \quad \text{agg}_{1.*,\vec{g}(2.*)}(\sigma_{\phi}(E_0^{(1)} \times E^{(2)})) \tag{11}$$

which can also be computed by an index nested loop join $E_0 \bowtie \text{Ind}_{\text{agg}}(E)$ with a precomputed index structure $\text{Ind}_{\text{agg}}(E) = \text{agg}_{1.*,\vec{g}(2.*)}(\sigma_{\phi}(E^{(1)} \times E^{(2)}))$. The efficient computation of such index structures is discussed in Section 5.3.

5.2 Algebraic Optimization

We can now rewrite the queries obtained from SGL scripts using the algebraic laws that hold in our algebra. These are to the greater part known from relational algebra[4], but some additional rules hold for $\oplus$ and its interaction with the other operations.

EXAMPLE 5.1. Consider the script of Figure 3. For clarity of exposition, the names of aggregation functions, built-in action functions, and conditions are abbreviated as $\text{agg}_1, \text{agg}_2, \text{agg}_3$, $\text{act}_1, \text{act}_2$, and ϕ_1, ϕ_2, respectively. (The ordering is as they appear in the script.)

By our rewrite rules that take SGL to our algebra, we obtain the query plan of Figure 6 (a). This query plan is actually already quite good. While the SGL script suggested an evaluation one unit at a time, the query plan employs set-at-a-time processing.

One optimization that we can achieve is to push $\pi_{*,\text{agg}_2(*)}$ up across the selections. In the right branch of the expression, agg_2 (in the form of the attribute `away_vector`) is not used and can be removed. The aggregate index for agg_2 will only have to be computed for the units that satisfy condition ϕ_1. We obtain the query plan of Figure 6 (b).

Next we optimize the combination of the result of main$^{\oplus}$ with E. This combination takes place to ensure that each unit in E is also present in the result even if no action is taken on this particular unit in the current tick. There are two actions being carried out, `MoveInDirection` or `FireAt`. The first modifies each of the units on which it is applied; for these units we do not need to combine with E.

This optimization can be effected as follows.

1. Using rules (8), (9), and (10), we can turn the plan of Figure 6 (b) into the plan of Figure 6 (c).
2. By definition $\text{act}_1^{\oplus}(R)$ is of the form

   $$\pi_{\vec{f}(E.*)}(R \bowtie_K E)$$

   which can be simplified to $\pi_{\vec{f}(*)}(R)$. But then

   $$\text{act}_1^{\oplus}(R) \oplus R = \text{act}_1^{\oplus}(R).$$

This yields the plan of Figure 6 (d).

[4]For the monotonic operations that we introduce – those that do not perform aggregation – the laws are basically the same as for relational algebra with set semantics.

5.3 Indexes and Geometric Algorithms

As we saw in Section 2.2, the most expensive part of a unit's script is often the processing of the aggregate functions. If every friendly unit is processing the aggregate to count the number of skeletal warriors, and all the enemy units are skeletal warriors, then the naive computation is $O(n^2)$. As we noted in the previous section, we can optimize this behavior by sharing the computation for agg across several units and processing $\pi_{*,\mathrm{agg}(*)}$ with an index nested loop join. Of course, to do this, we have to be able to construct the index for the aggregate.

Our choice of index structure does not just depend on agg. It also depends on the selection σ_ϕ; this selection appears outside the join in our index look-up optimization. For example, the index structure to count the number of skeletal warriors is not the same as the index structure to count the number of units belonging to the blue player.

In traditional databases, it would be prohibitively expensive to create indices for each individual query plan. However, SGL queries do not change rapidly over the course of the game; the player issues a command, and that command performs the same query for many clock ticks. In that regard, SGL queries are similar to continuous queries in streaming databases. We can afford to construct an index specifically tailored to each query plan.

Note that our indices are used to share computation between units, not between clock ticks. It is usually the case that the number of index probes *in each clock tick* is comparable to the number of entries in the index. Therefore, we are still likely to see significant performance gains even if, at each clock tick, we discard the index and build a new one from scratch. For data that is updated often – such as unit positions – it may even be more efficient to do this than to maintain a dynamic index.

In constructing our indices, we assume that ϕ is a conjunctive query. This is commonly the case in games and is evident in all of the aggregate queries in AMAI file `common.eai` [2]. Moreover, it is true for all of the examples in this paper. Given this assumption, we can ignore those conjuncts of ϕ that are not part of joins. For example, suppose we want to count the number of moderately wounded units (without regard to location). We typically define a unit `u` as moderately wounded if `u.health < 0.5*u.max_health`. This particular selection can be pushed into the index nested loop join, and so we do not have to consider it when building the index. On the other hand, if we want to count the number of visible enemy units, then determining whether an enemy unit is visible requires both the position of the enemy unit and the position of the unit performing the query. Thus this selection condition must be factored into the index.

Given that we have reduced ϕ to those conjuncts necessary for the join, we now present index structures for aggregates commonly found in games. These aggregates include all of the ones in our examples, as well as ones that appear in the scripts for *Warcraft III*.

5.3.1 Orthogonal Range Queries

The most common type of selection condition ϕ in a game script is an orthogonal range query. Conditions such as whether the unit can penetrate the armor of the enemy or can move faster than the enemy are inequalities comparing one value to another. For categorical data, this may be a degenerate range query, such as determining if a unit is of a certain type. Even determining if a unit is in range can be an orthogonal range query. For performance reasons, games often choose to use rectangles, not circles, to determine area of effect as is demonstrated in Figure 4. This is evident by the prevalence of functions in the AI scripts for *Warcraft III* that select units in a rectangle, like `GroupEnumUnitsInRect()`. Other games optimize by using circles with an L^1 norm; however, these are just squares rotated 45° and so they can be modeled as orthogonal range queries as well.

In the case where all of ϕ is an orthogonal range query, we can process it with a layered range tree [9]. We order the levels of the layered range tree according to the volatility of each axis. Attributes that do not change often, such as the type of the unit or its maximum health, form the top layer of the index, while data that is constantly updated, such as position, is at the bottom. This way we can reuse as much of the index as possible across clock-ticks. In particular, we can preserve the upper layers that do not change, but dispose of the lower ones, which do.

We can build a layered range tree in $O(n \log^d n)$ time, and for each unit, we can enumerate those elements that satisfy ϕ in $O(\log^d n + k)$ time, where d is the dimension of the orthogonal range query and k is the number of elements selected by ϕ. In determining the dimension d, we can ignore all degenerate (i.e. categorical) range components, as those levels of the tree can be replaced by a hashtable with $O(1)$ look-up. As we mentioned above, it is not necessary for this index to be dynamic (see [7] for the additional cost of dynamic algorithms). Therefore, we can use fractional cascading [6] to reduce the time to $O(n \log^{d-1} n)$ and $O(\log^{d-1} n + k)$, respectively.

However, a layered range index by itself still does not give us the performance we want. If the units are all clustered together, as is often the case in combat, then the value k in $O(\log^{d-1} n + k)$ can be significantly large. If k is close to n, then the join will still be $O(n^2)$. However, recall that we are not actually interested in the orthogonal range query ϕ. What we really want is the value of agg on the elements returned by this query. If k is large, then there will be a high degree of overlap between the elements selected for each unit, and so we can share this computation in computing the aggregate.

We do this in one of two ways, depending on the nature of agg. The simplest case is when agg is *divisible*.

DEFINITION 5.1. An aggregate agg is *divisible* if there is a function f such that

$$\mathrm{agg}(A \setminus B) = f(\mathrm{agg}(A), \mathrm{agg}(B))$$

whenever $B \subseteq A$. The aggregate sum is an example of such an aggregate, since $\mathrm{sum}(A \setminus B) = \mathrm{sum}(A) - \mathrm{sum}(B)$ whenever $B \subseteq A$. The aggregate count is also divisible, as are all the statistical moments. However, min and max are not.

When the aggregate agg in $\pi_{*,\mathrm{agg}(*)}$ is divisible, we can improve the performance by replacing the last layer of the layered range tree with an index that contains the aggregates, not the elements. For example, suppose we have an orthogonal range query on just the position of the units. Normally, we would construct a layered range tree on the x and y values; assume for the purposes of this example that we layer these ranges x then y. In this layered range tree, each x node would contain the y-index of nodes with x values in that range. However, instead of placing the units at the leaves of the y-index, we put the aggregate value of all of units whose y value is less than or equal to the value at that leaf. This idea is illustrated in Figure 8. The fact that our aggregate is divisible means that we can recover the aggregate of any range in a fixed number (2^d) of queries of the tree. Furthermore, this technique is compatible with fractional cascading. Therefore, in this case, we can compute the index nested loop join for $\pi_{*,\mathrm{agg}(*)}$ in time $O(n \log^{d-1} n)$, where d is the number of continuous attributes in the orthogonal range query. This is a definite improvement over $O(n^2)$.

Many of the aggregates in our case study in Section 3.2, such as centroid or the number of units, are divisible aggregates over orthogonal range queries. The AMAI file `common.eai` contains

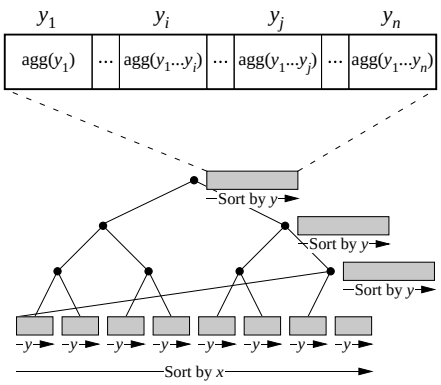

Figure 8: Divisible Aggregates in a Layered Range Tree

other examples, such as the use of sums weighted on troop type to measure the strength of an army. In some cases, such as our centroid query, the aggregate is really a tuple of aggregates over the same selection σ_ϕ. In that case, we can combine these aggregates into one index structure by replacing the list of aggregates in Figure 8 with a list of aggregate tuples.

However, two very important aggregates — maximum and minimum — are not divisible. These aggregates are necessary for queries such as finding the weakest (i.e. least healthy) unit in range. For these aggregates, we cannot use the technique illustrated in Figure 8 to get rid of the value k in the $O(\log^{d-1} n + k)$ look-up for an orthogonal range tree.

One option is to build a multi-resolution aggregate tree [15] for the entire space, and then query this tree for each unit. Unfortunately, these trees return only approximate results, and there is no guarantee on their query performance. However, there is another possible optimization. In many instances, the size of the range will be constant in one of the dimensions of the orthogonal range query. For example, units of the same type all have the same weapon and visibility range. If, as before, we assume that this visibility range is represented as a box, this means that all of these units have the same size for their x and y range queries. When this is the case, we can compute *max* and *min* using a sweep-line algorithm [9]. In two dimensions, the procedure is as follows:

- Choose an axis for which the size of the range is constant. Call this axis y and let the size of the range be r.
- Construct a binary tree ordered on the remaining axis x.
- Use this tree to perform a variant of a sweep-line algorithm on axis y.
  - Initially annotate each leaf of the tree with a default value: ∞ for $\min$ or $-\infty$ for $\max$.
  - Sweep with a range of r. When a unit moves into range r, replace the default value with the actual value. When a unit reaches the center of the range, use the tree to compute the aggregate within the unit's x range (this takes $O(\log n)$ time). When a unit moves out of the range r, replace the actual value with default value (∞ or $-\infty$).
  - At each step of the sweep, percolate any changed leaf values up the tree so each interior node is labeled with the aggregate of its leaf descendants.

This technique is illustrated in Figure 9. The technique generalizes to d dimensions, with performance $O(n \log^{d-1} n)$, A total of n items enter and exit the sweep, and it costs $O(\log^{d-1} n)$ time to percolate the aggregate values for each unit that enters or exits.

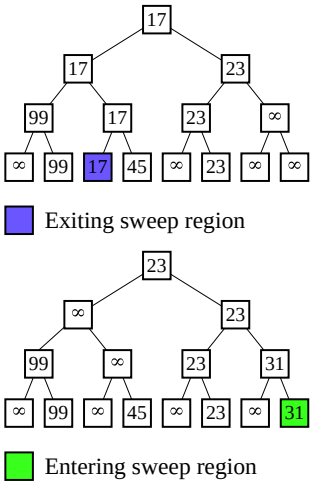

Figure 9: Sweep-Line of $\min$ on Constant Region Size

5.3.2 Spatial Aggregates

While many of our aggregates are $\sum$, $\max$, and $\min$ on orthogonal range queries, not all of them are. For example, we frequently use the aggregate that returns the nearest unit. The AMAI file `common.eai` contains other interesting spatial aggregates, like searching for the unit that can reach location (x, y) in the shortest amount of time. Many of these aggregates have been studied extensively in the area of computational geometry, and there exist specialized indices designed to solve many of them quickly. For example, an efficient way to find the nearest unit is to use a kD-tree [4]. Designing these types of indices is beyond the scope of this paper.

However, note that for many of these spatial aggregates, the non-spatial part of the query is still an orthogonal range query. We do not just want the nearest unit; we want the nearest unit that is an archer, or the nearest unit whose armor we can penetrate. Therefore, to process these type of queries, we place the spatial indices as the lowest level of a layered range tree. For example, to find the nearest unit whose armor we can penetrate, we create a tree for the armor values, and attach a kD-tree to each node in this tree. This structure can be created in $O(n \log^2 n)$ time and space; each probe requires $O(\log^2 n)$ time.

5.4 Processing the Combination Operator

The combination operator $\oplus$ serves two purposes: it allows us to combine different types of effects, and it allows us to combine several effects of the same type from different actions. When we look at $\oplus$ in the latter case, we can view $\oplus$ as an aggregate. Indeed, this is often the definition of $\oplus$. For example, in the case of attacks, $\oplus$ sums up all of the attacks on each unit to determine the total damage to apply. In this case of a nonstackable effect like our healing aura, $\oplus$ computes the maximum aura for each individual, so that we can perform that much healing.

For many actions, the effect of the action only applies to a single unit. Each move action only effects the unit itself; each archer can only fire at some target. However, some actions, like the healer's healing aura shown in Figure 5, can affect multiple units. In this case, we again may need to be concerned about $O(n^2)$ behavior; if roughly n units perform area of effect actions that apply to n units, then combining them is $O(n^2)$. In practice, this is unlikely. For reasons of game balance, while unit observations cover large areas, the area affected by an action is typically very small. There are exceptions to this rule, like the nuclear weapons in *Starcraft*;

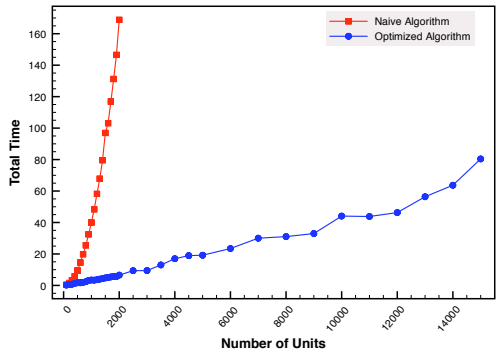

Figure 10: Comparison of Naive versus Indexing

however, only one nuclear weapon can be fired per clock tick, so this not an example of $O(n^2)$ update behavior.

With that said, our goal is to have a processing model that is generic enough for scaling games of the future, not just games of the past. Therefore, we want to optimize $\oplus$ so that it is as efficient as the rest of our operations. If $\oplus$ is just attribute-wise sum or max, as is commonly the case in games, then we can optimize this operation further.

We observe that all area-of-effect actions of the same type commonly have the same range. For example, all healers exude an aura of the same size. This means that determining all of the units in the range of an effect is the same as fixing a range and determining all of the effects in the range of each unit. Therefore, to optimize $\oplus$, we arrange our query plan to group together all actions of the same type. For each such action we construct an index that contains their centers of effect. Applying $\oplus$ now consists of performing an aggregate on this index; for stackable effects this action is sum, and for nonstackable effects it is max. Hence we can use the techniques of Section 5.3 to perform this optimally.

6. EXPERIMENTS

To validate our ideas we have built a preliminary version of the discrete simulation engine component of the data-driven game system architecture as described in Section 2. This is not (yet) a general framework — the index structures we build are tailored to the particular game example we have chosen — but it demonstrates that our techniques are practical and can greatly increase the scalability of a game engine.

For our tests, we built a faithful implementation of the battle simulation game described in Section 3.2. Every NPC unit executes a simple but decidedly nontrivial script. On each clock tick, each unit evaluates about 10 aggregate queries. Many of these are divisible aggregates, like “count the enemy archers” or “compute the centroid of enemy troops in my region,” others are nearest-neighbor queries, like “find the nearest healer,” and a few are MIN queries, such as “find the weakest unit in range.”

There are two “pluggable” versions of our aggregate query evaluator. One executes aggrate queries naively, using straightforward $O(n)$ algorithms, for a total cost of $O(n^2)$ per tick. The other uses in-memory indexing as described above to reduce the complexity to $O(n\ \log(n))$ per epoch.

All divisible queries (count, sum, higher moments) are implemented using a layered range tree with fractional cascading. All such queries share the same range tree. Since the game has only two players and three unit types, we push selection on player and/or unit type to the top, giving us a total of 6 range trees–one for each player/unit type combination – to implement all the divisible aggregate queries. These six trees are completely rebuilt for each tick.

Nearest neighbor queries are implemented with a kD-tree. Again there is one such tree for each player/unit type combination. The kD-trees share some structure with the range trees. MAX style aggregates are implemented using the sweepline technique discussed in Section 5.3.1. We sweep in the Y direction, and share the top-level (X-sorted) tree of the layered range tree to implement an $O(\log n)$ dynamic interval aggregate index. All the data structures share the work of (re-)sorting the units by position at the beginning of each clock tick.

Processing for each clock tick proceeds in several phases:

- A preliminary index building phase, in which we build most of indices described above to support aggregate queries in the next phase.
- A decision phase: each unit evaluates a number of aggregate queries and decides on its next action, possibly setting some per-unit state. For example, there is a per-healer variable that is set to the amount of healing energy the healer wants to broadcast in this tick.
- A second index building phase, which can depend on values generated during the decision phase. For example, a sweepline implementation of ”max healing in range” is done in this phase.
- An action phase, for example to determine the result of an attack.
- A movement phase: Units attempt to move in directions they have decided on earlier. This is done in random order, with collision detection and very simple pathfinding rules.

To facilitate our experiments (and since we wanted to measure performance faithfully), we added a simple rule to prevent the game from finishing prematurely: Whenever a unit dies, it is ”resurrected” at a position chosen uniformly at random on the grid.

6.1 Results

Our engine is written in C++, and we compiled it using gcc on MacOS X. We ran our experiments on a 2GHz Intel Core Duo with 1.5 GB of RAM. Timings were obtained simply using the MacOS ”time” command running the simulator with a given set of parameters. The number of clock ticks simulated has been chosen to be high enough that setup time is negligible. The times reported are the number of seconds of real time required to simulate 500 clock ticks on an otherwise unloaded machine. These numbers are repeatable, and are proportional to the number of ticks simulated, to within one percent. Thus, we do not provide error bars.

Scalability With The Number of Units. For both the naive and the indexed strategies, we ran experiments varying the number of units, and varying the size of the playing grid to maintain a constant density of 1 percent of game grid squares occupied. The results are shown in Figure 10. The quadratic behavior of the naive algorithm is clearly evident. Note that the overhead of index construction is quite low: the indexed algorithm dominates the naive algorithm even for very small numbers of Units, and it is an order of magnitude faster by 700 Units. If we assume a game engine should be able to simulate at least 10 clock ticks per second, the naive system does not scale to 1100 Units on this processor, while the indexed system scales to more than 12000 Units.

Varying Unit Density. For both the naive and the indexed strategies, we ran experiments fixing the number of Units at 500, and varying the unit density between 0.5 and 8 percent. Neither algorithm is particularly sensitive to this parameter; we omit the full results due to space constraints.

In summary, our experimental results show that our techniques are very successful, leading to an order of magnitude improvement in capacity on current hardware.

7. RELATED WORK

To the best of our knowledge, this is the first paper to treat computer games as a data management problem, with its own set of problems and solutions. Previous work on games in the academic literature has focused on classic AI problems, such as machine learning and pathfinding, or on network issues, such as jitter and latency. Papaemmanouil et al. [19] have examined the issue of message dissemination in games; however, this work is part of a general study of overlay dissemination trees, and does not address the specific needs of games.

Work in the game developer literature, on the other hand, has focused on leveraging database systems to solve problems in games. Tozour [26] has designed a game architecture that uses spatial databases to aid the AI engine in runtime spatial analysis. In the case of classic board games like Chess, databases are used to manage a knowledge base for strategy evaluation [5]. However, most of this work uses existing technology, and does not suggest new areas for data management research.

8. CONCLUSIONS

Innovation in game design occurs in tandem with innovations in game architecture. The upcoming game *Spore* [28] is a perfect example of this. *Spore* is able to provide players with an unprecedented level of character customization by leveraging the power of new techniques for procedurally generating animations. At the surface, those new techniques are a means to increase the effectiveness of development in traditional settings, offloading the burden of art creation to the game code. When used to their full potential, those techniques enable a new gameplay paradigm.

The power of our AI architecture follows this two-fold path as well. It enables existing development to be greatly enhanced with substantial performance improvements, but also opens up possibilities for new kinds of games. Given a robust, expressive AI system that is individualized rather than centralized, developers are free to expose that AI to the player in ways previously not possible. Games could be made where the central player experience is about creating an exciting game experience through interesting AI.

Restricting the expressive power of the player is a relic of traditional computational limitations. With our AI architecture that supports expressive AI without reducing the number of characters, developers can enable much more sophisticated choices for players. Players will then be empowered by the game's AI, instead of hindered by it.

9. REFERENCES

[1] Personal correspondence with *Sims* technical team, 2006.

[2] AIAndy and Zalamander. Advanced melee AI in *Warcraft III*. `http://www.ecs.soton.ac.uk/~lph105/AMAI/`.

[3] Entertainment Software Association. 2006 sales, demographic and usage data: Essential facts about the computer and video game industry. http://www.theesa.com/.

[4] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proc. SGC*, pages 187–197, 1990.

[5] Murray Campbell. Knowledge discovery in Deep Blue. *Commun. ACM*, 42(11):65–67, 1999.

[6] Bernard Chazelle and Leonidas Guibas. Fractional cascading. Technical report, Systems Research Center of Digital Equipment Corporation, 1986.

[7] Yi-Jen Chiang and Roberto Tamassia. Dynamic algorithms in computational geometry. Technical Report CS-91-24, Brown University, 1992.

[8] Bruce Dawson. Game scripting in Python. In *Game Developers Conf.*, 2002.

[9] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 2nd edition, 2000.

[10] Mark DeLoura, editor. *Game Programming Gems*, volume 1. Charles River Media, 2000.

[11] Michael Dickheiser, editor. *Game Programming Gems*, volume 6. Charles River Media, 2006.

[12] David Diller, William Ferguson, Alice Leung, Brett Benyo, and Dennis Foley. Behavior modeling in commercial games. In *Proceedings of the Thirteenth Conference on Behavior Representation in Modeling and Simulation*, 2004.

[13] Epic Games. http://www.unrealtechnology.com/html/homefold/home.shtml. Corporate Website, 2006.

[14] John Laird and Michael van Lent. Interactive computer games: Human-level AI's killer application. In *National Conference on Artificial Intelligence (AAAI)*, 2000.

[15] Iosif Lazaridis and Sharad Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD Conference*, 2001.

[16] Jonathan Littke. Eight years after the creation, blizzard releases a new patch for starcraft. GosuGamers; http://sc.gosugamers.net/features.php?i=a&id=2195, 2006.

[17] Michael Mateas and Andrew Stern. Natural language understanding in faade: Surface-text processing. In *Technologies for Interactive Digital Storytelling and Entertainment (TIDSE)*, 2004.

[18] Don Moar and Jay Watamaniuk. *Introduction to the Aurora Neverwinter Toolset*. Bioware, 2006.

[19] Olga Papaemmanouil, Yanif Ahmad, Ugur Cetintemel, John Jannotti, and Yenel Yildirim. Extensible optimization in overlay dissemination trees. In *Proc. SIGMOD*, 2006.

[20] Steve Rabin. Designing a general robust AI engine. In *Game Programming Gems*, volume 1, pages 221–236. Charles River Media, 2000.

[21] Philip Rosedale and Cory Ondrejka. Enabling player-created online world with grid computing and streaming. *Gamasutra*, September 18, 2003.

[22] Brian Schwab. *AI Game Engine Programming*. Charles River Media, 2004.

[23] Sega. *Rome: Total War Manual*, 2004.

[24] Jake Simpson. Scripting and Sims2: Coding the psychology of little people. In *Game Developers Conf.*, 2005.

[25] Mustafa Thamer. Act of mod: Building Sid Meier's Civilization IV for customization. *Game Developer*, August:15–18, 2005.

[26] Paul Tozour. Using a spatial database for runtime spatial analysis. In *AI Game Programming Wisdom*, volume 2, pages 381–390. Charles River Media, 2004.

[27] Wizards of the Coast. *Revised (v.3.5) System Reference Document*, 2006.

[28] Will Wright. The future of content. In *Game Developers Conf.*, 2005.