SGL: A Scalable Language for Data-Driven Games

[Demonstration Paper]

Robert Albright, Alan Demers, Johannes Gehrke, Nitin Gupta, Hooyeon Lee,
Rick Keilty, Gregory Sadowski, Ben Sowell, Walker White
Cornell University
Ithaca, New York
{ademers, johannes, niting, sowell, wmwhite}@cs.cornell.edu
{rfa5, hl364, rpk22, gjs33}@cornell.edu

ABSTRACT

We propose to demonstrate SGL, a language and system for writing computer games using data management techniques. We will demonstrate a complete game built using the system, and show how complex game behavior can be expressed in a declarative scripting language. The demo will also illustrate the workflow necessary to modify a game and include a visualization of the relational operations that are executed as the game runs.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Games, Scripting, Aggregates, Indexing

1. INTRODUCTION

Computer games are an important and rapidly growing class of applications that require extensive computational resources. In addition to remaking the entertainment industry, games have been used in many applications including military simulations and education [3]. While games have traditionally incorporated research from the graphics community, modern games must solve a wide variety of computational challenges and can leverage results from many other areas of computer science [2].

White et al. [1] recently described a broad class of games called *simulation games* that can be optimized using data management techniques. A simulation game is one in which the player has indirect control of some or all of the characters in the game. In this case, the game engine spends a significant amount of time simulating character behavior at each game tick, and this significantly limits the scalability and expressiveness the game. Strategy games and "doll house" games like *The Sims* are examples of simulation games, whereas as some graphics-driven 3d games are not. To improve the performance of simulation games, White et al. represent objects and characters in the game in a main-memory relational database and then split up each tick into a sequence of queries followed by a single database update. By leveraging query rewrite and aggregate indexing techniques, they show how to reduce many common computations from $O(n^2)$ to $O(n \log^d n)$.

We have developed a complete system for expressing games that extends the SGL scripting language from [1]. Since much of the industry is accustomed to object-oriented or procedural languages, we have abstracted away much of the relational framework while taking care to ensure that scripts can easily be converted to a multiset relational algebra.

In this demonstration, we will show a complete game built using the SGL language as well as a novel workflow for developing data-driven games. In Section 2 we will introduce a simple schema definition language and describe the workflow used to design games in our system. In Section 3 we will detail the demonstration and describe our interface for visualizing queries.

2. THE SGL SYSTEM

In order to facilitate code reuse, most modern games separate game content from the game engine. Game content includes media like art and music, but it also includes character behavior. Typically a game designer will write scripts to specify this behavior, which are then compiled and executed by the engine.

We have extended the SGL language from [1] to support user defined aggregation and schemas with multiple tables, while taking special care to ensure that it remains usable by game designers who may not have extensive experience with the relational model.

The core of the SGL system is a compiler that will translate SGL scripts into relational algebra and a produce a set of C# classes that can be used for rendering graphics. These classes are largely independent from the SGL scripts, and only need to be recompiled if the graphics changes. We will not discuss the problem of rendering here, except to remark that our system uses a threaded model in which rendering is decoupled from the game simulation.

We will defer a complete description of the syntax of SGL until Section 3. In the remainder of this section we will describe the workflow that a designer might use to develop games in our system and introduce the schema definition component of SGL.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
ACM 978-1-60558-102-6/08/06.

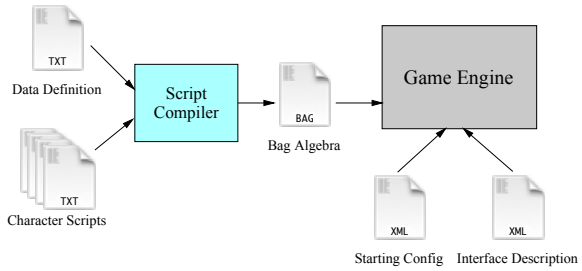

Figure 1: The SGL Workflow

2.1 The SGL Workflow

An SGL game consists of four components: i) a schema definition; ii) a set of game scripts; iii) a set of starting conditions; and iv) an interface description. We will consider how a game designer specifies each of these in our system.

Schema Definition. The game designer specifies the types of all of the objects and characters in the game by creating classes in a simple data-definition language. These are compiled into the relations that are used at runtime.

Game Scripts. All game-specific behavior is included in an SGL script that is executed at every tick. This script can access the attributes specified in the schema and can create temporary attributes for intermediate computations. SGL scripts are compiled with the schema-definition file into an intermediate file containing relational operators that is executed by the runtime system.

Starting Conditions. The initial game state is specified in a simple XML file that sets initial values for various objects in the game.

Interface Description. Finally, the designer needs to specify how the user interacts with the game. The SGL system includes an XML file that specifies what commands are executed by buttons in the GUI.

Figure 1 shows a diagram of this workflow.

2.2 The Data-Definition Language

The first component of our scripting environment is a data-definition language that allows designers to specify the schema in an object oriented way. Each relation is represented as a class, using syntax similar to C++ or Java. In Figure 2 we give an example of the `Unit` class that is part of our demonstration. The demonstration actually uses a more complex schema than the one presented here, but Figure 2 illustrate the salient parts of the data definition language.

As in an object-oriented language, every object has fields which correspond to attributes in our relation. Every attribute is either a primitive type (currently just a number), a class type, or a set type. Class types express relationships between objects, and are represented in the relational schema using foreign keys and join tables. For example, in Figure 2, the class fields of `Unit` are those under the heading of `State`. We see that most of the fields are simple numeric fields. However, one field — `squad` — is actually a set. Semantically, this is because a unit can belong to more than one squad, and so we need to represent this information as a set. In our implementation, the field `squad` is actually a separate table that must be joined with the `Unit` table. The schema of this table consists of a number (representing the squad identifier) and a foreign key into the `Unit` table.

```
class Unit {
  State:
     number unit_id;
     number player;
     number command;
     number pos_x, pos_y;
     number health;
     Set<number> squad;

  Effects:
     number move_x : AVG;
     number move_y : AVG;
     number damage : SUM;
     Set<number> joined : UNION;
     Set<number> left : UNION;

  Update:
     pos_x = pos_x + move_x;
     pos_y = pos_y + move_y;
     squad = squad UNION joined SETMINUS left;
     health = health - damage;
}
```

Figure 2: An Example Schema

In order to support efficient processing, we add two additional constructs to the data definition language. First, we allow the designer to specify the effects that can be applied to an object. Since we use set-at-a-time processing during a tick, there is only a single update phase at the end of every tick. Scripts cannot update the game state directly, they can only make assignments to effect attributes. Each such assignment occurs in isolation, and is combined into a single effect (which is used to update the state) at the end of the tick. Each effect is annotated with a combination operation that specifies how multiple effects should be combined.

Again, we see an example of this annotation in Figure 2. In this case damage should be summed, but that the movement amount should be averaged. The set-valued effects — `joined` and `left` — are all unioned. In general, we can use any aggregate function to combine effects, so long as the aggregate is defined on that data type.

Finally, the designer must provide the update rules that are used to apply effects to the game state. These rules are specified using assignments following the `Update` label. The lvalue of these assignments must be an attribute of the class. In Figure 2 we see that we update the health of a unit by subtracting the (combined) damage effect. Similarly, we use the `joined` and `left` effect sets to update the squads that include the unit.

2.3 Game Scripts

SGL is a declarative scripting language that can easily be compiled to a multiset relational algebra. As with the data-definition language, we have made the syntax as simple and imperative as possible.

The syntax includes standard constructs such as assignment using the `let` keyword, `if-else` conditionals, and common arithmetic operations. The syntax also includes the `me` keyword, which refers the object executing the script. It is important to stress that these features are ultimately compiled to relational operations. For instance, `let` simply adds a temporary column to a relation or creates a temporary join table, depending upon the data type being created.

Every expression in SGL is either numerical or set-valued. Numerical expressions represent (possibly temporary) attributes of some relation, whereas set-valued expressions represent one or more tuples. SGL uses a familiar dot notation for projections. For instance, `u.health` returns the health of the Units tuple `u`. By referring to an entire table instead of a single tuple (e.g. class instance) can also use this syntax to access an entire column; for example, `Units.health` would return the set of all health values for all units.

To support selections, we introduce the `all-where` construct. This returns the set of tuples satisfying a particular expression. For instance, the query

```
all (u in Units where u.health > 10)
```

returns the set of all units with health greater than 10. We can also apply aggregates to these selections, or to any set-valued expression. For example, `MIN(Units.health)` will return the smallest health of any unit.

The SGL language can also support more complex aggregate expressions involving group-by clauses. To present this to designers in an object-oriented fashion, we have the `over-each` control structure. This is a loop-like structure that allows the script to evaluate expressions over the elements of set. For example, the expression

```
over u in unit {
    let u.maxsquad = MAX(squad);
}
let allmax = MAX(Unit.maxsquad);
```

uses an `over-each` expression to determine the maximum squad identifier for each unit. Though `over-each` looks like like iteration, the statements in the body of this control-structure are actually computed in parallel and combined using the $\oplus$ operator described in [1].

2.4 The XML Files

As explained in Section 2.1, the game designer also needs to present two sets of XML files. Of these, the starting configuration is the most self-explanatory. This XML file defines the initial tables for each of the game objects defined in the data-definition language. For each object in the game, this XML file needs to specify a value for every attribute under the `State` heading.

The second XML file, the interface description, is much more subtle. For the most part it specifies obvious user interface elements like the location and size of buttons, or the icon on a button. However, it also contains specific logic for controlling the game script. To understand this game-specific logic, we first need to understand how the user interface typically works in simulation games.

In simulation games we control the game objects indirectly through *commands*. These commands are parameters that that influence the behavior of the object in the game script. For example, in Figure 2, we have implemented commands as a simple state attribute in the `Unit` data type. To determine its next action, a unit uses `if-then` statements to pick the correct behavior for that command.

The value of this `command` attribute is determined by the player's interaction with the game. While playing the game, a player uses a bounding box to select the units to control, and then issues the command by either pressing a button or a special key-combination. This, therefore, is the purpose of the interface description XML file. In addition to user interface elements, the interface description specifies how a button or key-combination changes the state of the currently selected game objects.

Figure 3: The Knights and Archers Game

3. DEMONSTRATION FEATURES

Our demonstration contains three major components: (i) A fully functional game; (ii) an interface for visualizing query execution at the relational algebra level; and (iii) a demonstration of modifying an SGL script and changing the game behavior.

3.1 An SGL Game

We have implemented a modified version of the knights and archers game described in [1] using Microsoft's XNA framework. This is an extension to C# that includes basic functionality for rendering graphics and handling user input. The game is squad-based, and users will be able to group units together and direct them to attack other squads. Additionally, the interface will include command buttons that can be used to perform more complex behaviors. Figure 3 shows the user interface of the game.

We have implemented all of the behavior in this game using the SGL language, and we will demonstrate a number of specific queries to show how complex actions can be expressed in our system. For example, the following fragment is used to direct a unit to move away from the centroid of all enemy units (i.e. those units controlled by a different player).

```
let enemies = (all u in Units
               where u.player != me.player);
let centroid_x = AVG(enemies.x_pos);
let centroid_y = AVG(enemies.y_pos);

let me.move_x = (me.pos_x - cosest.pos_x)/norm;
let me.move_y = (me.pos_y - closest.pos_y)/norm;
```

Here `me` is a special keyword refering to the unit executing the script, and `norm` is shorthand for `sqrt(move_x`^{2}`+move_y`^{2}`)`. This query first selects all units controlled by a different player and computes their centroid using the `AVG` aggregate. It then directs `me` to move along a vector in the opposite direction using an effect assignment. Note that if there were multiple statements assigning a value to an effect like

Figure 4: The query visualization tool

`me.move_x`, then the values would be combined using the appropriate aggregate (`AVG` in this case).

We will demonstrate the results of this query in two different ways. First, we will compile our game script into the bag algebra using the script compiler, as shown in Figure 1. We will display the bag algebra side-by-side with the game script and compare how they each express this same query. Then, we will run the game with this script and show how this behavior appears on the screen. We will use the starting configuration file to populate the game with several enemies arranged in a crowd, and show that friendly units use this query to run away from the center of this crowd.

3.2 Visualizing Queries

Looking at the bag algebra file is one way to understand how our game scripts are processed using relational operators. However, ideally we would like to see how a query plan corresponds to specific actions on screen. For that purpose we have created a query visualization tool that will allow users to see the relational operators as they are being executed in the game.

To understand how the query visualizer works, recall that a traditional query plan is a pipeline through which the tuples of our relations pass. In our game, the tuples are units, which are displayed onscreen as character avatars. Thus, to visualize our query pipeline, we need only show the relational operator currently being executed, and highlight the characters that are being processed by that operator.

Figure 4 shows this visualization interface for a simple query. We display the current relational operator in the box, while we highlight the subjects of this relational operator on screen. In order to better see this, our GUI has an option to enable "step mode." In step mode, gameplay slows down and its speed can be controlled by the user. In addition to visualizing the query pipeline, this feature can help the game designer debug and optimize the game scripts.

3.3 Modifying Scripts

In order to demonstrate the workflow necessary to develop a game using SGL, we will show how to modify a script to change the game semantics. First, we will demonstrate how to add new behavior to the game by adding the following script, which finds the enemy unit closest to `me`, and directs `me` to move toward that unit.

```
over each u in unit {
   if (x.player != me.player) {
      let u.rel_dist = sqrt((me.pos_x-u.pos_x)^2
                            +(me.pos_y-u.pos_y)^2);
   }
}
let closest = ONEOF(all x in Units
         where x.rel_dist == MIN(Units.rel_dist));

let me.move_x = (closest.pos_x - me.pos_x)/norm;
let me.move_y = (closest.pos_y - me.pos_y)/norm;
```

This script uses the `over-each` control structure to compute the distance from the unit to each of its neighboring enemies. It then uses a `MIN` aggregate to find the distance to the nearest enemy unit. Finally, it uses the `all-where` operator to choose only those enemies that are the minimum distance away, and then uses `ONEOF` to pick nondeterministically among them.

Given this new behavior, we will also demonstrate how to incorporate it into the user interface of the game. As described in Section 2.4, we use XML files to specify how a command button affects the behavior of a unit within a game script. As part of this demonstration, we will show how to modify the interface description XML file to produce a button that, when pushed, instructs a selected unit to move towards the nearest enemy unit.

3.4 Modifying the Schema

To make more complex changes, designers may also need to modify the game schema. As part of our demonstration, we will add the attribute `num_arrows` to the schema shown in Figure 2. We will also add the `num_fired` effect and an update rule that decrements `num_arrows` whenever the unit fires an arrow. Note that these changes do not require changing any of the graphics in game, so the engine does not need to be recompiled. Using these new attributes, we can demonstrate the following script, which directs an archer to fire at the nearest enemy if he or she has enough arrows or run away if not.

```
if (me.num_arrows > 0) {
   me.num_fired = 1;
   closest.damage = _ARROW_DAMAGE_;
} else {
   let me.move_x = (closest.pos_x-me.pos_x)/norm;
   let me.move_y = (closest.pos_y-me.pos_y)/norm;
}
```

As before, we will also show how to modify the interface description XML file to produce a button that commands a selected unit to perform the above behavior.

4. CONCLUSIONS

SGL is an expressive scripting language for developing data-driven games. By allowing designers to modify the schema, scripts and preference files, we have developed a system in which it is easy to change gameplay without altering the core engine. In addition, we have shown that game development can derive significant benefits from database techniques, and we have just scratched the surface of what we can do with this approach [2].

Acknowledgments. This research is based upon work supported by the National Science Foundation under Grant IIS-000492612, by the Air Force under Grant AFOSR FA9550-07-1-0437, by a grant from Microsoft Corporation, and by a Faculty Development Grant from the New York State Foundation for Science, Technology, and Innovation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

5. REFERENCES

[1] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proc. SIGMOD*, pages 31–42, 2007.

[2] W. White, C. Koch, N. Gupta, J. Gehrke, and A. Demers. Database research opportunities in computer games. *SIGMOD Record*, September 2007.

[3] M. Zyda. Introduction: Creating a science of games. *Communications of the ACM Special Issue: Creating a science of games*, 50(7):26–29, July 2007.