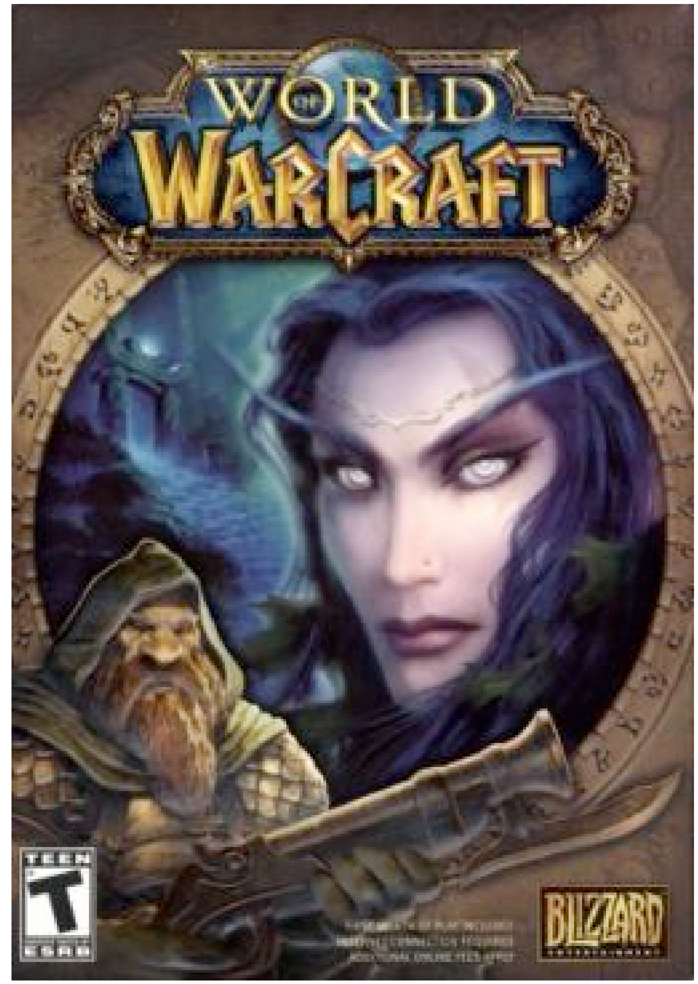
Scaling Computer Games To Epic Proportions

Walker White
Cornell University

Joint work with Al Demers, Johannes Gehrke, Christoph Koch, and Rajamohan Rajagopolan

Computer Games

- $7B in sales in 2005
  - Outperforming the movie industry
- Unique challenges
  - Virtual environments
  - High degree of interactivity



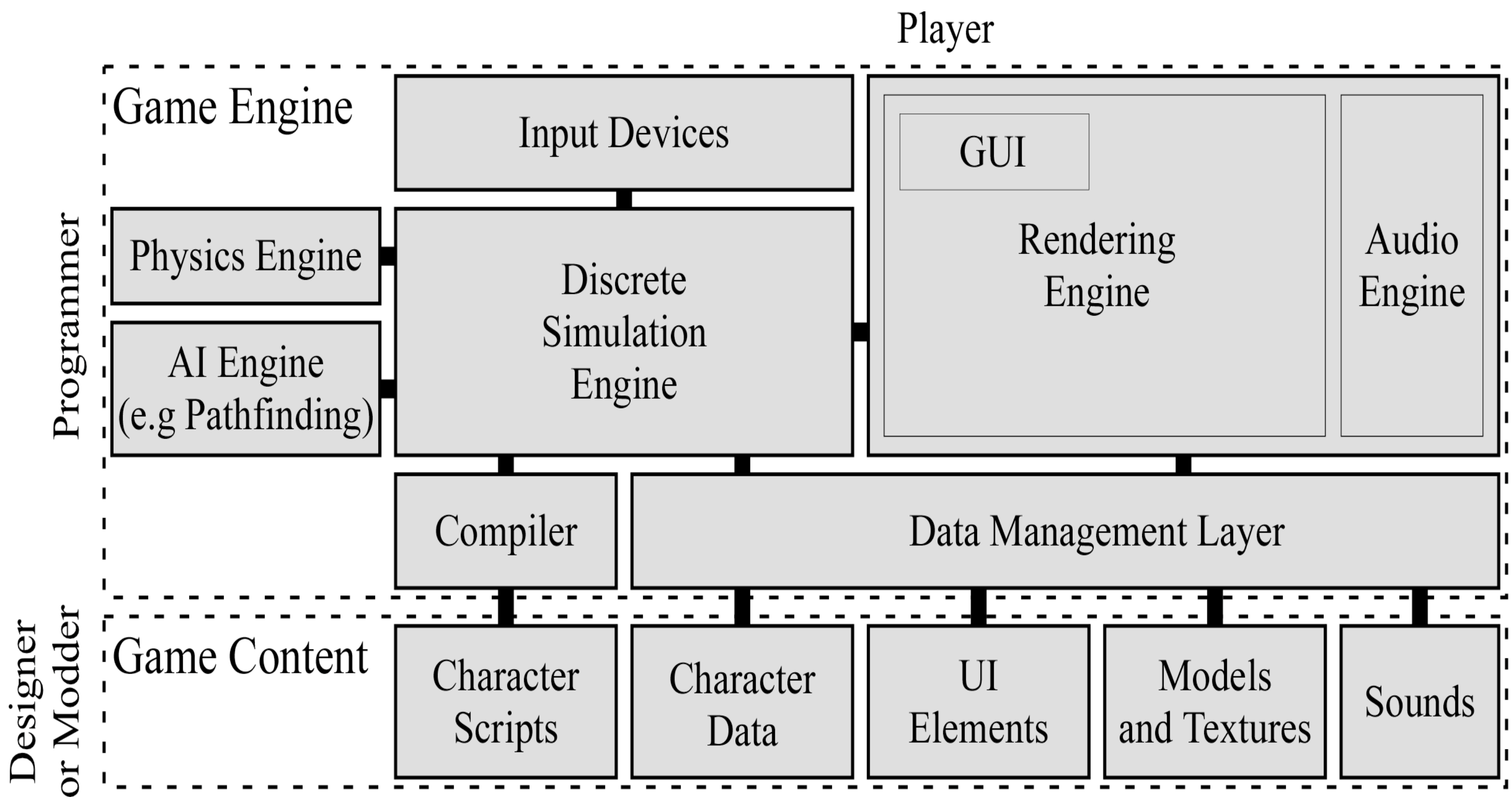
Game Design

- Game design brings together many disciplines
  - Art, music, computer science, etc...
- Development brings together different skills:
  - Programmers: Create the game engine
    - Focus on technological development
  - Designers: Create the game content
    - Typically artistic content
    - But may include (programmed) character behavior

Data-Driven Game Design

- Today's games are *data-driven*
  - Game content is separated from game code
- Examples:
  - Art and music kept in industry-standard file formats
  - Character data kept in XML or other data file formats
  - Character behavior specified through scripts
    - Programmed via scripting language

Data-Driven Game Design



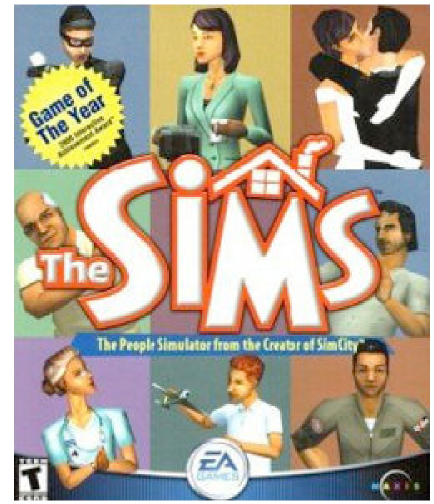
Advantages of Data-Driven Design

- Engine is reusable.
  - Able to recoup R&D costs over several games.
  - Possible to license engine to other companies.
    - Example: The Unreal engine

- Can extend the life span of the game
  - Modder communities develop around the game
    - Keep game fresh and new
  - User-created content becoming very popular

Talk Outline

- Simulation Games
- Scaling Games with SGL
- Optimizing SGL
- Experimental Evaluation



This Talk: Simulation Games

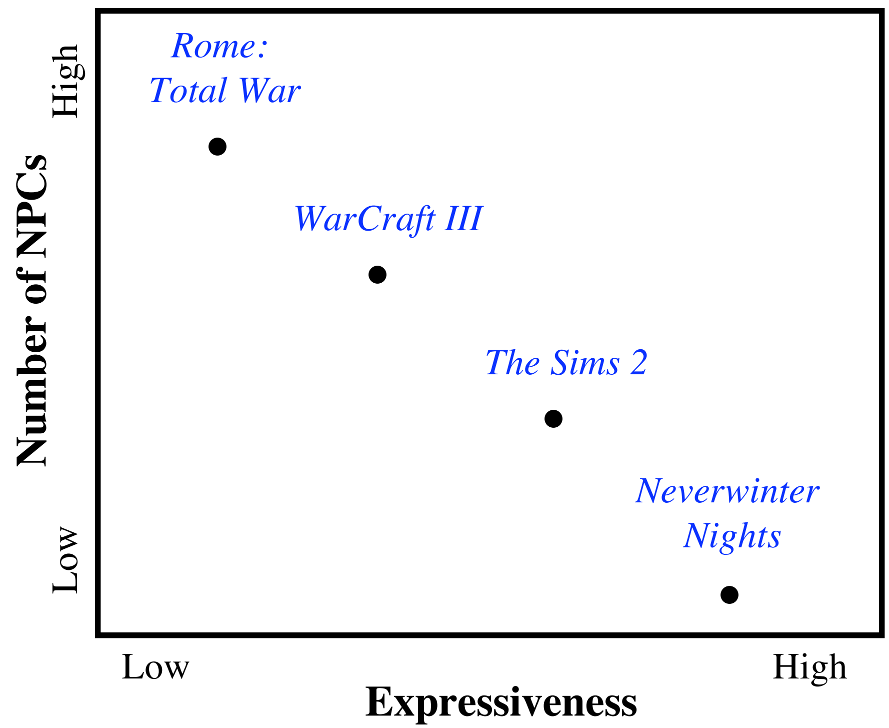
- What are simulation games?
  - Characters can interact w/o player input
  - Non-Player Characters (NPCs): indirect control
- Example: Real-Time Strategy (RTS) games
  - Troops move and fight in real time
  - Player control via limited number of commands
  - Player multitasks between large number of units

RTS Demonstration

Expressiveness vs. Performance

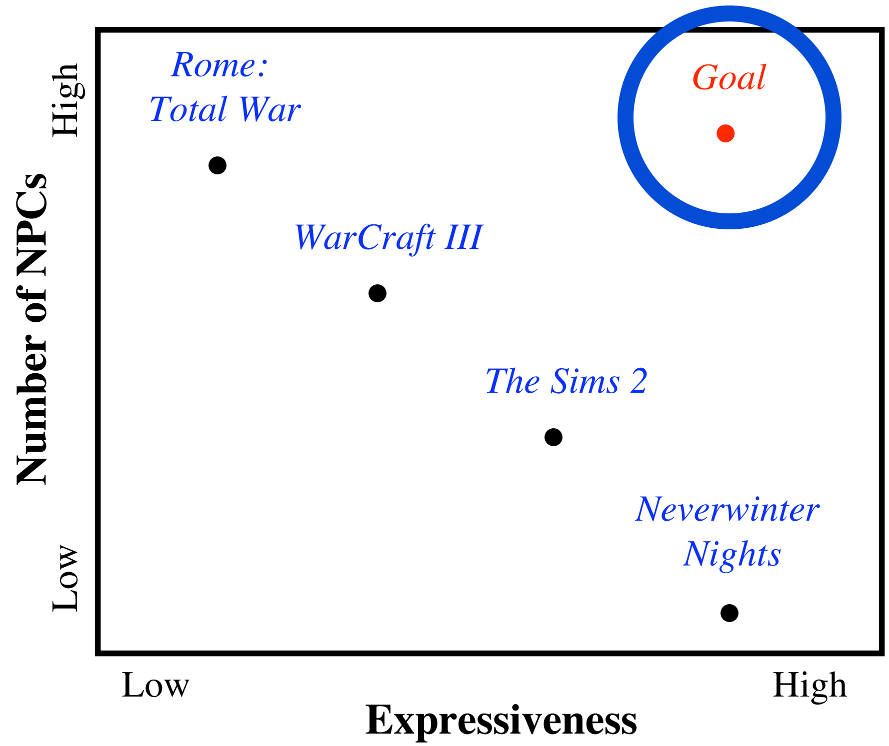
- Expressiveness:
  the range of behavior
  scriptable outside engine

- As # of NPCs increases
  expressiveness decreases
  - *Neverwinter Nights*
    - Each NPC fully scriptable
  - *WarCraft III*
    - Script armies, not NPCs
    - Can only “fake” NPC control
    - Little NPC coordination
  - *Rome: Total War*
    - No individual control at all



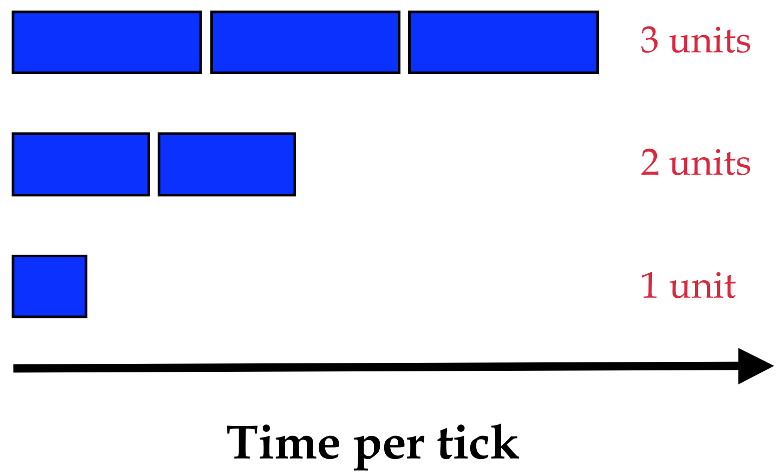
Expressiveness vs. Performance

- Expressiveness: the range of behavior scriptable outside engine
- As # of NPCs increases expressiveness decreases
  - *Neverwinter Nights*
    - Each NPC fully scriptable
  - *WarCraft III*
    - Script armies, not NPCs
    - Can only “fake” NPC control
    - Little NPC coordination
  - *Rome: Total War*
    - No individual control at all



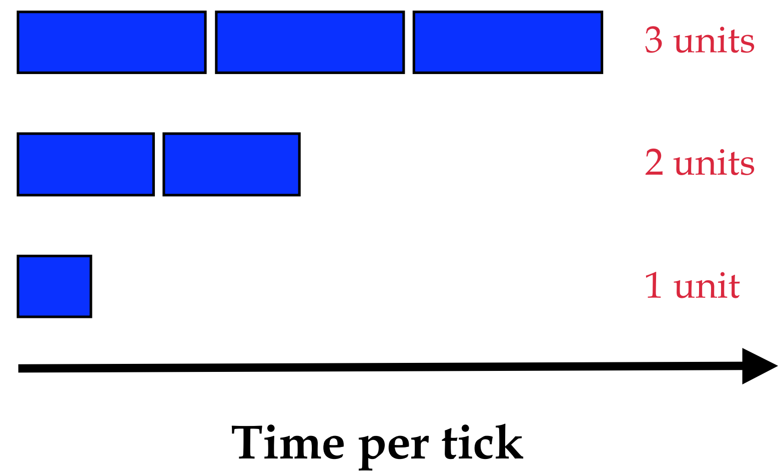
Why Is Scaling NPCs Hard?

- Can be very expensive
  - O(*n*) to process all units.
  - Observations may be O(*n*)
- Example: morale
  - Units afraid of skeletons
  - Chance of running proportional to # of skeletons
  - O(n) to count skeletons
  - O(n^2) to process all units.



Why Is Scaling NPCs Hard?

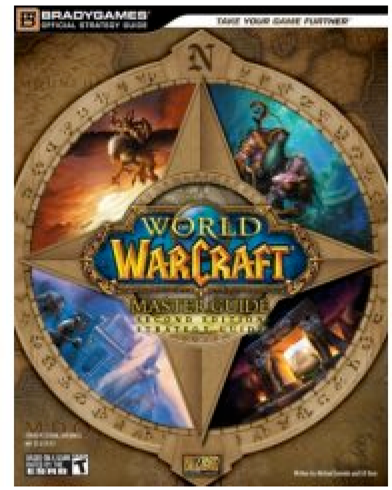
- Can be very expensive
  - O(*n*) to process all units.
  - Observations may be O(*n*)
- Example: morale
  - Units afraid of skeletons
  - Chance of running proportional to # of skeletons
  - O(n) to count skeletons
  - O(n^2) to process all units.



- Want computation close to graphics frame rate.

Talk Outline

- Simulation Games
- Scaling Games with SGL
- Optimizing SGL
- Experimental Evaluation



Scaling Scripts to Many NPCs

- Idea: Use declarative language for scripts.
- Analysis shows:
  - Typically a set of `if-then` rules.
  - Iteration is restricted to:
    - Computing an aggregate of a collection of objects.
    - Applying an update to the environment.
    - Processing an array of fixed size.

Talk Outline

- Simulation Games
- Scaling Games with SGL
  - Simulation == Queries and updates
  - The SGL Language
- Optimizing SGL
- Experimental Evaluation

Inside the Simulation Engine

- Actions divided into “ticks”.
- During a tick, each unit
  1. Reads the environment
  2. Determines its current action
  3. Performs action, creating one or more effects
     - An effect may alter a unit’s own state (i.e. movement)
     - An effect may alter the state of others (i.e. damage)

Inside the Simulation Engine

- Actions divided into “ticks”.
- During a tick, each unit
  - Database queries
    1. Reads the environment
    2. Determines its current action
  - Database updates
    3. Performs action, creating one or more effects

The Environment Table

- The environment is a single table *E*.
  - Each unit a row in the table.
- Schema is unit state *and* possible effects.
  - Position: Unit state
  - Movement: Unit effect

State			**Effect**	
`Name`	`Pos_x`	`Pos_y`	`Move_x`	`Move_y`
Bob	2	10	3	2
Doug	-4	3	1	0
Alice	0	-1	0	0

Processing Effects

- At end of tick, effects update environment
  - All effects are processed simultaneously
  - Have rules to combine effects
    - Must be order independent
    - Currently games use aggregate functions
    - Examples: sum, product, min, max
  - Combination is single effect, used for update

The Environment Table

- The environment is a single table *E*.
  - Each unit a row in the table.

- Schema is unit state *and* possible effects.

- Schema annotated to tell which is which.
  - State subschema annotated by `const`.
  - Effects annotated by combination function.
    - Examples: `sum`, `min`, `max`

Example Environment Table

E(`key`^{const},	"Key"; used to identify unit.	
`player`^{const},	Player controlling unit	STATE
`pos_x`^{const},	Current *x*-position of unit	
`pos_y`^{const},	Current *y*-position of unit	
`health`^{const},	Current health of unit	
`move_x`^{sum},	Amount to move unit on *x*-axis	
`move_y`^{sum},	Amount to move unit on *y*-axis	EFFECTS
`damage`^{sum},	Amount of damage to do to unit	
`heal_aura`^{max}	Amount to heal unit	
)		

Formal Processing Model

- Each unit performs a single action.
  - A query that produces a set of effects.
  - Returns the subtable of affected units.
    - `Const` attributes are unmodified.
    - Effect attributes modified with effect amounts.
- Effects of each action are combined.
  - Produces a new table E_u of all updated units.
- Post-processing step updates state from effects.
  - Produces the new table E for the next tick.

The Post-Processing Step

- Is just an SQL query!
- Example:

```
SELECT u.key, u.player,
       u.pos_x + u.move_x * norm AS pos_x,
       u.pos_y + u.move_y * norm AS pos_y,
       u.health - u.damage + u.heal_aura
         AS u.health,
FROM E u
WHERE u.health > 0
```

Talk Outline

- Simulation Games
- Scaling Games with SGL
  - Simulation == Queries and updates
  - The SGL Language
- Optimizing SGL
- Experimental Evaluation

Defining Actions: SGL

- **S**calable **G**ames **L**anguage
  - Functional language
    - Used to choose NPC actions
  - Aggregate functions to perform observations
    - Built-in or definable in SQL
  - Action functions to produce effects
    - Built-in or definable in SQL

Example SGL Script

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

Aggregate Function Definitions

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

Action Function Definitions

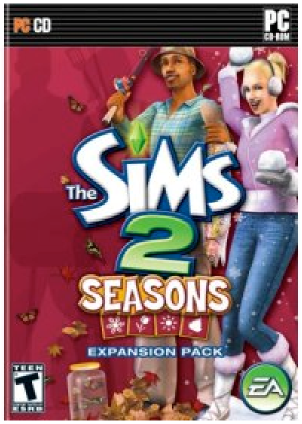
```
function MoveInDirection(u,x,y) returns
  SELECT e.key,e.player,e.pos_x,e.pos_y,e.health,
         x-e.pos_x AS move_x, y-e.pos_y as move_y,
         e.damage,e.heal_aura
  FROM E e WHERE e.key=u.key;

function FireAt(u,target_key) returns
  SELECT e.key,e.player,e.pos_x,e.pos_y,e.health,
         e.move_x, e.move_y,
         e.damage+(_ARROW_HIT_DAMAGE - _ARMOR) *
                  (Random(e,1) mod 2) as damage,
         e.heal_aura
  FROM E e WHERE e.key=target_key;
```

Advantage of this Model

- Units often perform a lot of shared computation.
  - Example: Units all processing the same command
  - Optimize with set-at-time processing.
    - Determine all effects with a single database query.
    - Apply all effects as single update at end of tick.
- Sometimes computation is only overlapping.
  - Example: Counting number of skeletons.
    - Units overlapping, not same, line-of-sight.
  - Optimize with indexing techniques.

Talk Outline



- Simulation Games
- Scaling Games with SGL
- Optimizing SGL
  - Set-at-a-time processing
  - Aggregate Indexing
- Experimental Evaluation

Combining Effects Together

- Combination operation $\oplus$.
  - Operates on a set: $\oplus E$
  - Merges rows of same “key” according to annotation.
  - Example:

$\oplus$

$\texttt{Key}^{\texttt{Const}}$	$\texttt{Player}^{\texttt{Const}}$	$\texttt{Damage}^{\texttt{Sum}}$
1	Bob	3
1	Bob	2
2	Bob	4

=

$\texttt{Key}^{\texttt{Const}}$	$\texttt{Player}^{\texttt{Const}}$	$\texttt{Damage}^{\texttt{Sum}}$
1	Bob	5
2	Bob	4

- Gives formal definition for combining of effects.

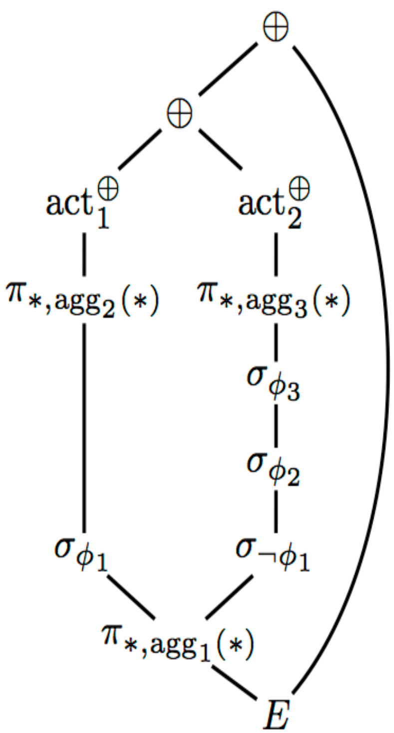
Set-At-A-Time Processing

- Define $[[f]]^{\oplus}(\mathrm{E}) = \oplus\,(\cup\{[[f]]_E(u) \mid u \in E\})$

- Process an entire "tick" as
  $[[\text{main}]]^{\oplus}(E) \oplus E = \oplus\,(\cup\,\{[[\text{main}]]_E\,(u) \mid u \in E\}) \oplus E$

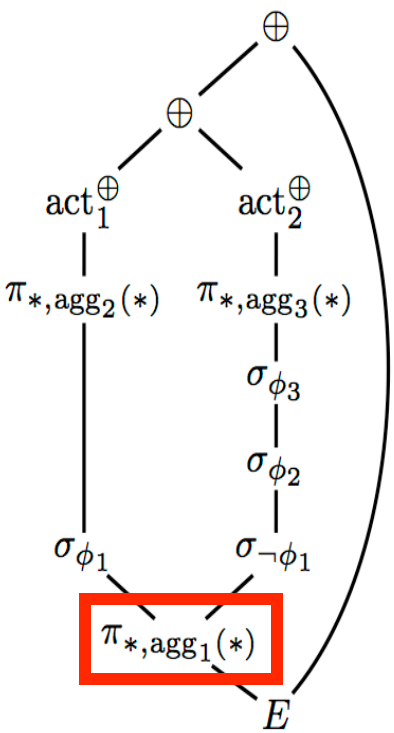
- Suggests set-processing semantics:
  $[[(\texttt{let}\ A := \vec{a})\ f\,]]^{\oplus}(E) \quad := [[f]]^{\oplus}(\pi_{*,a(*)\ \text{as}\ \vec{A}}(E))$
  $[[f_1\,;\,f_2]]^{\oplus}(E) \quad := [[f_1]]^{\oplus}(E) \oplus [[f_2]]^{\oplus}(E)$
  $[[\texttt{if}\ \varphi\ \texttt{then}\ f]]^{\oplus}(E) \quad := [[f]]^{\oplus}(\sigma_{\varphi}(\mathrm{E}))$
  $[[\texttt{perform}\ G]]^{\oplus}(E) \quad := [[g]]^{\oplus}(E)$

Algebraic Optimization



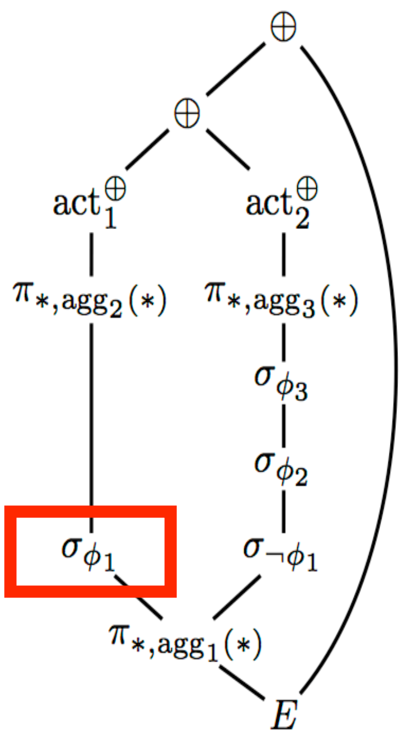

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
               (u.posx, u.posy) -
               CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



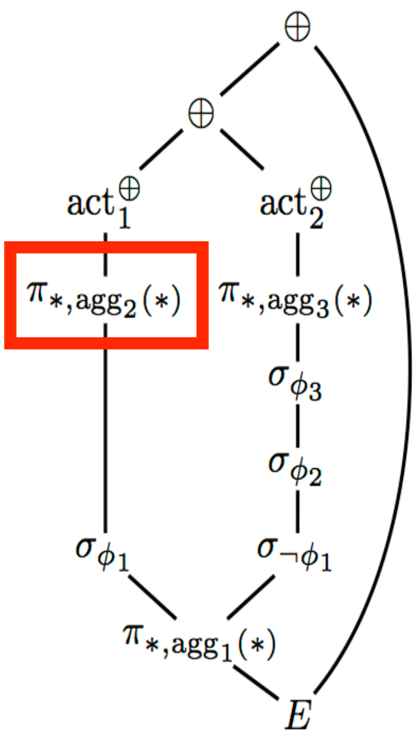

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



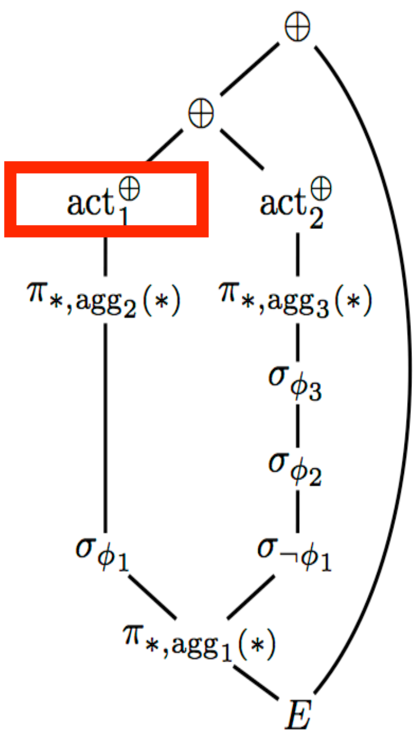

```
main(u) {
    (let c = CountEnemiesInRange(u,u.range)) {
        if (c > u.morale) then
           (let away_vector =
                (u.posx, u.posy) -
               CentroidOfEnemyUnits(u,u.range)) {
           perform MoveInDirection(u,away_vector);
         } else if (c > 0) then {
        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

Algebraic Optimization



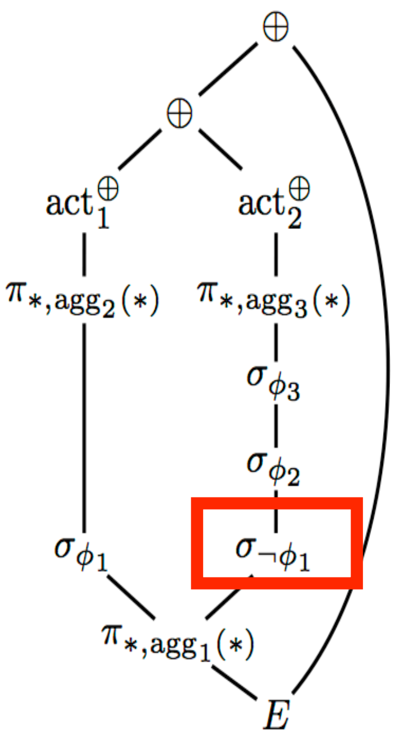

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



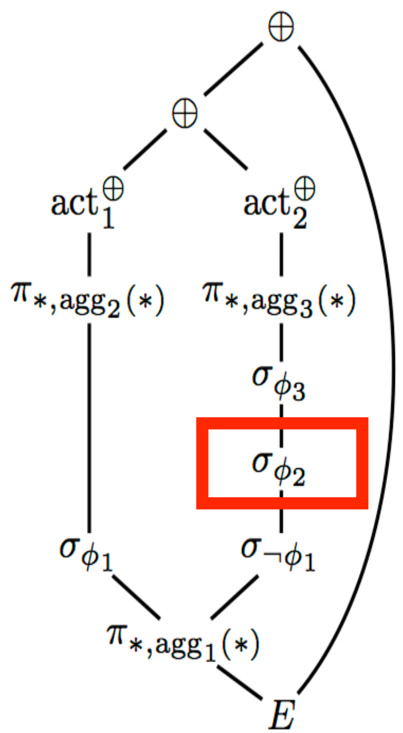

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



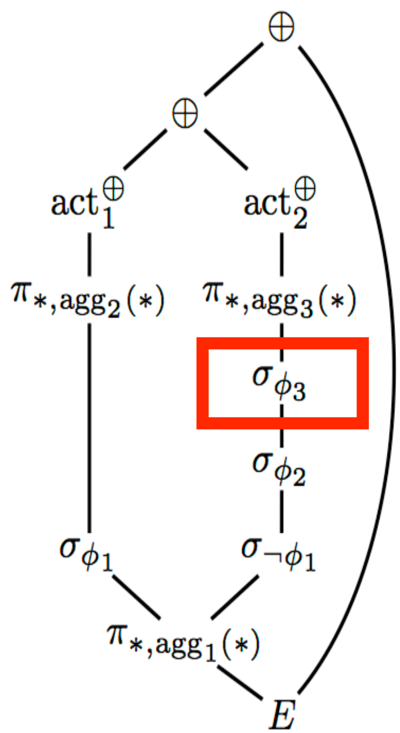

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



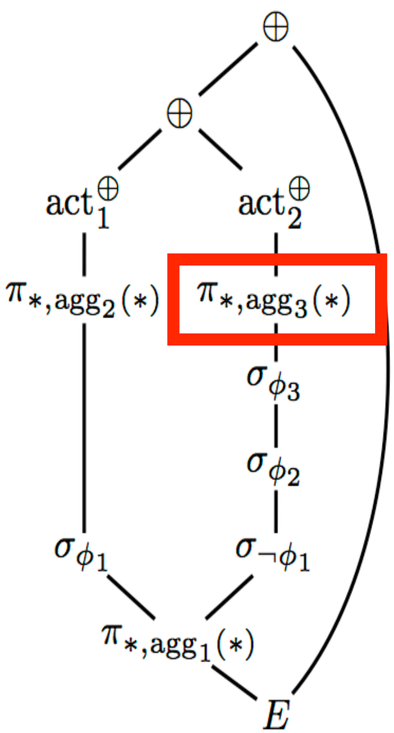

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



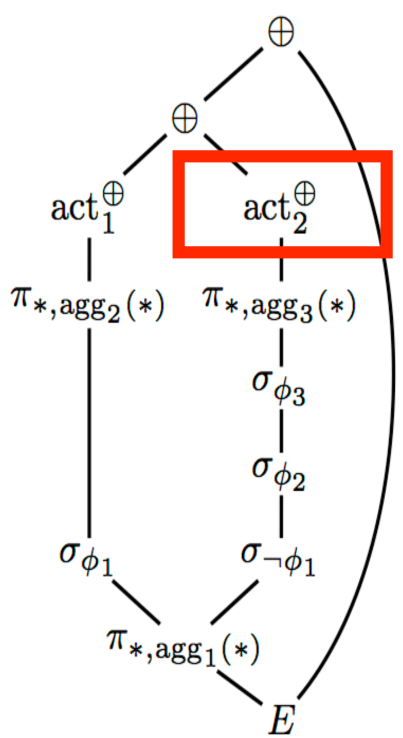

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization



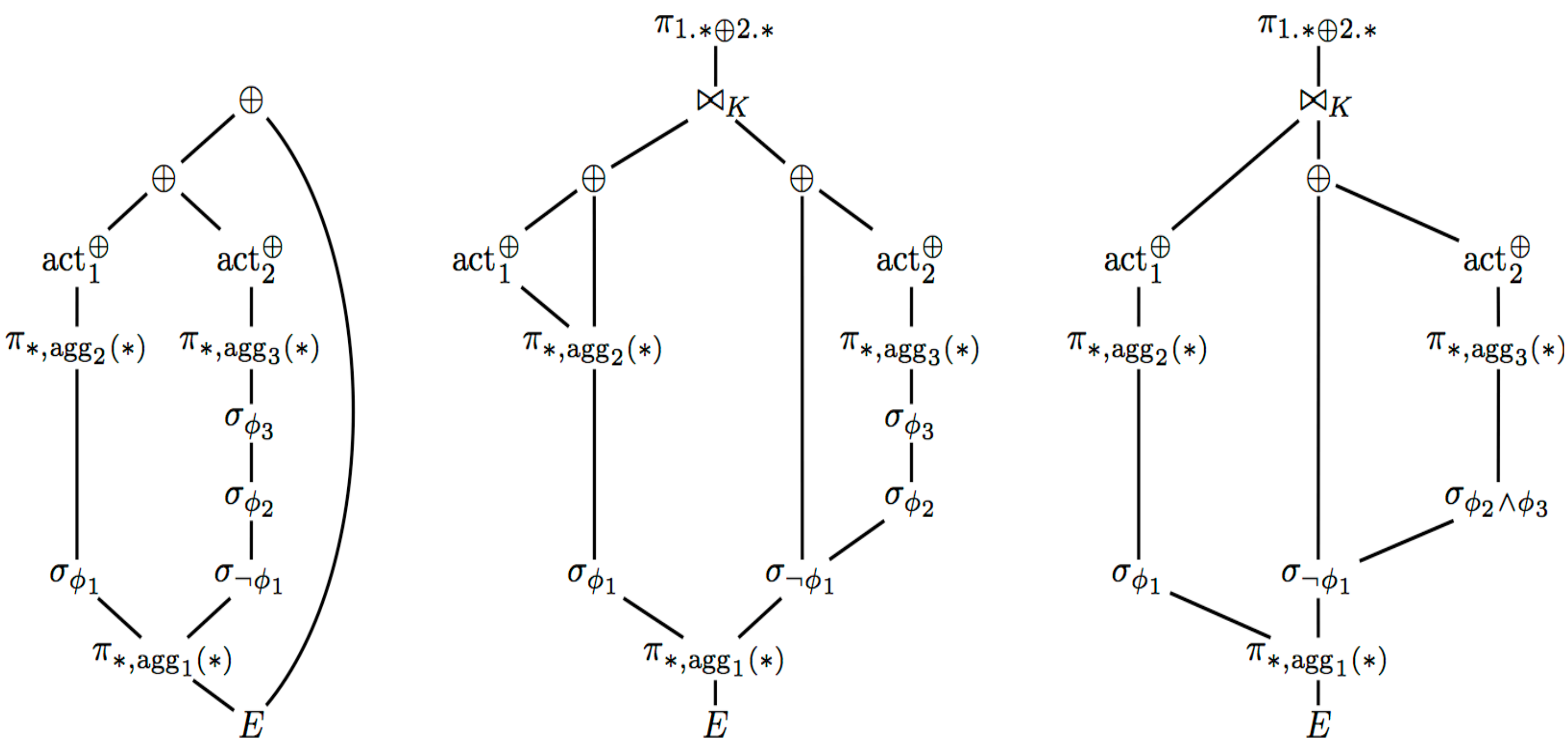

```
main(u) {
   (let c = CountEnemiesInRange(u,u.range)) {
       if (c > u.morale) then
         (let away_vector =
              (u.posx, u.posy) -
              CentroidOfEnemyUnits(u,u.range)) {
         perform MoveInDirection(u,away_vector);
       } else if (c > 0) then {
         if (u.cooldown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

Algebraic Optimization




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

Algebraic Optimization



(a) (b) (c)

Talk Outline

- Simulation Games
- Scaling Games with SGL
- Optimizing SGL
  - Set-at-a-time processing
  - Aggregate Indexing
- Experimental Evaluation

Optimizing Aggregates

- The problem:
  - Script associated with each NPC
  - Script performs aggregate computation
    - Count number of skeletons
    - In our query plans: $\pi_{*,\mathrm{agg}(*)}(R)$
  - Compute for every unit in the environment
    - $O(n^2)$ cost!
- But we "understand" the scripts!
  - Compute common aggregate for query plan

Optimizing Aggregates (Contd.)

- The problem is actually a bit harder:
  - Script associated with each NPC
  - Script performs aggregate computation that depends on the unit
    - Count number of skeletons in my neighborhood
    - In our query plans: $\pi_{*,\mathrm{agg}(*)}(\sigma_{\varphi}\ (R))$
  - Compute for every unit in the environment
    - $O(n^2)$ cost!

- But we "understand" the scripts!
  - Compute common aggregate for query plan

Solution: Aggregate Indexing

- Create an index to encode aggregates
  - Replaces computation with index lookup
  - $\pi_{*,\mathrm{agg}(*)}(\sigma_{\varphi}(R))$ now index nested loops join
- Indices for all aggregates in *Warcraft III*
  - See the paper for technical details
  - All indices are
    - $O(n \log^d n)$ to build
    - $O(\log^d n)$ to look-up
    - where d depends on arity of φ

Indices in the Processing Model

- Construct all indices at beginning of tick
  - O($n \log^d n$) for each aggregate index
- Scripts are read only queries on indices
  - Aggregates are index nested loops join
  - O($\log^d n$) look-up for each unit
  - O($n \log^d n$) costed for nested loops join
- Linear cost to post-process updates

Indices in the Processing Model

- Construct all indices at beginning of tick
  - $O(n \log^d n)$ for each aggregate index
- Scripts are read only queries on indices
  - Aggregates are index nested loops join
  - $O(\log^d n)$ look-up for each unit
  - $O(n \log^d n)$ costed for nested loops join
- Linear cost to post-process updates
- Total cost for entire tick: $O(n \log^d n)$

Talk Outline

- Simulation Games
- Scaling Games with SGL
- Optimizing SGL
- Experimental Evaluation

Experimental Evaluation

- System is currently under construction
- But performed an evaluation of the optimizations on a game simulation
    - Do we really see n^2 behavior in practice?
    - What about overhead of index construction?

Experimental Evaluation

- Combat simulation
  - Three types of units: knights, healers, archers
  - Complex, but reasonable NPC behavior.
    - Archers use knights as cover.
      - Compute centroids of archers, knights, enemies.
      - Make sure in a line, with knights at center.
    - Healers stay in between archers, knights.
      - Spread out for maximum healing.
    - Knights retreat to healers if too wounded.
- Uses d20 (D&D) mechanics for combat.

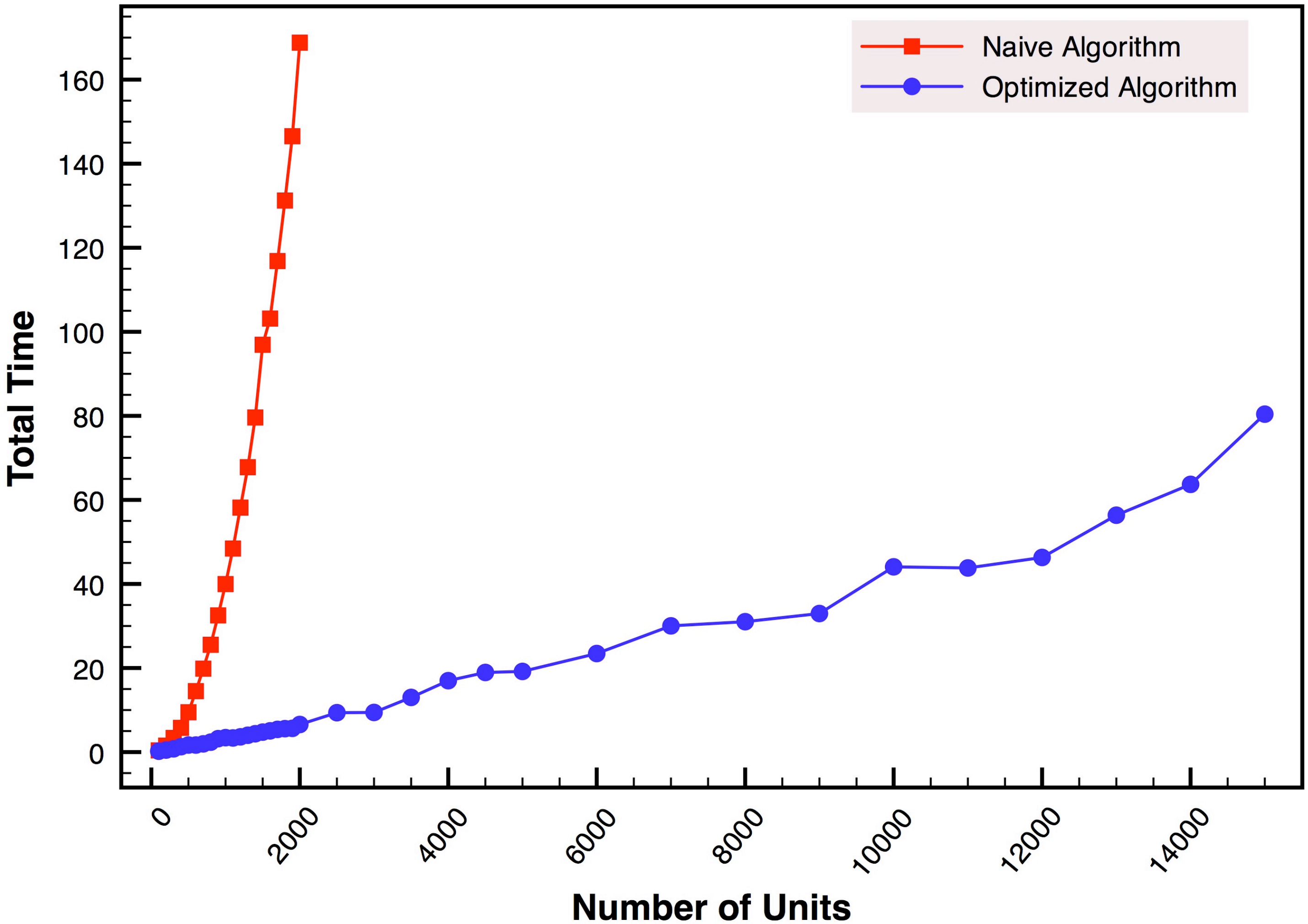
Experimental Design

- Number of NPCs vs. time for 500 clock ticks.

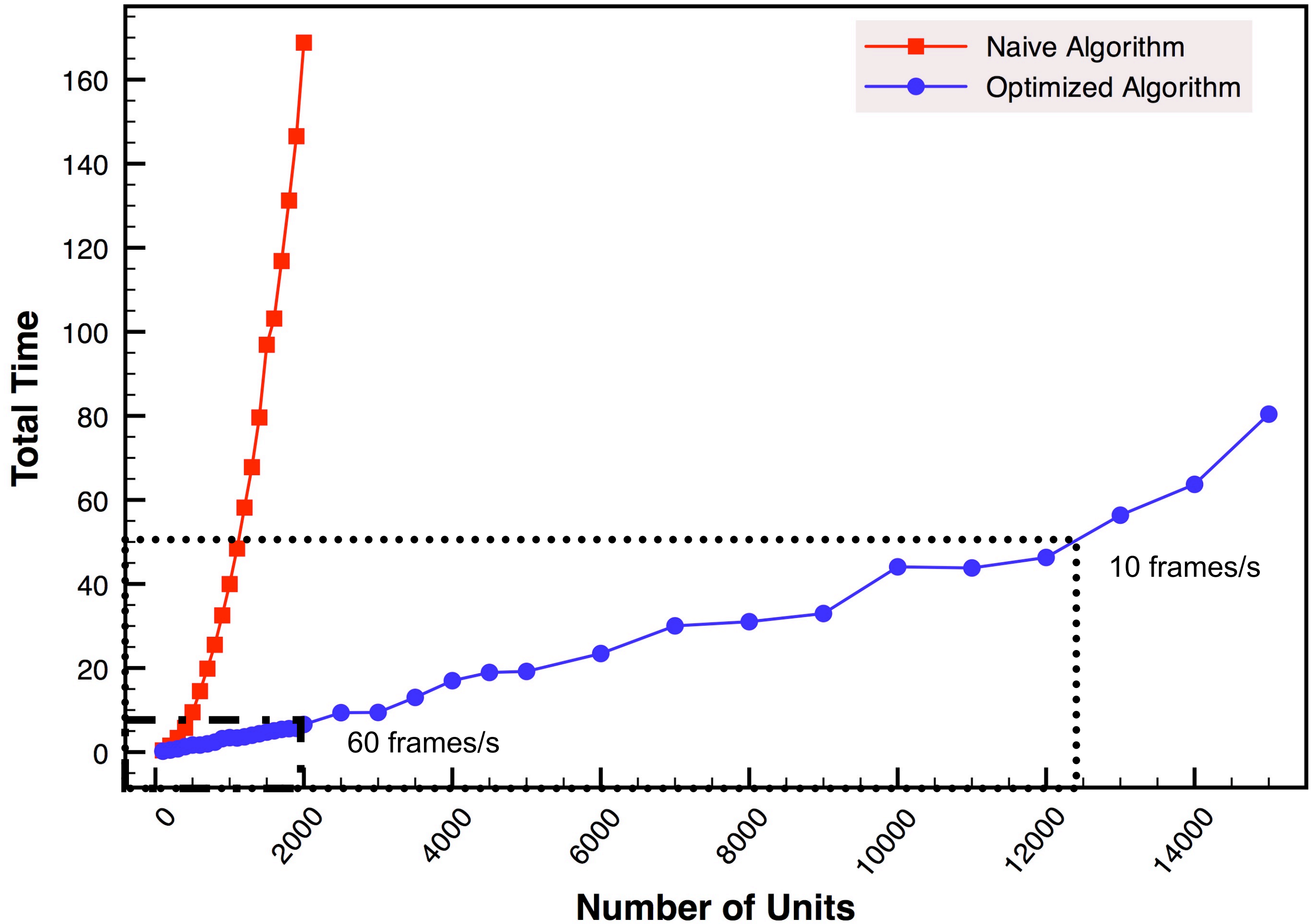
- Pluggable simulation comparing query plans
  1. Naive processing of aggregates.
  2. Use of indexing techniques.
     - The factor d from indexing techniques is d=1.
     - Performance is thus $O(n \log n)$ for each aggregate

- Hardware parameters:
  - 2Ghz Intel Core Duo running OS X in 1.5 GB RAM.
  - Compiled in C++ using GCC.

Experimental Results



Experimental Results



Future Work

- Working to implement SGL in XNA
  - Microsoft's new game development platform
  - Works on PC and XBox 360

- Lots of open problems:
  - Query processing
  - Query optimization
  - Further indexing methods
  - Implementation

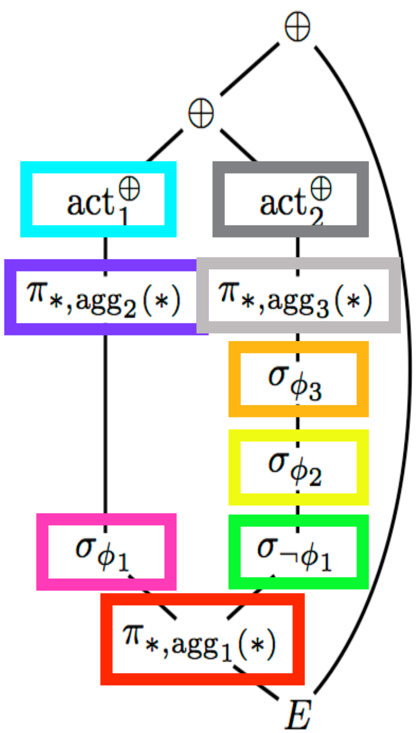
Final Words

- What was the success of databases?
  - Declarative specification vs. procedural retrieval
- This is the same program for simulations
  - Declarative behavior vs. procedural implementation
- Is a roadmap for multicore optimization
  - Declarative languages are highly parallelizable
  - What other problems can we apply them to?

Let’s Play!

Any questions?

Algebraic Optimization




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