Resource Entity Action: A Generalized Design Pattern for RTS games

Mohamed Abbadi, Francesco Di Giacomo, Renzo Orsini,
Aske Plaat, Pieter Spronck, and Giuseppe Maggiore
E-mail: {mabbadi,fdigiacomo,orsini}@dais.unive.it, {a.plaat,p.spronck}@uvt.nl,
maggiore.g@nhtv.nl

Università Ca' Foscari DAIS - Computer Science, Venice, Italy
Tilburg University, Netherlands
NHTV University of Applied Sciences, Breda, Netherlands

Abstract. In many Real-Time Strategy (RTS) games, players develop an army in real time, then attempt to take out one or more opponents. Despite the existence of basic similarities among the many different RTS games, engines of these games are often built ad hoc, and code re-use among different titles is minimal. We identify a design pattern called "Resource Entity Action" (REA) that abstracts the basic interactions that entities have with each other in most RTS games. This paper discusses the REA pattern and its language abstraction. We also discuss the implementation in the Casanova game programming language. Our analysis shows that the pattern forms a solid basis for a playable RTS game, and also that it achieves considerable gains in terms of lines of code and runtime efficiency. We conclude that the REA pattern is a suitable approach for the implementation of many RTS games.

1 Introduction

Real-time strategy (RTS) games have been highly popular for decades. As outlined by the ESA[1], RTS games are registering strong sales and a large number of play hours. Commercial RTS games are written by game developers of different backgrounds: from large studios to smaller independent developers of *indie games*. Indie developers [7] typically consist of small teams and their games are known for innovation [8], creativity [10] and artistic experimentation [3]. RTS games are also built as "serious games" [2], used for training and education, and as "research games" [4].

In general, the building of games is an expensive venture [5]. This is challenging in particular for indie developers and developers of serious and research games, who usually have access to few resources. They would benefit of cost-effective development methodologies for games, through the identification and automation/reuse of common patterns in games. Surprisingly, from a survey of game development research and literature, we noticed a lack of studies of

abstract patterns which characterize games, in particular RTS games. This motivates our research question: *to what extent can we capture the commonalities of RTS games in a re-usable design pattern?*

Section 2 discusses the essential elements of an RTS game. Section 3 specifies the *Resource Entity Action* (REA) design pattern [6] that captures these essential elements. Section 4 describes how the pattern is implemented as a language extension of the Casanova game programming language [9]. The language extension is purely declarative, using semantics that resemble SQL, providing an intuitive adoption for most programmers. We implemented the extension in a full-fledged RTS, which we discuss and analyze in Section 5.

2 Essential elements of RTS games

RTS are a variation of strategy games where two or more players achieve specific (often conflicting) objectives by performing actions simultaneously in real time. The typical elements which arise from this genre are *units* (characters, armies), *buildings*, *resources* and *battle statistics*. Players command units to perform different types of actions. These actions can affect several entities in the game world.

Units and buildings are the entities that players control to achieve their objectives. Units usually fight or harvest resources, while buildings may be used to create new units or research upgrades. Resources are gathered from the playing field and fuel the economy of the game entities. Battle statistics determine the offensive and defensive abilities of units in a fight. This taxonomy of the elements of an RTS game can be applied successfully to multiple games: Starcraft, C&C, and Age of Empires all feature units, buildings, resources, and battle statistics, amongst other elements.

In order to arrive at our design pattern we will now apply a simplification. Battle statistics can be interpreted as *resources*, as for instance: "the life of a unit is the cost for killing it, payable in attack power." We can also merge units and buildings together into a new category called *entities*. This leads us to a simpler view of an RTS as a game that is based on Resources, Entities and Actions:

1. Resources: numerical values in the battle and economic system of the game. In this group we find the *attack*, *defense*, and *life* patterns of entities. Resources also cover building materials and costs of production, deployment of units, development of new weapons, etc. (Resources are scalars.)
2. Entities: container for resources. They have physical properties and, as for the game logic, the difference among them is only the interactions. These interactions take place with resource exchanges through the actions. (Entities are vectors.)
3. Actions: resource flow among entities. Our model can be viewed as a directed weighted graph where the nodes are the entities, the weights are the amounts of exchanged resources, and the edges are the actions, that is, the elements which connect entities to one another. (Actions are transformation matrices.)

Next, we discuss how we model Resources, Entities and Actions.

3 The REA design pattern

In this section we will define a model for an algebra to show that the REA (Resource Entity Action) model can be reduced to a problem of linear algebra. We then show how games that use this model can be further simplified by linguistic constructs.

3.1 Action algebra

An action consists of a transfer of resources from a source entity to one or more target entities. We require that each entity has a resource vector, which contains the current amount of resources of the entity. The resource vector is sparse, since most actions involve only few resource types. An action is expressed by a transformation matrix A.

Consider a set of target entities $T = \{t_1, t_2, ..., t_n\}$, which are the targets of the action, and a source entity e. Each entity t_i (including the source entity type) has a resource vector $\mathbf{r_i} = (r_{i_1}, r_{i_2}, ..., r_{i_m})$. The source entity also has a transformation matrix A of size $m \times m$, which defines the interactions between all the resources of the source entity and all the resources of the target entities. We also consider an integrator dt which contains the time difference between the current frame and the previous one. We then compute $\mathbf{w_e} = (w_{e_1}, w_{e_2}, ..., w_{e_m}) = \mathbf{r_s} \times A \cdot dt$. From the definition of matrix multiplication, it immediately follows that each component of $\mathbf{w_e}$ represents how a resource will change by applying the effect of all the other resources to it. We compute the vector $\mathbf{r_i'} = \mathbf{r_i} + \mathbf{w_e}\ \forall e_i \in E$ which replaces the resource vector in each target entity.

For instance, consider the action of a spaceship entity using laser to damage (resource) an enemy spaceship (entity). This involves a vector resource of two elements: laser and life points. The action must transfer laser points to subtract from the enemy life points. Suppose that the vector resource of the targeting ship is $r_s = (20, 500)$ and the vector resource of the targeted ship is $r_t = (15, 1000)$. Let the transformation matrix be $A = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$ which means that the source entity will affect the life of the target with a negative number of laser points. Thus $w_e = r_s \times A \cdot dt = (20, 500) \times A \cdot dt = (0, -20) \cdot dt$. At this point, assuming $dt = 1$ second, we have $r_t' = r_t + w_e = (20, 1000) + (0, -20) \cdot dt = (20, 980)$.

3.2 A declarative language extension

We now describe a language extension that implements the REA design pattern and its associated algebra for the Casanova game programming language [9]. The language extension is purely declarative. Its semantics are described using the SQL query language, which has the advantage of familiarity to most programmers.

Implementing the action algebra is done using an abstract class which contains an abstract method which performs the action. Each action is a class which extends the previous abstract class and implements the abstract method. This

method will fetch the world looking for the information needed to find what entities are affected by the action execution. Each entity of the game will have a collection of actions it can perform, automatically run by Casanova.

To identify the set of target entities T given a source entity and its action, we create a new type definition called *action*. An action is a declarative construct which is used to describe not only the resource exchange between entities, but also what kinds of entities participate in the exchange. The resource exchange is based on *transfers* (Add, Subtract, and Set), while the target determination is based on *predicates*: we filter the game world entities depending on their types,attributes and radius (specifying the distance beyond which the action is not applied). Some actions, called threshold actions, are not continuous and make use of special predicates to delay the execution (Output) until certain conditions are met.

Using actions it is possible to specify an exchange of resources in a fully declarative manner, so that the developer does not have to rewrite similar pieces of code ad hoc for each action.

4 Action syntax and semantics

We now give the syntax and semantics of actions in Casanova. The grammar allows the definition of actions, which make up the body of spacial Casanova entities which act as placeholders for actions. When an entity contains such an action, the Casanova runtime will apply it to all appropriate targets.

4.1 Action Grammar Definition

We first provide a taxonomy for actions. We divide the actions into three kinds: (1) constant transfer actions, (2) mutable transfer actions, and (3) threshold actions.

Constant Transfer actions update the target fields with a constant value or a value taken from one of the source fields. The source field is not affected by the resource transfer. An example of a constant transfer action is a defense tower with infinite ammunition shooting an arrow at an infantry unit.

```
TARGET Infantry; RESTRICTION Owner <> Owner; RADIUS 1000.0;
    TRANSFER CONSTANT Life - ArrowDamage;
```

Mutable Transfer actions are used when the resource exchange transfers resources from the source entity to the target entity, or vice versa. An example of a mutable transfer action is a spaceship transferring minerals from its holds to a shipyard.

```
TARGET Shipyard; RESTRICTION Owner = Owner; RADIUS 150.0;
    TRANSFER MineralStash + Minerals;
```

Threshold actions follow the same transfer semantics as the previous two types of actions. In addition, they have a collection of threshold values and output

operations. The output operations are executed once when all the threshold values are reached. The threshold values are on fields belonging to the source entity. The output operations modify only fields of the source entity following the semantics of the transfer operations. An example for a threshold action is a worker building a town hall. When the `integrity` of the town hall reaches 100, a flag `completed` is set (which is one of its fields) which warns the system to replace the partially constructed building with the complete building.

```
TARGET ConstructionTownHall; RESTRICTION Owner = Owner;
   RADIUS 10.0; TRANSFER CONSTANT Integrity + 1.0; THRESHOLD
    Integrity = 100.0; OUTPUT Completed := true
```

Below we give a formal definition for the grammar instances presented in the examples above, using the extended Backus-Naur form. A *Casanova Entity* is an entity in the game world represented as a record; the special keyword `Self` is used to refer to the entity owning the action as one of its fields.

```
<Action> ::= TARGET <TARGET LIST> <RESTRICTION LIST> [<RADIUS
    CLAUSE>] <TRANSFER LIST>
   <INSERT LIST> [<THRESHOLD BLOCK>]
<TARGET LIST> ::= <ACTION ELEMENT>+
<ACTION ELEMENT> ::= Casanova Entity | Self
<RESTRICTION LIST> ::= {<RESTRICTION CLAUSE>}
<RESTRICTION CLAUSE> ::= RESTRICTION Boolean Expression of <
   SIMPLE PRED>
<SIMPLE PRED> ::= Self Casanova Entity Field (= | <>) Target
   Casanova Entity Field
<TRANSFER LIST> ::= {<TRANSFER CLAUSE>}
<TRANSFER CLAUSE> ::= (TRANSFER | TRANSFER CONSTANT)
(Target Casanova Entity Field) <Operator> ((Self Casanova
   Entity Field) | (Field Val)) [* Float Val]
<Operator> ::= + | - | :=
<RADIUS CLAUSE> ::= RADIUS (Float Val)
<INSERT LIST> ::= {<INSERT CLAUSE>}
<INSERT CLAUSE> ::= INSERT (Target Casanova Entity Field) ->
   (Self Casanova Entity Field List)
<THRESHOLD BLOCK> ::= <THRESHOLD CLAUSE>+
<OUTPUT CLAUSE>+
<THRESHOLD CLAUSE> ::= THRESHOLD
(Self Casanova Entity Field) Field Val
<OUTPUT CLAUSE> ::= OUTPUT
(Self Casanova Entity Field) <Operator> ((Self Casanova
   Entity Field) | (Field Val)) [* Float Val]
```

4.2 Formal semantic definition

Given the fact that actions resemble queries on entities, we specify their semantics as translation semantics to SQL. This allows us to leverage existing discussions on SQL correctness [12].

In defining our translation rules formally, we consider a set $T = \{t_1, t_2, ..., t_n\}$ of target types and a source entity type s. In all actions we select a subset of targets in each t_i on which to apply the action, using restriction conditions if any exist. After that we apply the resource transfer.

We assume that each entity type is represented by an SQL relation and that there exists a key attribute called **Id** for each relation. We now consider each of the three translation cases. In the translation rules we use notations inside the SQL code taken from the Backus-Naur form for grammar definitions. We also extend the SQL grammar with a global variable dt which is the time difference between the current and the last game frame. In this way the increment of the entity attribute values are proportional to the elapsed time. All types of actions evaluate the predicates in the restriction conditions and apply a filter to their targets. All targets further than the radius are automatically discarded when executing the action. The transfer predicates are executed immediately on all filtered targets.

For a **CONSTANT TRANSFER** we must update each target with the value in the source fields or constant values specified in the transfer clause. For simplicity, we assume that constant values are stored as attributes of the source entity.

Consider a set of resource attributes $A = \{a_{j_1}, a_{j_2}, ..., a_{j_m}\}$ of the source entity used to update the target t_i. To compute the contribution of all sources of the same type on the target t_i, we specify a relation of which the tuples represent the target id, followed by the total amount of resource a_{j_r} to transfer, called Σ_r:

Transfer				
ID	Σ_1	Σ_2	$\cdots$	Σ_m

The following SQL instruction implements the relation definition above:

```
SELECT   ti.id, SUM(s.aj1} AS Σ1,
         SUM(s.aj2} AS Σ2, ...,
         SUM(s.ajm} AS Σm

FROM     Target ti, Source s
WHERE    <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY ti.id
```

$\forall t_i \in T$ we update the target attributes $A' = \{a_{t_1}, a_{t_2}, ..., a_{t_m}\}$ using one of the target operators defined in the grammar (Set, Add, Subtract) with the attributes of the previous relation scheme.

```
WITH     Transfer AS(
                 SELECT   ti.id, SUM(s.aj1} AS Σ1,
                          SUM(s.aj2} AS Σ2, ...,
                          SUM(s.ajm} AS Σm)

FROM     Target ti, Source s
WHERE    [<RESTRICTION LIST>] [AND <RADIUS CLAUSE>]
```

```
GROUP BY ti.id)
UPDATE  Target ti
SET     ti.a_t1 = u.Σ1 | ti.a_t1 = ti.a_t1 + u.Σ1 * dt | ti.a_t1 =
ti.a_t1 - u.Σ1 * dt\
...
FROM    Transfer u
WHERE   u.id = ti.id
```

For a **MUTABLE TRANSFER** the field of the source involved in the resource transfer is updated depending on the applied transfer operator. The resource is subtracted from the source field and added to the target field proportionally to dt, or vice versa.

To translate this semantic rule we must first determine how many targets (if any) are affected by each source entity, in order to obtain the following relation scheme:

TotalTargets	
Source ID	*TargetCount*

The SQL code implementing the previous scheme is the following:

```
TotalTargets =
SELECT  s.id,COUNT(*) AS TargetCount
FROM    Source s, Target t1, Target t2,...,Target tn
WHERE   <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY s.id
HAVING  COUNT(*) > 0
```

$\forall t_i \in T$ we need to obtain a relation storing what target each of the source entities is affecting, including a count of affected targets, using the following relation scheme:

OutputSharing		
Source ID	*Target ID*	*Output Sharing*

This scheme is implemented by the following SQL code:

```
OutputSharing =
SELECT  *
FROM    TotalTargets c, SourceOutput c1
WHERE   c.s_id = c1.s_id
        AND SourceOutput =
                SELECT  s.id AS s_id,ti.id AS t_id
                FROM    Source s, Target ti
                WHERE   <RESTRICTION LIST> [AND <RADIUS
                    CLAUSE>]
        AND TotalTargets = [...]
```

Each target attribute receives an amount of resources equal to the total transferred resources divided by the number of targets. The complete SQL code to update the target t_i is the following:

```
WITH     Transfer AS(
         SELECT  t_i.id, SUM(s.a_j1 / o.TargetCount) AS Σ0,SUM(s.a_j2
             / o.TargetCount) AS Σ2,...,SUM(s.a_jm / o.
            TargetCount) AS Σm
         FROM    Source s, Target t_i,OutputSharing o
         WHERE   OutputSharing = [...] AND s.id = o.s_id AND t
            .id = o.t_id)
         GROUP BY t_i.id
UPDATE   Target t_i
SET      t_i.a_t1 = u.Σ1 |t_i.a_t1 = t_i.a_t1 + u.Σ1 * dt |t_i.a_t1 = t_i.a_t1 - u.
    Σ1 * dt
...
FROM     Transfer u
WHERE    t_i.id = u.id
```

To update the Source relation we use a relation similar to the one use to update the target, but this time there is no need to save the count of the affected targets.

```
WITH     TotalTransfer AS(
         SELECT  s.id,s.a_j1,s.a_j2,...,s.a_jm
         FROM    Source s, Target t_1,...,Target t_n
         WHERE   <RESTRICTION LIST>
                 [AND <RADIUS CLAUSE>]
         GROUP BY        s.id,s.a_j1,s.a_j2,...,s.a_jm
         HAVING  COUNT(*) > 0)

UPDATE   Source s
SET      s.a_j1 = s.a_j1 - s.a_j1 * dt|s.a_j1 = s.a_j1 + s.a_j1 * dt
...
FROM     TotalTansfer u
WHERE    s.id = u.id
```

The **THRESHOLD** action is defined as the previous two types, i.e., it has a resource transfer definition which is always executed, and a set of threshold conditions that, if met, activate the Output operations, which are always towards the source entity. The attributes of the source entity affected by Output operations are updated with constant values, or with values from other attributes in the source entity. In the latter case the transfer is treated as for the mutable transfer case.

Consider a set of updating attributes $U = \{a_{k_1}, a_{k_2}, ..., a_{k_l}\}$ and a set of attributes to be updated $U' = \{a_{s_1}, a_{s_2}, ..., a_{s_l}\}$ in the output operation. We first check that all the conditions in the threshold clauses are met, then we update the attributes in the source entity appropriately.

```
WITH     TotalOutput AS(
         SELECT  s.id,s.a_k1,s.a_k2,...,s.a_kl
         FROM    Source s
         WHERE   <THRESHOLD CLAUSE 1>
```

```
                 [AND <THRESHOLD CLAUSE 2>]
                 .
                 .
                 .
                 [AND <THRESHOLD CLAUSE l>])
UPDATE   Source s
SET      s.a_s1 = o.a_k1|s.a_s1 = (s.a_s1 + o.a_k1)*dt; o.a_k1 = o.a_k1 -
    o.a_k1*dt|s.a_s1 = (s.a_s1 - o.a_k1)*dt; o.a_k1 = o.a_k1 + o.a_k1*dt
...
FROM     TotalOutput o
WHERE    s.id = o.id
```

5 Case study

We now present an RTS game we used as a case study, created with Casanova, and the benchmarks that test the action implementation. In the game players must conquer a star system made up of various planets. Each planet builds fleets which are used to fight the fleets of the other players and to conquer more planets. A planet is conquered when a fleet of a player is near it and no other enemy fleet is defending it.

5.1 Case study

Three actions are required in this game: The first action, called `Fight Action`, defines how a fleet fights enemy fleets in range. The fight action subtracts $0.5 \cdot dt$ `life` points from the in-range enemy fleet during every frame (action tick).

```
Fleet = {Position: Rule Vector2;FightAction: FightAction;
    Owner: Ref Player;Life: Var float32;Fight: FightAction }
```

The Fight Action is defined as follows:

```
FightAction = TARGET Fleet; RESTRICTION Owner <> Owner;
    RADIUS 150.0; TRANSFER CONSTANT Life - 0.5;
```

The target is an entity of which the type is **Fleet**. The condition to execute the action is that the fleet must be an enemy (i.e., not the player). The `attack range` is 150 units of distance. 0.5 `life` points are subtracted for every attack.

The second action is called `BuildAction`. It allows a planet to create a ship. In order to build a ship, a planet must gather 10 mineral units. Each planet has a field called `GatherSpeed` which determines how fast it gathers minerals. Every tick the planet's mineral stash is increased by this amount. This action is a threshold action where the threshold value is the minerals of the planet. As soon as the threshold value is reached, we set the field `NewFleet` to TRUE (it is used by the engine to create a new fleet), and `Minerals` to 0 to reset the counter. The planet and its actions are:

```
Planet = {Position: Vector2;Owner: Rule Ref Player;NewFleet:
    Rule bool;BuildAction:BuildAction;
    EnemyOrbitingFleetsAction : EnemyOrbitingFleetsAction;
    GatherSpeed: float32;Minerals: Var float32 }
```

```
BuildAction =
TARGET Self; TRANSFER CONSTANT Minerals + GatherSpeed;
    THRESHOLD Minerals 10.0; OUTPUT NewFleet := true; OUTPUT
    Minerals := 0.0
```

A Casanova rule is appointed to read the value of NewFleet and, if it is true, it spawns a new fleet.

The third action is required to check if a planet can be conquered by a fleet. A fleet can conquer a planet if there is no enemy fleet near it and if it is sufficiently close. Thus the action definition is the following:

```
EnemyOrbitingFleetsAction =
TARGET Fleet; RESTRICTION Owner Not Eq Owner; RADIUS 25.0;
    INSERT Owner -> EnemyOrbitingFleets
```

The action will add an enemy fleet close enough to change the owner of the planet.

5.2 Evaluation

We evaluated the performance of our approach with the case study, and two extra examples: an asteroid shooter, and an expanded version of the case study with more complex rules. All were implemented in casanova. Table 1 shows a code length comparison between the REA implementation and standard Casanova rules for all three.

We note that in games with basic dynamics the code saving is low, due to the fact that there are few repeated patterns. The advantage of using REA becomes evident in a game with actions involving many types of targets, such as the expanded case study. Furthermore, we managed to drastically increase the performance of the game logic: as Figure 1 shows, using REA (labeled "with actions") results in a speedup factor of 6 to 25, due to automated optimizations in the query evaluation. We also note that our implementation is flexible and general since it is possible to use actions to express a behavior, such as a projectile collision.

6 Conclusions

In this paper we propose the Resource Entity Action (REA) pattern to define RTS games. Use of this pattern should protect developers from writing and rewriting large amounts of boilerplate code. The paper presents:

Table 1: CS (case study),Asteroid Shooter and Epanded CS code length.

	Game Entities	Rules	Actions	Total
CS with REA	41	71	19	131
CS without REA	40	90	0	130
Asteroid shooter with REA	33	33	6	72
Asteroid Shooter without REA	34	44	0	78
Extended CS with REA	135	138	40	313
Extended CS without REA	135	328	0	463

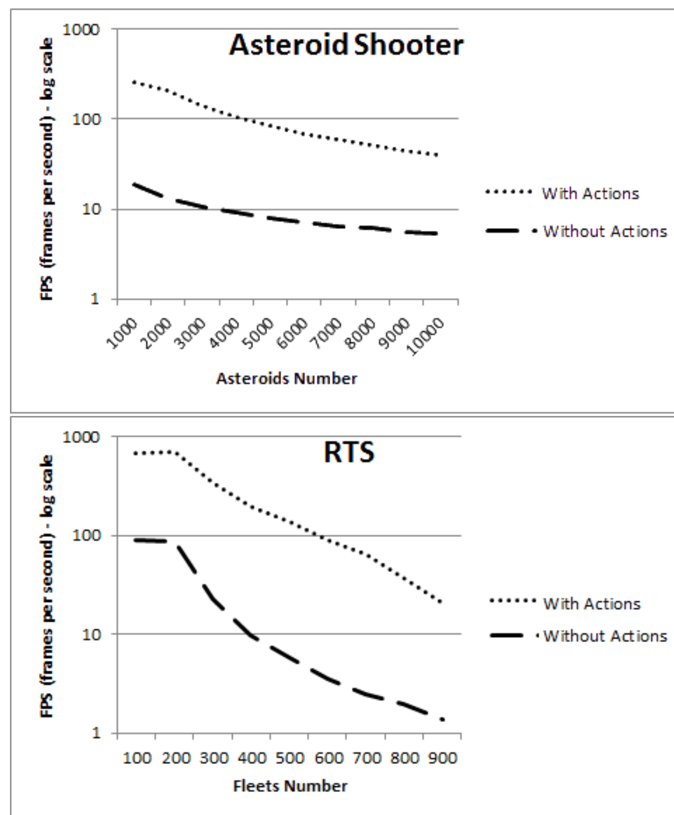

Fig. 1: *Frame rate as a function of numbers of entities*

- the **REA design pattern** for making RTS games which reduces the interaction among entities to a dynamic exchange of resources;
- an expressive, declarative, high performance **language extension** to Casanova, with an appropriate grammar with new syntax and semantics resembling SQL;
- an evaluation with three examples which provides evidence for an increase in programming efficiency using REA; and
- an evaluation that shows an increase in run time efficiency of 6 to 25 times for the Casanova language, using a native code compiler/opimizer.

In future work, we estimate that even better results can be obtained with an actual access plan optimizer that increases the performance when exploring the structure of both the action query and the entity structure. Given the significant results on position indexing, the chance of defining multi-attribute indexes would increase the performance. Moreover, a system like F# quotations [11] may be used to increase the expressiveness of the actions.

Bibliography

[1] Essential Facts About the Computer and Video Game Industry 2011, 2011.

[2] Clark Aldrich. *The Complete Guide to Simulations and Serious Games: How the Most Valuable Content Will Be Created in the Age Beyond Gutenberg to Google*. Pfeiffer & Co, 2009.

[3] Erik Andersen, Yun-En Liu, Rich Snider, Roy Szeto, and Zoran Popović. Placing a value on aesthetics in online casual games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1275–1278, New York, NY, USA, 2011. ACM.

[4] Michael Buro. Real-time strategy gaines: a new ai research challenge. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1534–1535. Morgan Kaufmann Publishers Inc., 2003.

[5] Michael Buro and Timothy Furtak. On the development of a free rts game engine. In *GameOn'NA Conference*, pages 23–27. Montreal, 2005.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] Juan Gril. The state of indie gaming, 4 2008.

[8] Johan Kristiansson. Interview starbreeze studios johan kristiansson. http://www.develop-online.net/features/478/Interview-Starbreeze-Studios-Johan-Kristiansson, 5 2009.

[9] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '12, pages 287–292, New York, NY, USA, 2012. ACM.

[10] David Michael. *Indie Game Development Survival Guide (Charles River Media Game Development)*. Charles River Media, 2003.

[11] Don Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[12] Günter von Bültzingsloewen. Translating and optimizing sql queries having aggregates. *Proc. 13th Int. Con. VLDB*, pages 235–243, 1987.