A gameplay loops formal language

Yannick Francillette, Abdelkader Gouaich, Nadia Hocine, Julien Pons

A Gameplay Loops Formal Language

Yannick Francillette, Abdelkader Gouaïch, Nadia Hocine and Julien Pons
LIRMM
University of Montpellier, France, CNRS
Email: {yannick.francillette, gouaich, nadia.hocine, julien.pons }@lirmm.fr

***Abstract*—In this paper we present an approach of procedural game content generation that focuses on a gameplay loops formal language (GLFL). In fact, during an iterative game design process, game designers suggest modifications that often require high development costs. The proposed language and its operational semantic allow reducing the gap between game designers' requirement and game developers' needs, enhancing therefore video games productivity. Using gameplay loops concept for game content generation offers a low cost solution to adjust game challenges, objectives and rewards in video games. A pilot experiment have been conducted to study the impact of this approach on game development.**

I. Introduction

Game development process has drastically evolved since the beginning of the video game industry in the late of 70's. Different factors have since influenced video game development process due to the growing evolution of:

- technical, narrative and aesthetic aspects of games. Game developers continuously try to go beyond existing solution in order to improve video games quality;
- technologies like internet, geolocalization, motion recognition and so on. These technologies offer interesting features but also require additional skills for game developers;
- size of teams in game projects that have been accordingly increased to reach for instance hundreds of members for some AAA games (games with high budget);
- financial risks management techniques as game production has become a cultural industry. Financial risks should be studied at the start of game projects and stakeholders expect financial benefits from any production.

Despite these changes in game production, the objective of game development process still devoted to create a game that provides an interesting and entertaining player experience while reducing development costs. Different practices have been emerged while trying to answer these issues. For instance Novak [1] suggests the following generic stages of game development process:

- **Concept**: an initial idea of the game is defined using for instance a brainstorming session;
- **Pre production**: a game design document is written to specify the gameplay and determine initial game levels to produce the first game prototype;
- **Prototype**: an implementation step of a game is carried out. The goal of the prototype is to check the playability of the game and fun aspects;
- **Production**: a heavy phase of actual game development with the production of all game levels, graphics, sounds and so on;
- **Localization**: a step that consists in adjusting the game to regions and countries' local requirements;
- α**-testing**: a test of the game from beginning to the end;
- β**-testing**: corrections of bug and tuning to specific platforms;
- **Gold**: the game distribution stage;
- **Post production**: the final step that may require updates, community management and so on.

Our context within this paper can be situated between pre production and prototype stages of game development process. Our objective in this work is to provide means to rapidly and iteratively design, implement and evaluate game prototypes. We focus in our approach on a formal language of objective-challenge-reward (OCR) gameplay loops [2]. This approach helps game designers and developers to have a common language to rapidly: define gameplay loops, implement the prototype, evaluate and adjust gameplay loops for the next game development iteration. Besides, development costs associated with this iterative process are reduced since most of elements can be reused for a future development cycle.

The rest of the paper is organized as follows: In the next section we introduce gameplay loops concept while discussing our motivation in this paper. Then, we describe in section III related work concerning existing technique to model the gameplay. After that we present in section IV the gameplay loops formal language and its semantics. Next, section VI presents a pilot experiment that have been conducted in order to evaluate the impact of our solution in game development. Finally, we conclude this paper by discussing the proposed language features and our perspectives.

II. Motivation

The advantage of the iterative process of game development is the experimental and evaluation scheme. In fact, it is difficult to know in advance players' reaction and how player experience metrics [3] will be affected by design decisions. To assess these effects, an evaluation is required to adjust design decisions until reaching desired player experience. Obviously, the iterative approach is possible only when the iteration costs can be reduced as the game prototype is not rebuilt from scratch.

In addition, level design that targets classical gaming environment, such as consoles and PC, makes often the assumption

that the player context does not influence the gameplay. This assumption does not hold in some environments such as mobile and geo-localized games. In fact, the position of the player and his/her geographical context become fundamental parameters that determine the gameplay content.

In order to overcome these challenges, a rational game design method can be used to explicitly model the structure of a game and to understand the interplay between game structure elements and player experience. The philosophy of this approach is somehow related to structuralism movement [4]. In fact, video game aspects and elements are decomposed, identified and studied during the interaction to understand how they affect the player experience.

Being inspired by the rational game design approach, the Objective Challenge Rewards (OCR) level design method has been used to creates hierarchical game cycles [2]. This method identifies the concept of OCR gameplay loop where:

- The objective is the state of a game that steers the behavior of the player;
- The challenge represents all elements that the player has to face and overcome to achieve the objective;
- Finally, the reward is what the player gets when achieving the objective.

Besides, OCR loops can be hierarchically composed to create different levels from atomic micro gameplay loops to more complex gameplay loops.

OCR gameplay loops are coherent with the iterative and rational game design approaches. In fact, adjusting player experience metrics can be carried out by modifying OCR loops. However, the OCR method only provides an informal description and general guidelines to structure a gameplay. Our objective in this paper is to formally describe a language and its semantics to handle OCR loops. In this regards, OCR gameplay loops are considered as gameplay components allowing a procedural game level generation.

III. Related Work

From game design perspective, the gameplay is composed of different elements allowing to describe the overall game experience, including for instance players' activities, mechanics and feeling.

As game design document usually provides only a textual description of game elements, the programmer has to interpret these textual elements and transform them into specifications before the implementation. This can create ambiguities and miss-interpretations.

Different methods have been introduced to easily avoid ambiguities of natural languages through game concepts modelling in a formal or informal way.

The first approach focuses on describing game rules as core element representing a gameplay. For example Frasca [5] considers three kind of rules constituting the gameplay: (i) manipulation rules that defines what players can do in the game (ii) goal rules that defines success and failure conditions and (iii) meta-rules that defines how game rules can be modified while the game-player interaction. As for Djaouti et al [6], rules are considered as game bricks that can be defined in a similar manner. Rules are determined by using a common template which represents the game behavior.

As described in [7] a gameplay is composed of rules constituted of (i) pre-conditions to execute rules and (ii) post-conditions which are the consequences of rule execution. In this model, three kind of rules are defined: (i) *action rules* are actions which can directly be performed by a player thought the input controller, (ii) *internal rules* are events which can occur during the game and (iii) *goal rules* are events which lead to the end of the game, victory, failure or draw.

Using a rule-based description can be a useful implicit model of the gameplay. Nevertheless, rules definition have seldom clear semantics regarding natural language description. In addition, updating game challenges, objectives and rewards requires to identify all rules and entities that constitute these gameplay components.

In [8] Rollings et al. propose a model based on tokens. A token is an element which represents a game entity and has a set of states. Using this model, game designers have to describe a set of tokens, define their states and then implement an interaction matrix. The interaction matrix defines relationships between two states of a token. This model focuses on game entities and their actions, and it allows to get a quick overview of the elements composing the game mechanics. However, this model does not highlight players activities in the game and it does not provide an easy method to modify challenges. For example, to increase or decrease the difficulty of the *Pacman* video game model, the method requires studying the interaction matrix to define which element constitute the challenge.

The second way to describe a gameplay is to consider it as a behaviour process based on a mathematical structure or model. The objective is to provide game designers with a formal and precise rule definition without the ambiguity of natural language. For example, Gronvogel [9] suggests a mathematical abstract control system to describe the different game behaviour regarding the player interaction. Furthermore Bura [10] and Dormans [11] use Petri Nets diagrams to specify game rules. Nodes and links represent concrete game design concepts such as atomic transitions, texts, resource and flows. According to Araujo et al. [12] who also propose a variant of this approach, when the game is growing in terms of complexity, it is difficult for a game designer to read the game logic.

In [13] Smith et al. propose an event calculus based (EC) model. EC is a logic-based formalism for representing and reasoning about events and their effects. This framework focus on modelling game rules and does not explicitly highlight challenges elements and game objectives. In addition, game designers without a solid knowledge of mathematical notation would have difficulties to express their designs.

Finally, procedural content generation techniques have also been interested in modelling the dynamic of the game content including for instance game levels and quests [14]. For instance Doran et al, propose a method to model quest in order to

develop a quest generator [15]. A quest can be defined as a task where the player is challenged to complete some objectives in order to get a reward. Quests are modelled by a tree of atomic actions and the root represents the main objective. In order to achieve the latter, the player has to overcome all sons of this node. Like previous work, quests-based models do not offer an explicit way to modify the challenge of atomic actions.

Our objective in this paper is to suggest a gameplay representation or model that can be useful to both game designers and developers. The purpose of our model is to fill the gap between game designers and game developers needs.

IV. Gameplay Components

A. Introduction

Atomic gameplay components are the elementary building blocks of the game. In fact, they are set within the scene to define the player objectives, allowing different levels of challenge. The player can also be considered as a rational agent seeking to have either explicit or implicit rewards. By the latter, we mean all psychological attributes that define the player experience such as fun, control, flow, excitement and so on [3]. Explicit rewards, on the other hand, are attributes defined within the game world such as killed enemies, gathered items and so on. These elements are provided to the player when he/she achieves the objective of a gameplay component and could be used to assess player progression within the game.

Usually, it is the responsibility of the game designer to provide such a set of gameplay components and check that the combination of these elements is consistent with the targeted players' experience metrics.

Our proposal is to consider this set of atomic gameplay component as an alphabet to build a broader language of gameplay components.

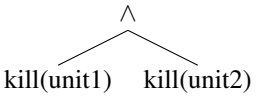
Elements of these language are built using specific operators with a defined semantics. The goal of this section is to formally present the construction of such a language of gameplay components.

The advantage of this approach is that the game designer focus his/her attention on providing the atomic GCs. The combination of these elements is performed in later phases to create game levels that optimize player experience metrics.

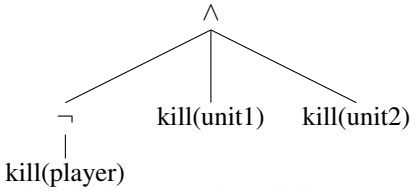
Furthermore, creating new situations using the GC language does not require the game to be reprogrammed from scratch. In fact, thanks to the operational semantics of operators it is possible to implement a virtual machine to interpret all elements of the language.

B. Game loop operators

Operators are provided to combine existing elements and express infinite situations using only finite elements. As an example, let us consider a basic shooting game. For this game the game designer has provided only one atomic GC "kill (unit)"' The objective of this gameplay component is simply to kill the unit put in parameter. The level designer can build for an instance a situation where the player has to kill 2 units. The simplest way to express this, is to introduce an (and) $\wedge$ operator that simply composes two sub components and run them simultaneously (the complete semantics of this operator is provided in V):



After a brainstorming session, a team member can suggest that something is missing in the game and that the player has as an objective to survive. To express this new situation, a (negate) $\neg$ operator can be used combined by the the existing {}"kill" gameplay component. The new expression would be:



This expression is interpreted as follows: to accomplish completely his/her mission within the level, the player has to survive, and to kill unit1 and unit2. So the GCs interpreter is able to check if the player: *(i)* is still trying to achieve his/her game objectives, *(ii)* has already achieved gameloop objective, or *(iii)* game objectives cannot be achieved.

This approach makes us introduce a paradigm for level design in which a meta objective of a player is to resolve GCs. Thanks to the explicit language of GCs, it is possible to either: statically build and assign gameplay components to a level, or use procedural generation at runtime by taking into account, for instance, player profile and context.

The previous example has introduced two operators. The next paragraphs informally present all operators that we have defined.

1) Atomic game loop: This is a utilitarian operator that takes as an input an atomic GC provided by the game design phase and makes it an element of the formal language.

2) Parallal And: The idea behind the parallel and operator is that all sub-components are running simultaneously; when at least one sub-component states that its objective can never be achieved then the overall expression is considered unachievable.

3) Parallal Or: The idea behind the parallel or operator is that all sub-components are running simultaneously; when at least one sub-component states that its objective has been completed then the objective of the overall expression is considered as achieved.

4) Sequential And: By contrast to parallel and, the evaluation of sub-components is ordered and sequential from the left to the right. When a sub-component states that its objective has been attained then the next sub-component is considered running and evaluated. This process is stopped when a sub-component states that its objective cannot be achieved. When all sub-components state that their objectives have been achieved then the overall expression is considered achieved. The overall expression is considered not achievable when at least one sub-component is not achievable.

5) Sequential Or: By contrast to parallel or, the evaluation of sub-components is ordered and sequential from the left to the right. When a sub-component states that its objective is not achievable then the next sub-component is considered running and evaluated.

This process is stopped when a sub-component states that its objective has been achieved. When all sub-components state that their objectives cannot be achieved then the overall expression is not achievable. The overall expression is considered achieved when at least one sub-component is achieved.

6) Not: This operator simply inverts statements of the sub-component. So, when the sub-component states that is has been achieved then the expression states that the objective cannot be achieved. Similarly, when the sub-component states that the objective cannot be achieved then the expression states that the objective has been achieved.

7) Continuation (C_f): The continuation operator allows to continue with another GC when the current GC has finished.The new gameplay component is generated by the function f

8) First: The idea behind the first operator is that the evaluation of the overall expression is equal to the evaluation of the first ended GC.

9) Last: The idea behind the last operator is that the evaluation of the overall expression is equal to the evaluation of the last ended GC.

10) If Then Else: The if then else operator is a tri-any operator. This operator evaluate the state of a GC and if the ended state if this component is equal to a particular state, the evaluation of this expression is equal to the evaluation of a second GC, else the evaluation of this expression is equal to the evaluation of a third GC.

V. Formal description

To build a formal language of gameplay components, we distinguish two phases : *(i)* defining the set of well formed formulas (wff) and *(ii)* defining the meaning or semantics of these wff. Section V-A presents definition of the formal language while sections V-A1 and V-A2 present the semantics of the language.

A. The formal langage

The GC formal language $\mathcal{G}$ is given by $< A, \Omega, N >$ where:

- A represents a finite set of atomic gameplay loops identified during the game design phase;
- Ω represents the set of operators. It is usual to decompse this set as a union $\bigcup_{n\in\mathbb{N}} \Omega_n$ where each set Ω_n contains operators taking exactly n arguments;
- N represents a set of functions $\mathcal{G} \to \mathcal{G}$ that generate GC.

Definition 1 (GC language)**.** *The formal language $\mathcal{G}$ is defined inductively as follow:*

- $A \subset \mathcal{G}$*, all atomic gameplay loops are well formed formulas (wff)*
- *if $P_1 \dots P_n \in \mathcal{G}$ are wff then $\forall \omega \in \Omega_n$: $\omega(P_1 \dots P_n)$ is also wff*

Within the context of this article, we have identified the following generic operators for GC:

- $\Omega_o = \{\epsilon^+, \epsilon^-\}$, this set contains constants respectively, achieved GC and not achievable GC.
- $\Omega_1 = \{\neg, C_{f\in N}\}$, this set contains two unary operators: not and continuation
- $\Omega_2 = \{\wedge, \vee, \overrightarrow{\wedge}, \overrightarrow{\vee}, F, L, If\}$, this set contains the following operators: parallal and, parrallal or, sequential and, sequential or, fist, last and if then else .

1) Evaluation of gameplay components: To evaluate GCs, we need to define two basic concepts : a game state and an algebra used for the semantics of GCs.

Each game owns a state containing all necessary information to evaluate expressions. The game state should contain enough information to evaluate weither a GC objective has been reached (noted $\top$) or cannot be reached ($\bot$). When neither of these statement are applicable then the answer will be unknown (noted '?')

Game states naturally owns a partial order relationship modelling precedence. So, $S \to S'$ means that game state has evolved to S' from S in a single update cycle. This notation is generalized to $S \xrightarrow{n} S'$ and means that S' has been reached from S in n update cycles.

The evaluation of GCs is performed using an evaluation function noted $\mathcal{E}[\![g]\!]\mathtt{S}$ where g represents the expression to evaluate and S the game state in which the evaluation is performed. As said previously, depending on the game state the evaluation of an expression returns three possible values representing the following situations:

- The objective of the game loop has been successfuly reached;
- The objective of the game cannot be achieved;
- Neither of previous claims can be stated on the game loop.

The three previous situations are represented respectively by the following symbols : $\top$, $\bot$ and ?. The three previous constants have not to be confused with classical logic operators of truth since their semantics is different. For instance, the $\bot$ constant is very different from classical logic false operator. In fact, we are not seeking if the GC has not been reached in the current state but if the GC will never be reached in all future states. This is to know if failure conditions has been met so the player will never reach a state where he/her will resolve the GC objective. More formally this can be defined as an axiom for our framework.

Axiom (monotonicity)**.** *We state the following axiom to unsure that evaluation of GC when known cannot be changed:*

$$\forall g \in \mathcal{G}, \forall S, S' \in \mathcal{S} : \left.\begin{array}{c} \mathcal{E}[\![g]\!]\mathtt{S} \neq ? \\ S \xrightarrow{n} S' \end{array}\right\} \implies \mathcal{E}[\![g]\!]\mathtt{S} = \mathcal{E}[\![g]\!]\mathtt{S}'$$

Definition 2 (GC eval)**.** *The evaluation of a GC in a state S, is defined as a function:*

$$\begin{array}{rccl} \mathcal{E}[\![-]\!]\mathtt{S} : & \mathcal{G} & \to & \{\top, \bot, ?\} \\ & g & \mapsto & \mathcal{E}[\![g]\!]\mathtt{S} \end{array}$$

Futhermore, the set $\{\top, \bot, ?\}$ is attached with . (dot) , $+$ and $\neg$ operations that behave similarly as in boolean algebra for $\top$ and $\bot$. They are only extended as follow to handle the $?$ constant:

$$\begin{aligned} \top \wedge ? &= ? \\ \top \vee ? &= \top \\ \bot \wedge ? &= \bot \\ \bot \vee ? &= ? \\ \neg ? &= ? \end{aligned}$$

a) Evaluation rules of GC elements:

$$\begin{aligned} \forall g \in A, \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} &\in \{\top, \bot, ?\} \\ \mathcal{E}[\![\epsilon^+]\!]\mathtt{S} &= \top \\ \mathcal{E}[\![\epsilon^-]\!]\mathtt{S} &= \bot \\ \mathcal{E}[\![\mathtt{g} \ \wedge \ \mathtt{h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S}.\mathcal{E}[\![\mathtt{h}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{g} \ \vee \ \mathtt{h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} + \mathcal{E}[\![\mathtt{h}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{g} \ \overrightarrow{\wedge} \ \mathtt{h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{g} \ \overrightarrow{\vee} \ \mathtt{h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \\ \mathcal{E}[\![C_f \ \mathtt{g}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \\ \mathcal{E}[\![\neg \ \mathtt{g}]\!]\mathtt{S} &= \neg\mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{F \ g, \ h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} + \mathcal{E}[\![\mathtt{h}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{L \ g, \ h}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} + \mathcal{E}[\![\mathtt{h}]\!]\mathtt{S} \\ \mathcal{E}[\![\mathtt{If \ g \ then \ h1 \ else \ h2}]\!]\mathtt{S} &= \mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \end{aligned}$$

2) Reduction rules:: Having a starting GC tree the player interacts with the game and makes its state evolve at each update cycle. Consequently, the GC tree has to be updated in order to remove all unnecessary information. In fact, the axiom stated previously ensures that the evaluation of GC element when known cannot change in future states. This introduces the process of simplification GC.

Having evaluations of GC it is possible to rewrite a new GC tree according to the semantics that we have described informally in section . The following section presents more formally the simplification rules and presents the generic algorithm used within the game loop update cycle to manage GCs during the game.

a) Idempotent: This rules says that when an atomic GC state is not known, then the GC remains unchanged.

$$\frac{g \in A \quad \mathcal{E}[\![\mathtt{g}]\!]\mathtt{E} = ?}{g \to g} \tag{1}$$

b) Substituion: This rules says that any element can be replaced by its simplest form.

$$\frac{g_i \to g_i'}{\omega_n(g_0, \ldots, g_i, \ldots, g_n) \to \omega_n(g_0, \ldots, g_i', \ldots, g_n)} \tag{2}$$

c) Constants: These rules say that GC are replaced by success or failure constants according to their evaluations:

$$\frac{g \in A \quad \mathcal{E}[\![\mathtt{g}]\!]\mathtt{E} = \top}{g \to \epsilon^+} \qquad \frac{g \in A \quad \mathcal{E}[\![\mathtt{g}]\!]\mathtt{E} = \bot}{g \to \epsilon^-} \tag{3}$$

d) Not operator: The negation of a failure will rewritten as a success and a negation of a success will be rewritten as a failure.

$$\text{(not)}\frac{}{\neg\epsilon^- \to \epsilon^+} \qquad \text{(not)}\frac{}{\neg\epsilon^+ \to \epsilon^-} \tag{4}$$

e) And operator: The and operator can be simplified if a sub components is already known to have succeeded. However, if a component fails then the expression is entirely replaced by a failure.

$$\frac{}{\epsilon^+ \wedge g \to g} \qquad \frac{}{\epsilon^- \wedge g \to \epsilon^-} \tag{5}$$

f) Or operator: When a sub component successes then the overall expression is replaced by a success. The expression is simplified if we already know that a subcomponent has failed.

$$\frac{}{\epsilon^+ \vee g \to \epsilon^+} \qquad \frac{}{\epsilon^- \vee g \to g} \tag{6}$$

g) Sequential and operator: If the first sub component of this operator has succeeded then the overall expression is replaced by the next one. On the other hand, if the head component fails then the overall expression is replaced by a failure.

$$\frac{}{\epsilon^+ \overrightarrow{\wedge} g \to g} \qquad \frac{}{\epsilon^- \overrightarrow{\wedge} g \to \epsilon^-} \tag{7}$$

h) Sequential or operator: If the first sub component of this operator has failed then the everall expression is replaced by the next one. On the other hand, if the head component succeed then the overall expression is replaced by a success.

$$\frac{}{\epsilon^- \overrightarrow{\vee} g \to g} \qquad \frac{}{\epsilon^+ \overrightarrow{\vee} g \to \epsilon^+} \tag{8}$$

i) Continue operator: If the state of the sub component of this operator is known then the expression is replaced with the newly generated using the function f GC.

$$\frac{\mathcal{E}[\![\mathtt{g}]\!]\mathtt{S} \neq ?}{C_f g \to f(g)} \tag{9}$$

j) First: When a sub component ends then the overall expression is replaced by the end state of this component.

$$\frac{}{F\epsilon^-, g \to \epsilon^-} \qquad \frac{}{F\epsilon^+, g \to \epsilon^+} \tag{10}$$

k) Last: The last operator can be simplified if a sub components is ended. The expression is replaced by the remaining component.

$$\frac{}{L\epsilon^-, g \to g} \qquad \frac{}{L\epsilon^+, g \to g} \tag{11}$$

l) If then else: If the first sub component of this operator has succeeded then the overall expression is replaced by the second sub component. If the first sub component of this operator has failed then the overall expression is replaced by the last sub component.

$$\frac{if\epsilon^{+}then, g1else, g2}{g \rightarrow g1} \quad \frac{if\epsilon^{-}then, g1else, g2}{g \rightarrow g2} \tag{12}$$

B. The GC interpreter

```
GC_interpreter_update()
{
GC gc = getMainGC();
gc = gc.simlify(getCurrentGameState());
setMainGC(gc);
if(gc == epsilonSuccess || gc == epsilonFail)
{
        gameOver();
}
}
```

The behaviour of the interpreter is quite simple: at each update cycle the main GC element is simplified. When the main GC becomes either ϵ^{+} or ϵ^{-} then the game level is terminated in respectively success or failure situations.

C. The GC state machine

The GC component proposes to link to the objective a set of challenges and a set of rewards (or a set of penalties if the player do not achieve the objective) according to the OCR model. We propose to represent challenges and rewards by the execution of particular scripts. The figure 1 shows the state machine of the gameplay component.

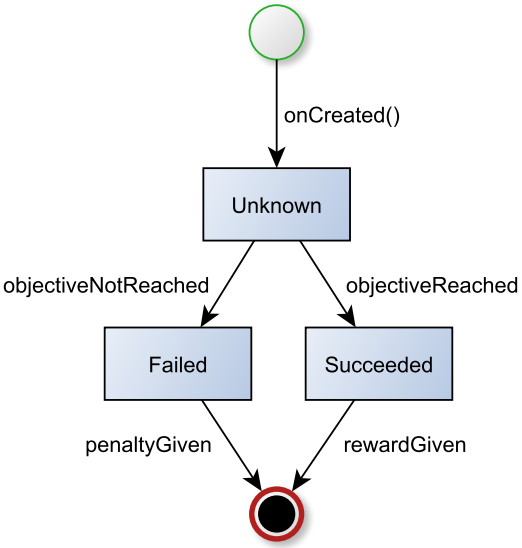


Fig. 1. The gameplay component state machine

The GC starts on unknown state. At this state, GC applies challenge scripts. According to the result of the evaluation function the GC switches in succeeded state or failed state. In succeeded state, the GC applies reward scripts whereas in failed state the GC applies penalty scripts. Finally, the GC is destroyed.

VI. Experiment

The objective of this pilot experiment is to evaluate the impact of using gameplay components on the game prototyping process and to evaluate how these concepts are understood by programmers.

A. Protocol

The experiment follows an independent-measures design with two independent groups. A group test using a game development framework with the gameplay components and a control group using the same game development framework without the gameplay components. The base game development framework is an open source framework named AGDE accessible from http://gforge-lirmm.lirmm.fr/gf/project/agde/frs

The experiment proceeds as follows:

1) Each group gets a quick introduction about the AGDE framework. This course aims to provide an introduction to AGDE engine for both groups.
2) The same game design document is given to candidates. Both groups have access to the same set of resources (graphics, scripts, music). The test group has to develop the game with gameplay components.
3) After one hour, the number of implemented features is taken for all games. The game is tested by a player that validate the presence of each features or not.

Game prototypes are evaluated by providing the following scores for each feature specified in the document:

- 0, if the feature is missing.
- 1, if the feature is achieved.

B. The Game: My Duck Hunt

My Duck Hunt is a single player shooting game. The player moves a reticle on the screen and must eliminate targets that are presented. It requires the player to be able to shoot moving ducks. The player is evaluated on its ability to destroy all targets.

The game allows the player to choose between three difficulty levels at the start of the game. It is played in ten successive waves. Each wave requires to eliminate eight ducks. At the end of the last wave, the player gets his final score.

C. Participants

Candidates are developers with basic knowledge in Java and oriented object programming. They do not have a lot of experience on game development.

D. Hypothesis

In order to check the differences between the groups, the following hypotheses are stated:

- H0. there is no difference between the test group and the control group in terms of number of features that are implemented.

Fig. 2. A screenshot of My Duck hunt

Group	+1 hour	+2 hour
Test ($n = 5$)	3.25 ($\pm$1.095)	4.2 ($\pm$1.48)
Control ($n = 4$)	0.23 ($\pm$2.06)	3.25 ($\pm$1.29)

TABLE I
DATA OF THE PILOT EXPERIMENT

E. Results and discussions

The data of this pilot experiment are presented by table I.

In average, the test group has performed better that the control group. However, considering the small number of participants in this pilot experiment, the differences between the groups cannot be considered statistically significant. This means that this pilot experiment cannot prove that using gameplay components accelerates game prototyping when compared to using usual scripting methods.

Nonetheless, qualitative interviews conducted after the experiment have highlighted some promising points.

We were expecting difficulties to make programmers reason about their game in terms of tree structure and gameplay components. The interviews revealed that the logics behind gameplay components has been accepted and adopted by programmer very quickly. In fact, tree structures and logical reasoning is part of their background and culture.

However, when the gameplay situation becomes too complex and involves many conditions and operators then programmers face difficulties to model these situations as gameplay component expression. For instance, modelling situations when a defined number of ducks have to be killed was quite easy. However, modelling situations when the number of ducks is generated dynamically was found difficult. For instance, AND and OR operators were found very intuitive to express game situations. Other operators have been considered as less intuitive and need more practice to understand their semantics and usage.

VII. CONCLUSION

In this paper we have introduced a formal language of gameplay components (GC) that allows to easily design, create and adjust gameplay loops.

GC language composition propriety allows us to :

- easily reuse game components in order to quickly prototype a video game
- develop programs which are able to adapt the game to some constraints

This document has presented the result of a pilot experiment. This experiments aims to evaluate the impact of using gameplay components on the prototyping stage of game development process, and evaluate how these concepts are understood by programmers. This experiments cannot prove that using gameplay components accelerates game prototyping when compared to using usual scripting methods. However, qualitative interviews revealed that programmer adopted this concepts very quickly.

As future work, we aim to perform the experiments that invalids or not our hypotheses: there is no difference between the test group and the control group in terms of number of features that are implemented at the end of development. This experiment will be conducted with more candidate that will be trained before the experiment.

We plan to develop tools which allow the visualization of the gameplay component tree. The purpose of this tool is to help game developers by displaying the state of the game tree. Moreover we want to create a library of game components to allow game designers to share GC as resources with a community of game developers.

Other perspectives are to compare our approach to less formal methods such as storyboarding. Furthermore we intend to answer these questions : *How does our approach scale well to more complex games ? Does our approach work better for particular types of games ?*

REFERENCES

[1] M. Krawczyk and J. Novak, *Game Development Essentials: Game Story & Character Development*. Thomson Delmar Learning, 2006, vol. 14, no. 1.

[2] M. Albinet, *Concevoir un jeu vidéo: Les méthodes et les outils des professionnels expliqués à tous !*, ser. Entreprendre: Développement professionnel. Fyp éditions, 2011. [Online]. Available: http://books.google.fr/books?id=iwOFZwEACAAJ

[3] L. E. Nacke, A. Drachen, K. Kuikkaniemi, J. Niesenhaus, H. J. Korhonen, W. M. van den Hoogen, K. Poels, W. A. I. IJsselsteijn, and Y. A. W. de Kort, "Playability and player experience research [panel abstracts]," in *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference*, A. Barry, K. Helen, and K. Tanya, Eds. London: Brunel University, September 2009. [Online]. Available: http://www.digra.org/dl/display_html?chid=09287.44170.pdf

[4] M. Frank and C. Berner, *Qu'est-Ce que le Neo-Structuralisme?*, ser. Passages (Paris. 1986). Cerf, 1989. [Online]. Available: http://books.google.fr/books?id=rWt4QgAACAAJ

[5] G. Frasca, "Simulation versus Narrative: Introduction to Ludology," in *The Video Game Theory Reader*. Routledge, 2003, ch. 10.

[6] D. Djaouti, J. Alvarez, J.-P. Jessel, G. Methel, and P. Molinier, "A gameplay definition through videogame classification," *Int. J. Comput. Games Technol.*, vol. 2008, pp. 4:1–4:7, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1155/2008/470350

[7] E. Montero-Reyno and J. . Cars-Cubel, "A platform-independent model for videogame gameplay specification," September 2009. [Online]. Available: http://www.digra.org/dl/display_html?chid=09287.28003.pdf

[8] A. Rollings and D. Morris, *Game Architecture and Design: A New Edition*. New Riders Games, 2003.

[9] S. M. Grünvogel. (2005) Formal Models and Game Design - last accessed 29.04.2012. [Online]. Available: http://www.gamestudies.org/0501/gruenvogel/

[10] S. Bura. (2006, Mar.) A Game Grammar. [Online]. Available: http://www.stephanebura.com/diagrams/

[11] J. Dormans, *Engineering Emergence: Applied Theory for Game Design*, 2012. [Online]. Available: http://www.jorisdormans.nl/pdf/dormans_engineering_emergence.pdf

[12] M. Araujo and L. Roque, "Modeling games with petri nets," in *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference*, A. Barry, K. Helen, and K. Tanya, Eds. London: Brunel University, September 2009.

[13] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[14] R. Lopes and R. Bidarra, "Adaptivity challenges in games and simulations: A survey," *Computational Intelligence*, vol. 3, no. 2, pp. 85–99, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5765665

[15] J. Doran and I. Parberry, "A prototype quest generator based on a structural analysis of quests from four mmorpgs," in *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, ser. PCGames '11. New York, NY, USA: ACM, 2011, pp. 1:1–1:8. [Online]. Available: http://doi.acm.org/10.1145/2000919.2000920