

ajME – Making Game Engines Autonomic

Pedro Martins
Department of Computing
Imperial College London
pm1108@doc.ic.ac.uk

Julie A. McCann
Department of Computing
Imperial College London
jamm@doc.ic.ac.uk

ABSTRACT

Autonomic Computing is now showing its value as a solution to the increased complexities of maintaining computer systems and has been applied to many different fields. In this paper, we demonstrate how a gaming application can benefit from autonomic principles. Currently, minimal adaptivity has been used in games and is typically manifested as bespoke mechanisms that cannot be shared, extended, reused etc. In this paper we show the advantages of Autonomic Computing in terms of not only improved performance, but also show that decoupling adaptivity mechanisms from the managed game can be done efficiently whilst improving its software engineering.

To this end we implement and evaluate a proof of concept architecture using the popular Java game engine jMonkeyEngine and in doing so produce autonomic extensions for the jMonkeyEngine (namely ajME). We show that this framework enables easy adoption of autonomic computing in games created using this games engine but also how this relates to other engines. We conclude that autonomic computing in gaming is viable (i.e. performance is improved while leaving the game quality minimally changed), has advantages over other approaches from a software engineering point of view and all with a minimal overhead. We then discuss the difficulties that are still present in the implementation of autonomic gaming systems, and suggest some further work that could be done in order to improve this area.

Categories and Subject Descriptors

I.2.1 [Application and Expert Systems]: Games; D.2.10 [Design]; D2.7 [Distribution, Maintenance, and Enhancement]

General Terms

Design, Performance, Reliability

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Fun and Games 2010, September 15-17, 2010, Leuven, Belgium
Copyright 2010 ACM 978-1-60558-907-7/10/09 ...\$10.00.

Keywords

Autonomic Computing, Self-adaptive, Self-healing Systems, Software Engineering, Game Engine

1. INTRODUCTION

In 2004, the U.S. game industry as a whole was worth USD \$10.3 billion and this has steadily grown over the past few years. At the same time, customers are demanding more from games not only in terms of art and narrative but also systems performance. As a result the profit margins expected from developing a game are becoming very tight and larger numbers of staff are required to address the ever increasing graphical and programming complexities, etc. As a result, we are seeing the industry advocating the use of stronger software engineering principles with more generic (reusable) solutions being favoured over bespoke.

Gaming technology is one of the more complex areas of technology because it involves so many different and complex subjects, such as computer graphics, sound and AI which require the integration of a significant number of complex frameworks. Therefore, the automation of some of the development and maintenance processes as well as the ability to squeeze the best performance out of the technology is paramount to the success of the future of gaming systems.

To address the latter, we are now beginning to see the use of adaptivity throughout all areas of gaming technology e.g. AI [20, 5], audio [22] and graphics [13] (more details on existing adaptive gaming solutions can be found in section 8). However, this technology is usually only applied to a specific project, in a very ad-hoc and tailored fashion that is not easily reusable in future projects. Moreover, most of these systems are architecturally monolithic, so even if the code was to be made available, it would be extremely hard to adopt it for another project.

To this end, we aim to specify an approach that implements the typical principles of Autonomic Computing, not only to allow a gaming engine to be adaptable to its environment and users, but also to provide a solution that can be generic enough to be reusable in many projects by being built on accepted software engineering techniques. Furthermore, this generic solution must not impede the performance of the game in any way. As far as we are aware such an Autonomic Computing approach to adaptive gaming is novel and has not been attempted before.

Consequently, by using an independent framework for autonomic computing one gains extensive configurability and extendibility over the autonomic algorithms. Due to these characteristics the autonomic framework becomes highly

reusable and can be adapted to an existing game as long as the game is modified to expose the sensors and effectors that the framework will need. The overhead of having an additional system monitoring the game and of communication should be measured to ensure that the system keeps its performance. Therefore we wish to answer the following question

Can configuration features that are embedded in game systems be decoupled and governed by an autonomic manager while maintaining user satisfaction and performance?

We begin this paper by introducing autonomic computing, and how it can help or hinder the gaming experience. In section 4 we introduce the various elements that will compose our proof-of-concept autonomic game engine. Sections 5 and 6 look at the two phased implementation of the game with section 6 showing how well they perform. Section 8 discusses related work and we conclude in terms of evaluation, experiences and proposed future work in section 9.

2. AUTONOMIC COMPUTING

Autonomic Computing was a concept suggested by IBM in 2001 in their Autonomic Computing manifesto [16]. In it, IBM suggests that in order to tackle the increasing complexity and need for maintenance of existing software systems, one should think about automating those tasks. The approach they suggest mirrors the autonomic nervous system, that deals with subconscious human body operations, to ensure that it keeps working optimally. The manifesto highlights the following desired properties of an autonomic system, that guarantee that a system has optimal uptime and performance:

- **Self-configuration:** An autonomic system should be able to configure itself in order to optimize its behaviour, in accordance to some pre-established policy.
- **Self-optimization:** Such a system should be able to optimize itself and its use of resources in order to improve its performance or quality of service(QoS).
- **Self-healing:** An autonomic system should detect problems, diagnose them and attempt to solve them.
- **Self-protection:** The system should be able to protect itself from attacks from either users or other systems that might affect its behaviour.

Along with these properties, it is usually accepted that an autonomic system should also be able to apply these changes in a dynamic manner, continuously reoptimizing as the system runs [17].

2.1 The MAPE-K loop

The reference model suggested by IBM for autonomic computing is called the MAPE-K loop [19], and is depicted in Figure 1. It describes their proposal for achieving autonomic management of an independent hardware or software element, called a *managed resource*¹. To be used as building

¹The emphasized words in the following paragraphs correspond to keywords in the original document.

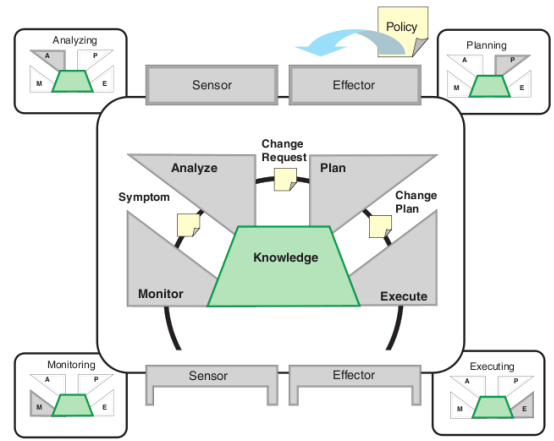


Figure 1: The MAPE-K loop [19]

blocks within the architecture, they need to provide *touch-points* which allow monitoring via *sensors* and acting via *effectors*.

An autonomic manager has the role then to implement the processing of the data from the sensors in order to derive an appropriate plan for acting upon the actuators. The stages that the autonomic manager should execute to go from the sensed data to instructions for actuators is as follows:

- **Monitor** the sensors and produce the behaviour metrics that might help the planning stages, by correlating the statistics with the *symptoms* that might be expected.
- **Analyze** the report produced by the previous stage and determine if the system can respect an established policy. This function might thus model and predict future behaviour, to better understand the system and help in producing a sound plan.
- **Plan** the appropriate actions to be taken on the managed resource. Generates a desired set of changes for the managed resource.
- **Execute** the previous actions, scheduling them and executing them through the effector interface in the touch-point.

Throughout the process the *knowledge* of the system needs to be stored, eventually in a shared *knowledge source* that can be used by many autonomic managers.

3. THE GAMING EXPERIENCE

In this section we discuss the ways autonomicity can impact the game experience from both technical performance and gaming atmosphere points of view. The gaming experience was analyzed by Roger Caillois in his book “Man, Play and Games” [9] where he argues that there are four basic types of gaming experiences. Mihály Csíkszentmihályi, in “Flow: The Psychology of Optimal Experience” also analyses aspects that are common to all optimal experiences, where the user feels in control and is efficient. This theory has been used in other analyses of the gaming experience [33, 15].

Combining both authors' concepts we extract the parameters which autonomic gaming systems can optimise:

- **Challenge** - is described by Caillois both in terms of Competition and Chance. Csikszentmihályi also points it out as significant in the optimal experience; a challenge that requires skill. Autonomicity can be used to improve the technical performance of the gaming engine by tailoring its operation to the system parameters (hardware, operating system etc), as well as current system conditions (e.g. available processing power). Furthermore, some players might be more proficient and require a more significant challenge while others need to be introduced to the genre first. In order to maximize the player enjoyment the system can try to balance difficulty and player performance to dynamically adjust the challenge accordingly.
- **Believability** - In order for the *Make-believe* experience to work, the world created has to be believable and not feel artificial when compared to the real world. In Csikszentmihályi's theory, this also contributes to the sense of clear goals and facilitates control by providing a friendly environment. Therefore it is extremely important that the flow of the game/graphics etc is not impaired by the systems performance and when resources are lessened the system can gracefully degrade.
- **Coherence** - Similarly to believability, all the different elements of the game need to make up a coherent experience. This can be optimized by ensuring that independently designed elements (e.g. audio and visual) match.
- **Performance** - In order to optimize the previous parameters, the game will have to not be hampered by artificial technical constraints. Self-management and adaptability bring with it essentially extra code to both probe and change system parameters and provide the adaptivity logic (control). When the system is decoupled from the autonomic logic, indirection is required to allow the probes to communicate with the Autonomic Management software and for this to communicate with the effectors. These extra costs should not impair the system's performance in such a way that it detracts from coherence and believability.

4. TOWARDS AN AUTONOMIC GAMING SYSTEM

In this section we will describe the reasoning behind our design and the choice of technologies for the proof of concept.

4.1 Games Architectures

Games are usually compiled from a number of different engines that manage particular aspects of the gaming experience. Thus, there is usually a graphics engine that deals with the rendering of the models and special effects on the screen; a physics engine that does physics computations for movements, collisions etc.; and a sound engine that plays audio. These are all connected together with a central controller, which will constitute the entry point for the game. The controller is typically structured as a sequence of loading the models, setting up game state, and entering a main

loop [1, 12]. The loop typically consists of a series of phases that will ultimately determine what is rendered on the screen for each frame. Typically we have:

- **Update** - Check for user input, call the physics engine for an update on the positions of the objects and perform game logic computations (e.g. if the player is near a door and has the key, open it). This phase will determine which objects are present in the game as well as their new states.
- **Cull** - From the objects that are logically present in the game at this given frame, extract the ones that are indeed going to be visible on the screen and need to be rendered. This phase saves computation effort on rendering by performing visibility tests.
- **Draw** - Render the objects that are going to be visible by calling the graphics engine. This stage could also handle calls to the audio engine in order to play sound effects and music.

Our proposed architecture is to add another engine - an "autonomic engine" - that is fully decoupled and optional, which will inject itself into the update loop for sensing and actuating with minimal changes to the existing code.

4.2 Adding Autonomicity

Believability, coherence and performance are the high-level variables that can be calculated from observable lower-level features like frame rate and variation of score. Using the terminology used in the autonomic implementation *Kinesthetics eXtreme* [21] we can say that the latter are the sensors in the system, and the gauges are responsible for estimating the former high-level variables. These will mirror the touchpoints, the sensors in the autonomic MAPE-K loop [19].

Reasonable defaults compiled from playtesting, combined with user preferences could be used by the autonomic framework to decide which component of the gaming experience to value the most (e.g. if the user prefers performance over believability).

From the previous description of the technologies, experimentation has to be done in order to provide generic effectors that can be used in more than one case. To optimize the variables, the effectors should enable better performance, e.g. optimizing the graphics and physics computations for the available processing power while not compromising the enjoyment of the player, by monitoring performance and response to the game.

Following the traditional MAPE-K loop the autonomic manager will have to monitor the sensors, analyze them to derive symptoms of a bad experience and predict the player's future experience, plan which effectors should be used, and then execute these plans.

Adaptive tessellation/decimation of surfaces, as described in [8, 25] is an approach to minimize computation for the rendering phase in an almost unperceivable manner by adaptively adjusting the detail of the 3D models according to the distance the user will be from the object. In this way, the model will remain believable and the computation power required is minimized. The different levels of detail for the models can be automatically created, by using decimation algorithms.

A particularly good algorithm for adaptive simplification of surfaces, that is implemented in the gaming engine jMonkeyEngine, is the Garland-Heckbert algorithm [14].

This algorithm is well balanced in efficiency, quality and generality [14] and is thus a good choice for the decimation in our proof-of-concept managed game.

5. ARCHITECTURE AND IMPLEMENTATION

In this section we discuss the implementation of the components involved with the proof-of-concept implementation and the tools used therein.

5.1 Autonomic Computing Architecture - Apache Felix and iPOJO

There are many options for building an Autonomic System architecture, with different protocols and communication schemes. The metrics used in selecting an architecture were autonomy, abstraction and ease of implementation. Apache Felix with the iPOJO project [4] makes the implementation of managed components as easy as annotating a POJO (Plain Old Java Object) with special annotations for exporting methods as JMX [32] [7]. This reduces the implementation effort, especially important when using JNI[24] for exporting touchpoints. This architecture also has a very simple communication protocol by using JMX, which is implemented in the JRE and provides a service registry and an easy API for calling registered methods.

Further autonomy of the managed system and managed component is also ensured by the OSGi framework, by making them run as separate bundles within the Felix runtime [3]. Thus, they can run completely independently and dependencies can be specified. The connections via the JMX handler are then made on demand.

As far as abstraction goes, due to the fact that the managed component can be implemented simply as an annotated POJO means that the communication and JMX code is abstracted away and injected by iPOJO, such that no special code needs to be provided for that purpose.

5.2 Gaming Engine - jMonkeyEngine

jMonkeyEngine (jME) is a 3D gaming engine for Java [1]. It was chosen to ease the development of the proof of concept as it uses the same language as the rest of the toolchain (Java) and also can run easily on the JVM; this facilitated an exploratory approach to building our proof of concept. However, it should be noted that using the JNI (Java Native Interface) [24], any engine can eventually have its calls exported into the Felix framework, and thus be remotely managed.

jMonkeyEngine already provides the framework for building meshes with multiple levels of detail as well as a model implementation of a continuous level of detail (CLOD) mesh that adjusts its level of detail using the Garland-Heckbert algorithm [1], according to the area that it occupies on the screen. In order to export this behaviour to the autonomic manager, a new class (`ExportedCLODMesh`) was implemented, that exposed this level of detail as a modifiable parameter to the Felix runtime.

5.3 Controllers

There are two different types of controllers in Control The-

ory. The simplest controller is the open-loop controller [30]. This type of controller calculates the appropriate inputs to feed to the system in order to make it produce a desired output by using only the state of the system and a model of its behaviour. Thus, for this sort of controller to perform appropriately it needs prior knowledge of the system. This is very difficult in systems that depend on complex and dynamic environments.

Another, more interesting approach to control is the feedback controller [11]. In it, control is represented as a feedback loop, where the current output of the managed system compared to a reference value serves as an input to a controller that will calculate the appropriate input to send into the system.

In practical control systems, one of the most widely used controllers is the PID (proportional-integral-derivative) controller [23]. This is a generic solution, widely adopted for controlling a single variable in order to make it reach a desired setpoint. This controller is composed of a generic feedback loop, as depicted in figure 2.

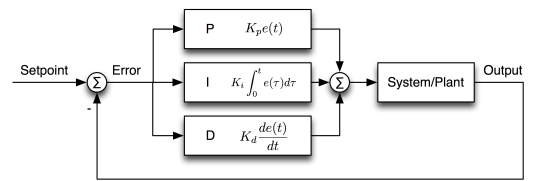


Figure 2: PID Controller block diagram.

The controller bases its operation on producing an error signal by subtracting from the output the desired setpoint. It then applies three distinct operators to this error signal that each have a very specific role in the control loop: A proportional component ($P_{out} = K_p e(t)$) is responsible for adjusting how much the system's inputs are changed in response to a change in output, whereas an integral ($I_{out} = K_i \int_0^t e(\tau) d\tau$) and derivative ($D_{out} = K_d \frac{d}{dt} e(t)$) components affect the system depending on the integral and derivative of the error signal respectively. The two last components can thus be used to better shape the curve for fast response and low overshoot.

The PID controller, having all these components weighted, then produces the final control signal by adding all their contributions together:

$$P_{out} + I_{out} + D_{out} = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \quad (1)$$

There are many methods for tuning a PID controller in order to ensure that the system behaves optimally under the desired circumstances. Manual methods usually start by adjusting the proportional term until the response starts to oscillate, and then adjust the integral and derivative terms to make sure the system adapts quickly and does not overshoot too much or become unstable. We use this approach in the work presented here.

5.4 Writing the app-specific autonomic manager

Having chosen the technologies and architectures to build the autonomic game system we implemented it in two phases.

Firstly, the autonomic adaptation logic was decoupled from the main body of code essentially implementing an application specific autonomic manager. This was done in order to test the overhead of decoupling using iPOJO, before properly engineering the manager. This version will also serve as a reference model for the final manager, which takes this implementation, finds the possible variation points and provides interfaces for replacing them with custom-tailored versions, see figures 3 and then 4 respectively.

Using these tools, the first task was to write a Java application using the jME that loads and displays a graphic model with a large number of polygons, and make it able to have its polygon count reduced remotely. That application was then bundled into an iPOJO bundle that was run in Felix and the sensors and effectors (framerate and level of detail (LOD), respectively) for remote controlling were exported. The final task was to implement an autonomic manager that accesses these exported calls and manipulates the level of detail to ensure that the framerate was kept within an interval of desired levels.

For this phase, we set up a jME application by using the helper class `SimpleGame` provided by the jME. We used an existing jME 3DS max importer [1] to import the .3ds version of one of the Stanford laser scanning 3D graphic models [31] which are widely-used for evaluation in the graphics research community, chosen because of their high polygon count. This will be featured in the evaluation section as a bunny model.

Then, using the CLOD class with the level of detail parameter exported previously we wrapped all the models in an `ExportedCLODMesh` and added them as a list of `ExportedCLODMesh` to the state of the main application, so that it can be exported to the Felix runtime.

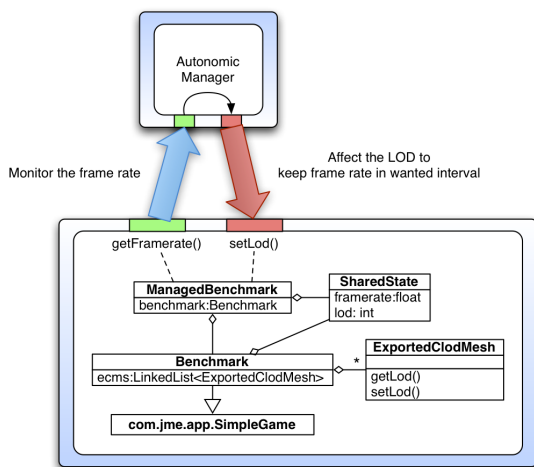


Figure 3: Simplified system architecture.

We did that by inspecting the model graph returned by the model loader to find raw mesh nodes, and then wrapping those into an `ExportedCLODMesh` node in order to have access to the level of detail of the mesh.

This will ensure that the level of detail for all the meshes can be exported. It is worth noticing how this technique can easily apply to any jME game, by traversing the tree,

finding all the models and wrapping them in a CLOD node.

For further detail on architecture and implementation, please refer to [26].

5.5 iPOJO, Sensors and Actuators

Due to the architecture adopted, it is necessary for the game bundle to export the calls corresponding to its sensors and effectors. iPOJO provides a JMX handler that can be used for this purpose. By using this handler the bundle will run as an MBean and can be remotely managed by the standard JConsole or any Java MBean client.

We declared a bundle such that the class `ManagedBenchmark` will be instantiated right after the bundle is loaded. This bundle will then act as a JMX MBean that exports the methods `getFramerate` and `setLod` for remote invocation.

For this to work then, the first phase implementation benchmark would also need to access this state and update all its models once per frame (in the update method, `tpf` stands for time per frame and is provided by jME):

```
protected void simpleUpdate() {
    super.simpleUpdate();
    state.setFramerate(1.0f/tpf);

    for(ExportedCLODMesh ecm : ecms) {
        ecm.setLod(state.getLod());
    }
}
```

Listing 1: Update method.

With this setup, the benchmark was bundled using the ant buildfile and the auxiliary tool `bnd`, as suggested in the Felix documentation. After the target is run, the benchmark bundle is produced and can be loaded into Felix. Upon loading, the engine will start rendering the scene and Felix will export the methods specified in the metadata (`setLod` and `getFramerate`) and make them available for remote invocation.

5.6 Autonomic Manager

For the autonomic manager, a simple MBean client was implemented. The core loop consisted of querying the frame rate and incrementing or decrementing the level of detail, if this frame rate was respectively above or below the desired interval:

```
while(!exit) {
    sleep(5000);
    float framerate =
        (Float)mbsc.invoke(mbeanName, "getFramerate",
            new Object[0], new String[0]);
    if (framerate < MIN_FPS && lod != MIN_LOD) {
        lodArgs[0] = new Integer(--lod);
        mbsc.invoke(mbeanName, "setLod", lodArgs,
            lodTypes);
    }
    else if (framerate > MAX_FPS && lod != MAX_LOD)
    {
        lodArgs[0] = new Integer(++lod);
        mbsc.invoke(mbeanName, "setLod", lodArgs,
            lodTypes);
    }
}
```

Listing 2: The ad-hoc autonomic manager main loop.

This corresponds to a discrete-time SISO(Single Input Single Output) fixed rate controller with a rate of 1 LOD level per iteration. This worked very well and is evaluated in section 7.

6. AUTONOMIC EXTENSIONS FOR THE JME

Following the proof of concept, and in line with the main goal of the project, the next step was to implement a generic, reusable autonomic manager, that would ease the adoption of the framework for game developers. Our autonomic manager follows the MAPE-K loop scheme [19].

6.1 Design

The first step to make the autonomic manager very extendable was to create type system constructs for all the entities referred to in the MAPE-K loop scheme. Thus, a Java interface was created for managed sensors and effectors, detailing the methods they should implement.

A managed sensor is defined as an entity that is readable via a JMX exported method, and is of a given type T (extending Comparable for convenience). The `ManagedSensor` class also defines a range that is the desired reference interval.

A managed effector is very similar in structure to the managed sensor, differing in the roles of the different components. The JMX exported method is now used to write a value, and because this value might need to be set to a relevant initial value, that facility is provided in the constructor. There is also a range provided for convenience, that specifies the acceptable possible range. Range is an utility class that represents numerical ranges.

Having defined a type system representation for sensors and effectors, the autonomic manager can now be defined. Both sensors and effectors have to be registered with the autonomic manager and will store the necessary JMX call names for reading/setting their values. The data structure used in the autonomic manager was a generic `LinkedList` for both cases.

With these interfaces defined, the monitor part of the MAPE-K loop was defined as such:

```

for(ManagedSensor ms : sensors) {
    // Call the appropriate method to read the
    // value from the sensor over the JMX exposed
    // call
    Comparable value =
        (Comparable)mbsc.invoke(mbeanName,
            ms.jmxMethodName, new Object[0],
            new String[0]);

    // if it is outside of the desired range it is
    // a low-level symptom
    if(!ms.desired.contains(value)) {
        kb.put(ms,
            ms.desired.calculateSymptom(value));
    } else {
        kb.remove(ms);
    }
}

```

Listing 3: Monitor.

The monitor part of the loop is responsible for reading all the values from the sensors and extracting symptoms that need to be addressed by the manager. In our case, a

symptom would be the frame rate being below the minimum threshold, or above the maximum threshold. The symptoms extracted from the raw data are then stored in a general purpose Knowledge Base, implemented as a Java map.

The next phase of the MAPE-K loop, Analyse, will then analyse the symptoms, conditions and target situation in order to check if the end goal can be reached. As previously mentioned, this phase is not necessary in our case because we are using policies. However, it is open for extension so that a more sophisticated system can be implemented.

Following that, the autonomic manager needs to plan what actions to execute. For the planning phase, a strategy class was defined. In this case, a *strategy* is a possible approach to effect the value of a given sensor/set of sensors, so that they respect the goals of the system. Strategies were defined to ensure that alternate policies can be declared in the system, and an intelligent planner can choose between them. A very low-level `Strategy` interface was defined, such that the strategy could be performed on a given managed object.

This strategy mechanism can also be used for declaring deterministic policies, that act as simple rules. This simplifies the planner immensely and still works quite well for simple cases. Again, in more elaborate cases, the planner could take the symptoms (or the analysis made by the analysis phase) and make some more complex planning decisions, so it was left open for extension. The execute phase of the loop will then just take the applicable strategies and execute them.

With all these components the final framework representing the MAPE-K loop looks like the diagram in figure 4. With this generic autonomic gaming framework it should be easy to add autonomic capabilities to an existing game. The framework was designed in order to be flexible and extensible. In addition to that, the usage of iPOJO and JMX means that the autonomic manager and managed system can be completely decoupled and independent.

7. EVALUATION

As noted, most of the existing adaptive game research has been applied to AI [6] [18] [10], music [22] and game difficulty [29] adaptation. There was also work done in using adaptive triangulation in meshes for real-time performance optimization [34] or in adjusting shading level of detail for performance [28]. However, in all these systems, the focus of research was in creating the adaptation technique. We could find no system that attempted to automate existing manual configuration parameters to achieve optimal performance that could eventually serve as a reference system.

7.1 Performance and Controller Evaluation

In this paper, our goals were to show both that autonomic computing was applicable to gaming, and that adjusting the level of detail of the geometry was a very efficient way to adjust the performance of a game. One of the controllers we first used corresponds to a constant increment/decrement controller, which can be thought of as a proportional controller with rate limited to 1 LOD unit per iteration. Thus, it should have a very smooth but slow adaptation.

The benchmark was run on a MacBook Pro 2.53GHz Intel Core 2 Duo with an NVIDIA GeForce 9400M graphics processor with 256MB of DDR3 SDRAM and it consisted of running the application, showing no model for 3 seconds and then switching the rendered geometry to the full bunny model, adding more polygons to be rendered; thus decreas-

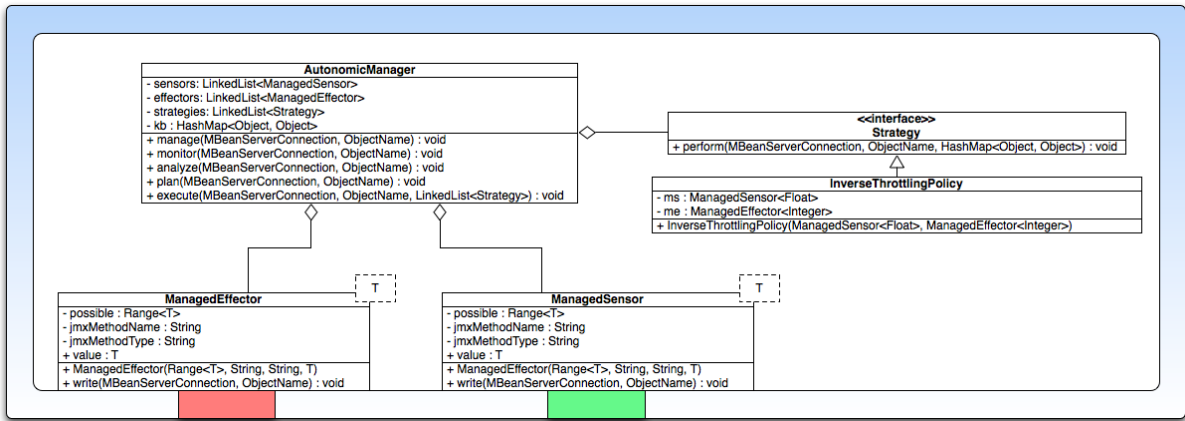


Figure 4: UML Diagram representing the type system representation of the reference architecture.

ing the framerate below the desired values and forcing the system to adapt the level of detail as can be seen in figures 5 and 6. (Please note that the clocks in the different experiments were not synchronised, therefore systems will start their adaptation at different times as can be seen in the level of detail graph in figure 6.)

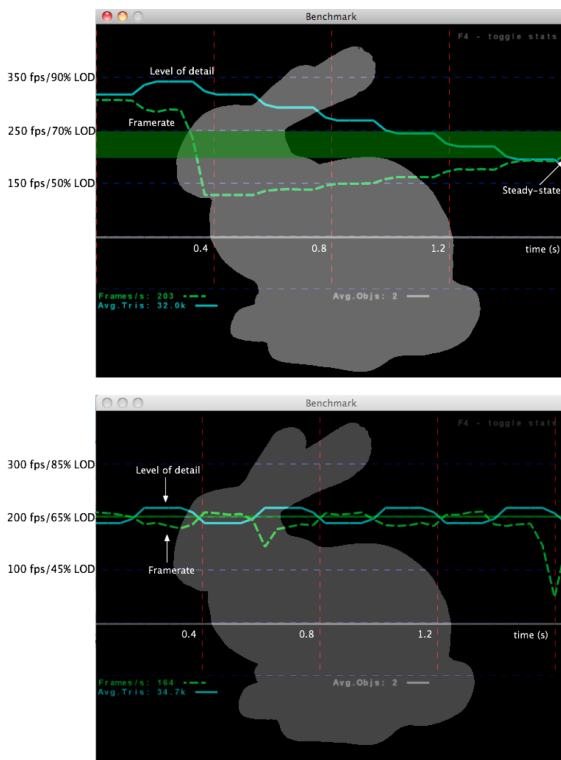


Figure 5: Proof of concept graphs, showing the possibility of oscillation

The results gathered in that benchmark can be seen in figure 5, where the benchmark application was allowed to run unmanaged until it reached a steady state, and after that the autonomic manager was activated and a transient change was produced. The initial environment did not dis-

play any geometry, and so ran at a very high framerate with maximum level of detail. The bunny model was then loaded, so the system's framerate dropped below acceptable levels and forced the system to adapt. The resulting behaviour can be observed in the graph. As the legend indicates, we show both level of detail and the framerate.

It can be seen in figure 5 that the autonomic manager reduced the level of detail progressively until the framerate had an appropriate value according to the interval specified in the policy used. It can also be seen that the very simple incremental controller behaves as expected, slowly adapting the model to make the system achieve the desired state.

This shows that this framework works for the simple case of a single variable, and is able to slowly reach the desired operating point. Further, by experimenting with the sample intervals and acceptable frame rate intervals the system can even be made to oscillate as shown in figure 5. These oscillations can be removed by widening the target frame rate interval.

Taking advantage of the flexibility of the implementation, we then replaced the simple controller with a PID controller. For tuning we began with the implementation of the proportional controller and testing and experimenting with the proportional constant until we achieved the required behaviour. The graphs for many levels of K_p can be found in figure 6 where the desired framerate range is shaded.

It can be seen that the proportional-only controller behaves exceedingly well on this example, adapting to the necessary level with very little overshoot and oscillation, and in the tuned version, taking only one iteration to achieve the desired value. After achieving a stable state, all the examples will maintain performance within the desired bounds with no level of detail changes. Thus, a proportional only controller would work very well for a similar practical scenario. Please note that we have not varied the machine's available resources, however this impacts the graphics in the same way as adding the bunny model geometry to the scene. It can also be seen that it is safer to use a controller with smaller values of K_p , as if K_p is too large, overshoot combined with maximum and minimum² value clamping makes the system oscillate endlessly between the two extremes (see

²The maximum and minimum values for the level of detail in this experiment were respectively 0 and 20

case for $K_p = 0.3$ in the graph in figure 6).

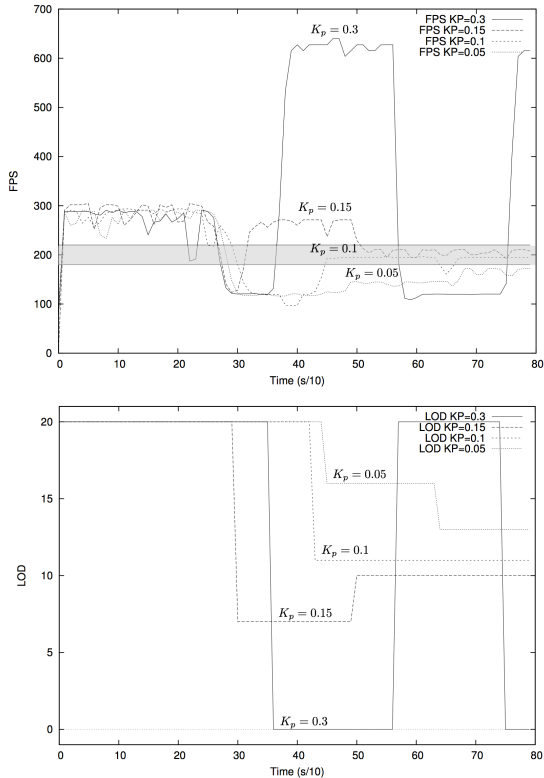


Figure 6: Frame rate and level of detail measurements for proportional-only controller test, with varying levels of K_p .

However, in a practical scenario, all these fast rates of change in the level of detail might be too drastic and so the maximum rate should be limited.

In this small example it can be seen that adapting only one variable with a simplified autonomic manager can be simple and effective. To compare the overhead, the application was tested running with and without a stub autonomic manager that just set the default state in every iteration (with the actual manager, the framerate would be at the setpoint). We measured 121 frames/second (fps) for the model being rendered at full level of detail with no autonomic management, and 116 fps for the full model being rendered with an autonomic manager that set the level of detail to full in every iteration.

It can be seen that the overhead added by the autonomic manager is very minimal in this example application, corresponding to 4.1% of the application running with no autonomic management. However, this behaviour could eventually be explained by the fact that the game itself is already running on the JVM. An interesting further research project would be to adapt a non-Java game by using the JNI and manage it with a Java autonomic manager, to measure the overhead in that scenario.

With this minimal overhead, the gaming experience exhibits near optimal characteristics, adapting at an adjustable rate to any framerate that is chosen as a setpoint, as long as it is achievable with level of detail manipulation. The transient used in this experiment was indeed quite large, from

0 polygons on screen to 69451 triangles being rendered. In this situation the adaptation can take as few as one iteration (as in the graph for $K_p = 0.1$ in figure 6), and as long as 8 iterations (as in figure 5). The possibility of state flapping is also minimised by combining the size of the discrete level of detail steps with a large setpoint interval, to ensure that the frame rate will never be oscillating around one of the extremes.

7.2 Autonomic System Evaluation

The implemented autonomic manager was built to respect the autonomic computing white paper from IBM [19]. Thus, all the MAPE-K loop stages were isolated and implemented with separate sections, with the communication between them being made by using a separate knowledge base.

A subject that the MAPE-K loop scheme leaves open to implementation is the planning mechanism. In order to be flexible and support further extension, a conceptual class hierarchy was built, that supports strategies to be defined. In order to adapt the previous examples, a simple Policy class was defined as a very limited Strategy, that allowed easy implementation of ECA (Event/Condition/Action) rules.

This autonomic manager should allow easy adaptation of any application that exported the touchpoints using any protocol that is readable on the Java side. If the application does not export these attributes in a readable way, it should be easy to use the JNI in order to access these attributes, while adding the appropriate synchronisation precautions.

Using the topics presented in [27], the system was evaluated thus:

7.2.1 Performance and Costs

In the specific case of the first benchmark, the level of detail was set at a level that would optimise the achieved frame rate. Thus, in terms of raw performance the autonomic system effectively improves the base system by maintaining an average of 203 fps where the required interval was 200-220 fps. The cost of implementing autonomicity in this case would be the overhead in terms of extra coding and communication as well as development time. However having provided a reusable framework that should ease the implementation (particularly when comparing with hard coding adaptivity) this development cost is insignificant.

7.2.2 Failure Avoidance

In the case of our proof of concept, failures could be defined as times when the frame rate goes beneath a certain minimum acceptable value. That would happen regardless of the system being managed, when it is operating at the minimum level of detail (so there is no further adaptation possible) and the scene is complex. However, in all other circumstances the system tries to avoid failure, bringing the system as close to the target performance as the adaptation capabilities allow it to.

7.2.3 Time to Adapt and Reaction Time

As seen in the previous section, when we were using the proportional-only PID controller, the time needed in order to adapt to changes in the parameters (for a tuned controller) was very minimal, in some cases as low as one iteration. Of course this requires tuning for the specific situation, as the model used may not be completely correct. However, we found that the system always behaved acceptably for

many levels of complexity in the scenes. Also, in this specific adaptation case, minimal time to adapt is not critical and might not be desired. As discussed, adaptation that is too fast can result in jumps in level of detail, and provide a jarring experience, that interferes with immersion and believability. Thus, this controller should always be tailored and tuned by the game designer, in order to provide the best experience for each specific case. For future work, if the other, more subjective parameters quantified we could optimise the experience automatically, and represent this sudden drop in level of detail as something undesirable that should be avoided.

7.2.4 Sensitivity

Sensitivity in our case was adjustable, and as can be seen in the graphs, too much sensitivity can make the system oscillate endlessly when it reaches steady-state. Too little sensitivity and the system might be caught in a suboptimal level of detail. It should be adjusted in order to provide the optimal performance under a wide array of cases; a further feedback loop could facilitate this.

7.2.5 Stabilisation

In this case the environment consisted of the geometry that was displayed at a given time given the processing power that was available to it. The geometry changed as the user moved through the world and the processing power can change due to external factors like background applications. As seen in the graphs, when appropriately tuned, the system with a PID controller can reach the neighbourhood of the setpoint very quickly. However, because of the application, the amount of fine tuning around the setpoint needs to be minimised and so it was found that stabilisation was quicker and smoother when the target interval was larger. The exact value of the performance metrics achieved is not relevant as long as it is enough to provide a good experience to the user. Thus, using larger intervals reduces the probability of state flapping and provides an optimal experience to the user.

8. RELATED WORK

We are seeing a move towards the use of adaptive techniques to tailoring the game to the player or system's performance; as adapting the game to your audience and their environment is a very desirable characteristic. Many games provide some way of autodetecting appropriate settings for the graphics engine at startup or on the configuration screen.

In addition to this, some games have used machine learning techniques to improve artificial intelligence(AI) systems [20], most notably Lionhead Studios' Black and White. In this game both the creature and the villagers continuously evolve their behaviour according to the player's actions. In fact the sheer intelligence of the creatures and villagers is one of the main points that makes the game fun. Another game in which machine learning techniques were used to improve the AI is Defcon, by Introversion Software, which used advanced machine learning and planning to achieve an optimal bot AI. [5].

Another notable usage of adaptive techniques to enhance the gaming experience on the audio/video coherence level, appeared in the recent game Spore from Maxis, which featured procedurally generated content extensively [22]. In this game, the developers used a modified version of the

open source sound design tool *pure data* to achieve sound synthesis and algorithmic composition. With the help of Brian Eno, a composer notable for his work on generative music, intelligent systems were trained to recognize what sort of sounds were appropriate.

In the domain of physics, some work has been done by AGEIA, in order to provide a physics engine which adapts to the characteristics of the hardware and available processing power by optimizing the computations being performed to varying degrees. [2] To reiterate, as far as we are aware this is the first attempt at applying traditional Autonomic techniques to a gaming engine.

9. CONCLUSION

This project aimed to explore the applicability and advantages of applying Autonomic Computing to gaming systems, with the aim of providing a reusable basis for the implementation of adaptivity. In direct response to our objectives, we found that applying autonomic computing to gaming systems is beneficial and compares positively to other approaches.

Regarding the implementation of autonomic solutions, the choice of iPOJO and Apache Felix significantly reduced the development effort and allowed for a better, more reusable architecture. Compared to an embedded autonomic solution this approach is better in code quality, reusability and development effort, while adding very little overhead.

We also found that while adapting variables that affected performance in gaming, the controller used is not as relevant as in typical control theory applications. This is due to the fact that a very fast adaptation might result in a big skip in level of detail, which will yield a jarring experience for the user. Thus, the adaptation should always be unobtrusive and gradual. In that case we found that a gradual, fixed rate controller worked optimally, because it gradually adapted the level of detail, and while it could oscillate (by having a small target interval or a small time step), it never became unstable. If the adaptation needs to be quick, a proportional-only controller does an excellent job, although it can, in certain extreme cases of proportional constant, overshoot and become unstable.

The final product of this project was a complete set of autonomic extensions for the jME, that provide a flexible framework for easily integrating autonomic capabilities into any jME game. With some more work in exporting the necessary calls into the Felix MOSGi platform, by using the JNI, the framework can even be used in a non Java game. This would be an interesting further research project, as the overhead created in that situation is bound to be higher.

Other possible further work could be performed in developing strategies and planning algorithms that work well when there are complex interactions between managed variables. This should be applied to a real world scenario of different configuration parameters that apply to a real game. Some additional interesting expansion would be to provide an even more complete set of extensions that applied the Garland-Heckbert CLOD mesh to an arbitrary scene graph. That should be fairly easy to do by just traversing the scene graph and wrapping all mesh objects in `ExportedClodMesh` objects.

Having implemented a more complex planner, in order to provide a more complete and correct adaptation, some work can also be done to quantify the more ambiguous parameters

like believability and coherence. That would allow assembling systems that adapt to provide as good an experience as possible, weighing all the different qualitative parameters according to a user profile.

10. REFERENCES

- [1] jMonkey Engine. <http://www.jmonkeyengine.com> (July, 2009).
- [2] AGEIA. AGEIA Technologies Introduces the AGEIA Adaptive Physics EXtensions (APEX) Development Platform. <http://www.marketwire.com/press-release/Ageia-793470.html>, March 2009.
- [3] Apache Project Team. Apache Felix. <http://felix.apache.org> (July, 2009).
- [4] Apache Project Team. iPOJO. <http://felix.apache.org/site/apache-felix-ipojo.html> (July, 2009).
- [5] R. Baumgarten. Combining Artificial Intelligence Methods: Automating the Playing of DEFCON. Imperial College London, 2007.
- [6] R. Baumgarten, S. Colton, and M. Morris. Combining AI Methods for Learning Bots in a Real-Time Strategy Game. *International Journal*, 2009.
- [7] J. Bourcier, C. Escoffier, and P. Lalanda. Implementing home-control applications on service platform. In *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, pages 925–929, Jan. 2007.
- [8] M. Bunnell. Adaptive tessellation of subdivision surfaces with displacement mapping. *GPU Gems*, 2:109–122, 2005.
- [9] R. Caillois. *Man, play and games*. University of Illinois Press, 2001.
- [10] J. Chen. Flow in games (and everything else). 2007.
- [11] J. Doyle, B. Francis, and A. Tannenbaum. *Feedback control theory*. Macmillan New York, 1992.
- [12] D. Eberly. *3D game engine design: a practical approach to real-time computer graphics*. Morgan Kaufmann Publishers, 2007.
- [13] J. Euh, J. Chittamuru, and W. Bursleson. A low-power content-adaptive texture mapping architecture for real-time 3d graphics. *Lecture notes in computer science*, pages 99–109, 2003.
- [14] M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1997.
- [15] A. Hejdenberg. The Psychology Behind Games. *Gamasutra.com*. http://www.gamasutra.com/features/20050426/hejdenberg_pfv.htm, 2005.
- [16] P. Horn. Autonomic Computing: IBM’s Perspective on the State of Information Technology. Manifesto, IBM Research, Oct. 2001.
- [17] M. Huebscher and J. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.
- [18] R. Hunnicke. The case for dynamic difficulty adjustment in games. In *ACE ’05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 429–433, New York, NY, USA, 2005. ACM.
- [19] IBM. An architectural blueprint for autonomic computing, white paper, 2004.
- [20] D. Johnson and J. Wiles. Computer games with intelligence. *Australian Journal of Intelligent Information Processing Systems*, 7:61–68, 2001.
- [21] G. Kaiser, J. Parekh, P. Gross, and G. Valetto. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Autonomic Computing Workshop, 2003*, pages 22–30, 2003.
- [22] D. Kosak. The Beat Goes on: Dynamic Music in Spore. gamespy.com GDC 08 Articles, February 2008.
- [23] W. Levine. *The control handbook*. CRC press, 1996.
- [24] S. Liang. *Java Native Interface: Programmer’s Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.
- [25] D. Luebke. A Developer’s Survey of Polygonal Simplification Algorithms. *IEEECGA: IEEE Computer Graphics and Applications*, 21, 2001.
- [26] P. Martins. ajME - Autonomic Extensions for the jMonkeyEngine. Master’s thesis, Imperial College London, September 2009.
- [27] J. McCann and M. Huebscher. Evaluation issues in autonomic computing. *Lecture notes in computer science*, pages 597–608, 2004.
- [28] M. Olano, B. Kuehne, and M. Simmons. Automatic shader level of detail. In *HWWS ’03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.
- [29] M. Ponsen, H. Muñoz-Avila, P. Spronck, and D. Aha. Automatically acquiring domain knowledge for adaptive game AI using evolutionary learning. In *Proceedings Of The National Conference On Artificial Intelligence*, volume 20, page 1535. Menlo Park, CA; Cambridge, MA; London; AAI Press; MIT Press; 1999, 2005.
- [30] S. Schaal, C. Atkeson, and C. MIT. Open loop stable control strategies for robot juggling. In *1993 IEEE International Conference on Robotics and Automation, 1993. Proceedings.*, pages 913–918, 1993.
- [31] Stanford University. The Stanford 3D Scanning Repository. <http://graphics.stanford.edu/data/3Dscanrep/> (July, 2009).
- [32] Sun. Java Management Extensions (JMX). <http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/> (July, 2009).
- [33] P. Sweetser and P. Wyeth. GameFlow: a model for evaluating player enjoyment in games. *Computers in Entertainment (CIE)*, 3(3):3–3, 2005.
- [34] M. White. Real-time optimally adapting meshes: terrain visualization in games. *Int. J. Comput. Games Technol.*, 2008:1–7, 2008.