G-factor Portability in Game Development Using Game Engines

Ahmed BinSubaih & Steve Maddock
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield, U.K. +44(0) 114 2221800
{a.binsubaih, s.maddock}@dcs.shef.ac.uk

ABSTRACT
Many games today are developed using game engines. This development approach supports various aspects of portability. For example games can be ported from one platform to another and assets can be imported into different engines. The portability aspect that still remains underdeveloped and requires further examination is the complexity involved in porting a 'game' between game engines. In this work the game elements that we aim to make portable are: game logic, object model, and game state which represent the game's brain. These are referred to as the game factor (or G-factor). We describe how a typical game development approach using the Torque game engine makes the G-factor dependent on the engine and we contrast this with a new approach which enables G-factor portability between engines.

Categories and Subject Descriptors
I.3.6 [**Computer Graphics**]: Three-Dimensional Graphics and Realism.

General Terms
Design, Architecture, Experimentation.

Keywords
Game engine, Portability, Game development.

1. INTRODUCTION
The shift in game development from developing games from scratch to using game engines was first introduced by Quake and marked the advent of the game-independent game engine development approach [20]. In this approach the game engine became "the collection of modules of simulation code that do not directly specify the game's behaviour (game logic) or game's environment (level data)" [27]. This makes the game engine reusable for (or portable to) different game projects. However this shift produces a game which is notoriously dependent on the game engine. For example why can't a player take his favourite game (say Unreal) and play it on Quake engine or vice versa?

Hardware and software abstractions have facilitated the ability to play a game on different hardware and on different operating systems (in some cases with some modifications). These abstractions have also facilitated the ability to use level data assets such as 3D models, sound, music, and texture across different game engines. This ability should also be extended to allow for the game itself to be portable. The goal of our work is to make the game engine's brain portable, where the brain holds the game state and the object model and uses the game logic to control the game. We collectively refer to these three things as the G-factor. We see the portability of the G-factor as the next logical step in the evolution of game development and following Lewis and Jacobson's terminology [20] we call it the game-engines independent game development approach (see Figure 1).

A benefit of making the G-factor portable would be to encourage more developers to make use of game engines, since a particular game engine's future capability (or potential discontinuation, as was the fate of Adobe Atmosphere which was used for Adolescent Therapy – Personal Investigator [10]) would not be a worry as a different game engine could easily be substituted. This problem has recently been referred to as "the RenderWare Problem" [9] after the acquisition of RenderWare engine by Electronic Arts (EA) and its removal from the market. We see the issue of rewriting the G-factor from scratch every time we migrate from one engine to another as similar to the undesired practice of developing games from scratch which was deemed unfeasible and resulted in the advent of game engines.

Section 2 describes the aspects of portability in relation to game engines and the techniques that have been tried to aid G-factor portability. Section 3 demonstrates the issues with the typical game development approach through the development of a sample game which is then contrasted to the development of the same game using the new approach which enables the G-factor to be portable. Section 4 describes the evaluations conducted and discusses the two development approaches. Finally section 5 presents the conclusions.

2. PORTABILITY AND G-FACTOR
Figure 2 illustrates the current aspects of portability addressed in game engines. First, with hardware and software portability the game can be played across different platforms and operating systems by employing hardware and software abstractions. Second, portability of assets means that 3D models, textures and sounds can be used across different game engines. Third, middleware portability allows for components to be used across game engines such as AI and physics.

The aspect of portability that requires further investigation is the G-factor portability. Examining what has been done to aid this portability we found initiatives and projects which can be grouped into four areas: artificial intelligence (AI) architectures, interfaces, standards and file formats, and frameworks or protocols.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CyberGames 2007, September 10–11, 2007, Manchester, UK.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

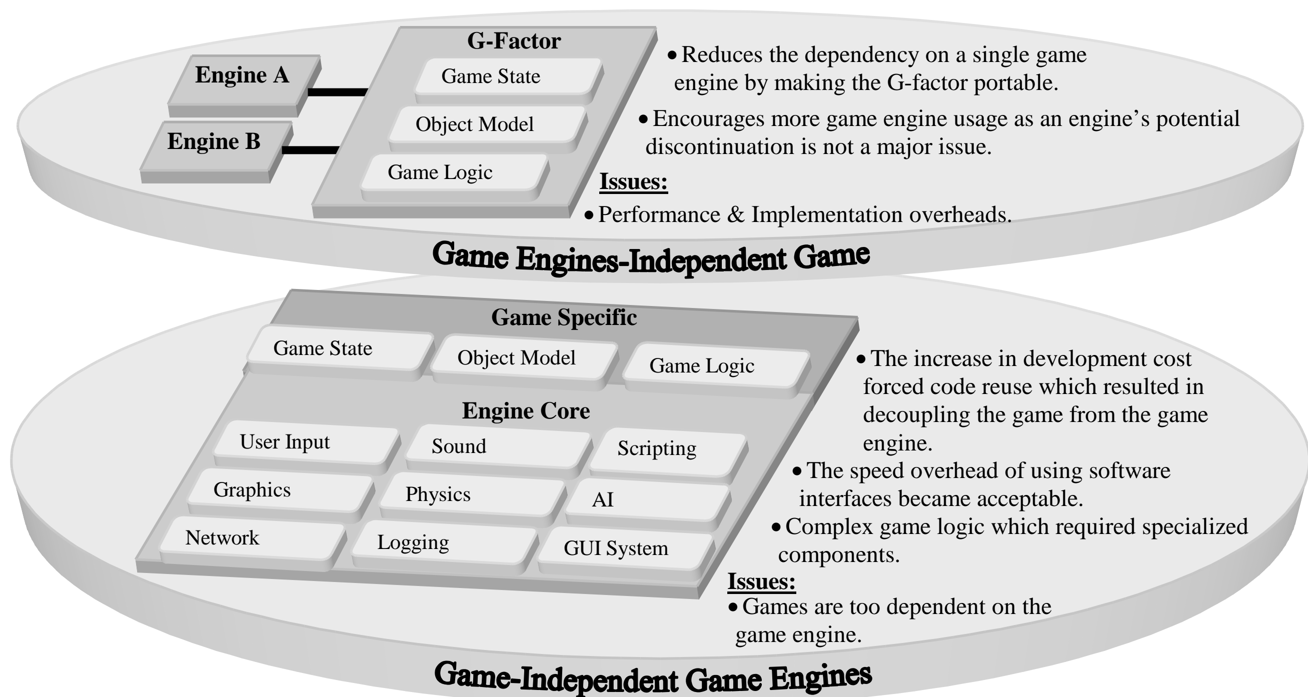


Figure 1: Game development evolution.

The AI architectures use custom made or off-the-shelf components such as the AI Middleware (e.g. SOAR [19], AI.Implant[1], etc). The need for using a component to handle the AI emerged because of the increase in AI complexity and the increase in the processing time allocated for it. This made reinventing the AI wheel every time a game is developed a redundant process. From a software engineering perspective this practice is encouraged as it promotes above all reusability. The practice of specifying the game using the AI middleware format is not what we eventually want since this merely moves it from one proprietary format (game engines) to another (AI middleware). Nevertheless it is a step in the right direction of moving the game away from the game engine's format. The architectures that promote portability more than others are those that allow complete removal of the game from the game engine such as TIELT [2]. Others that only partially remove the game are obviously less portable such as Mimesis [28] and MissionEngine [26]. The AI architectures promote the use of their own proprietary format which is similar to what game engines do. Furthermore suggesting a monolithic architecture as a complete entity is not what is needed. Instead initiatives must examine the causes of the G-factor portability problem and provide practical solutions that can be employed even if their architecture or middleware is not chosen.

The interfaces aim to provide access to external programs and in game engines we found two types of interfaces: specific and common. These provide access to the G-factor elements to overcome the difficulty raised by the lack of interoperability. A number of interfaces have been developed to provide access to specific game engines. For example the interfaces that have been used to access Unreal are Gamebots [1] and GOLOG Bots [14]. To access Quake one can use Quakebot [18]. FlexBot [17] is used to access Half-Life and Shadow Door [13] is used for Neverwinter Nights. These provide interfaces for specific game engines. Other projects are attempting to provide common interfaces to game engines such as the initiative by International Game Developers Association (IGDA) for world interfacing [21] and OASIS [3]. Interfaces may have more success in the serious games community rather than a fast evolving games industry.

The third area is the standards and file-based formats such as VRML/X3D[2]. These still lack the maturity needed for game development. For instance VRML lacks the rendering capability required. It also suffers from speed and security issues [15].

The fourth area is the frameworks or protocols that aid interoperability between different simulations like the High Level Architecture (HLA) [23] and Java Adaptive Dynamic Environment (JADE) [22]. Despite the fact that this category focuses more on the interoperability between simulations and less on how the game is linked to the simulation it is mentioned here to illustrate that portability exists at different levels. HLA for instance promotes it at the simulation and object level and JADE promotes interoperability at the functionality level. HLA identified the simulation functionality that are generally required across all systems and thus should not only be part of a single simulation system but available for others. To achieve this it moved the general simulation functionality from the simulation system to the HLA infrastructure and thus made them accessible to other simulation systems [24]. An example of the functionality provided is object management which is used to share

[1] http://www.biographictech.com (accessed 5/5/2007).

[2] http://www.web3d.org/x3d/ (accessed 5/5/2007).

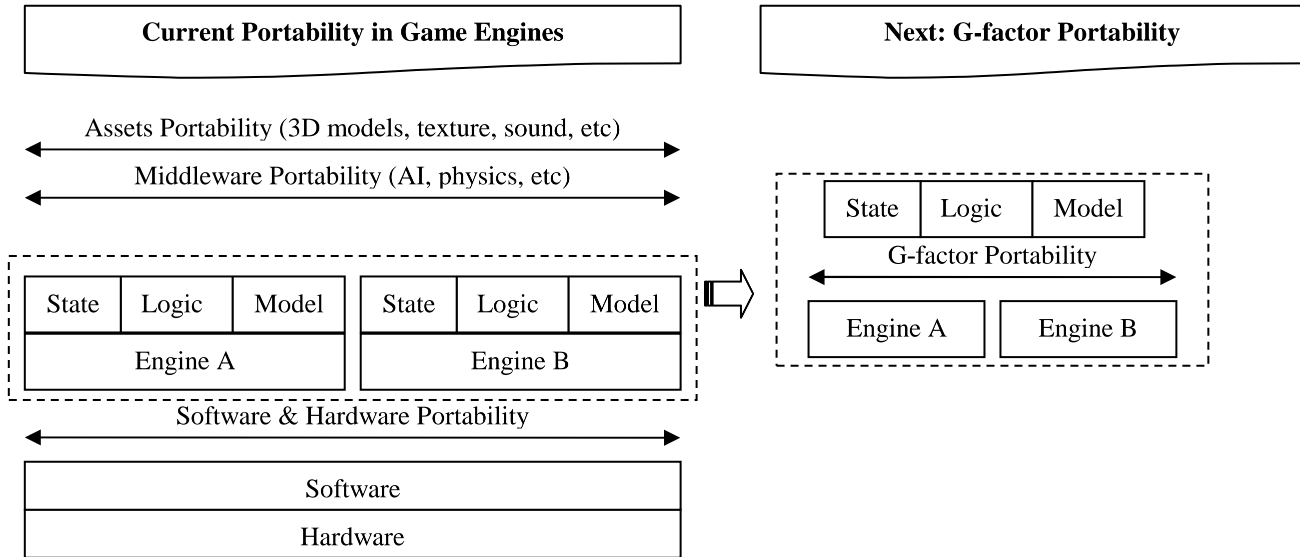


Figure 2: Portability in game engines.

object instances between different simulations. JADE was designed to address the monolithic nature of current Virtual Environment (VE) systems. Oliveira et al [24] argue that in current VE systems it is not possible to replace or increment the necessary functionality. JADE proposes to host Modules without the concern for their functionality which is the responsibility of the VE developer. A Module can encapsulate an entire system or a block of code and thus can be reused by others. These frameworks and protocols require the projects to comply with their infrastructure to be able to interoperate with other systems. The other challenge facing them is to create a generalizable infrastructure to support any kind of environment [16].

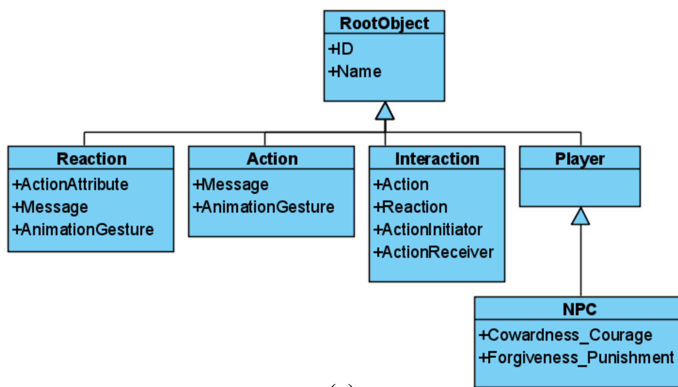
In a survey [7] of 30 projects that have used game engines we found that AI architectures and interfaces were used by 30% of the projects surveyed.

3. A NEW ARCHITECTURE FOR G-FACTOR PORTABILITY

This section contrasts a typical game development approach to the game development approach proposed in this work. Section 4.1 describes what is consider a typical development approach through the development of a sample game and highlights the dependencies associated with this approach. Section 4.2 then proposes a new approach to address these dependencies and describes an architecture called game space architecture (GSA) which has been implemented to validate this approach.

Table 1: Comparing a typical game development approach to GSA’s approach.

<table>
<tr><th>Step</th><th>Typical Approach</th><th>GSA’s Approach</th></tr>
<tr><td rowspan="2">1. Create the level data.</td><td colspan="2">• Create the decorative objects in the game engine.</td></tr>
<tr><td>• Create the game objects using the world builder or TorqueScript.</td><td>• Create the game objects using the world builder in the game engine and give them a unique ID which identifies these objects in the game space as well. Load these objects using TorqueScript.
• Create the game objects in the game space with the same unique ID using Jython.</td></tr>
<tr><td>2. Create the GUI.</td><td colspan="2">• Use the game engine interface builder or TorqueScript to create the interface. The behaviour is set as part of the game logic (step 4).</td></tr>
<tr><td>3. Create the object model.</td><td>• Use TorqueScript to extend the objects or create new ones.</td><td>• Create the object models for the game objects that require representation in the game engine and the game space.
• Create the other game objects models in game space.</td></tr>
<tr><td>4. Create the game logic.</td><td>• Use TorqueScript to set the behaviour in the game engine.</td><td>• Use Jython or Java to create the logic in the game space.</td></tr>
<tr><td>5. Create the adapter.</td><td></td><td>• Send the updates from the game engine to the game space.
• Create the adapter which translates between the game engine and the game space.</td></tr>
</table>



(a)

//Object Model: Create the Action object

```
function createAction(%ID,%Name,%Rate,%Message,%Gesture)
{
  %action = new SimObject()
  {
    ID=%ID;
    Name=%Name;
    Rate=%Rate;
    Message=%Message;
    Gesture=%Gesture;
  };
  $actionsArray.add(%action);
}
```

(b)

//Level Data: Add action and reaction game objects

```
createAction("40","action1","bad","You are an idiot!","looknw");
createAction("41","action2","good","Hi there!","celwave");
createReaction("80","reaction1","40","bad","No you are the
idiot!","looknw");
```

(c)

//Game Logic

```
function
calculateEmotionOutput(%actionType,%forgiveness,%punishment,
%cowardness,%courageness)
 {
   %actionWeight=0;
   if(%actionType$="verybad") {
     %actionWeight=-1;}
   else if(%actionType$="bad") {
      %actionWeight=-0.5;}
   …
```

(d)

Figure 3: Developing Moody NPCs using a standard game development approach.

3.1 A Typical Approach to Game Development

We will use a game we call 'Moody NPCs' to illustrate the typical approach to game development. The game consists of a number of non-player characters (NPCs) that react to a player based on their mood. The player can carry out actions such as greeting or swearing. Each NPC reacts to the action based on his mood which is governed by two variables: cowardness/courage and forgiveness/punishment. The game allows the user to navigate the level and click on an NPC which reveals its current mood and the actions available. The player can adjust the mood variables and try out different actions. The Torque game engine is used to demonstrate how the game is developed. The typical game development approach can be grouped into four main steps as shown in the typical approach column in Table 1.

To create the game level data Torque engine provides a level editor called World Editor. The level can also be created using other ways such as: scripting, API, configuration files, etc. The game level data contains the terrain of the environment and the decorative objects (e.g. houses, trees, etc). These objects can be exported from 3D modeling tools (e.g. 3D Studio Max) to Torque's format. The level also contains location markers for the game objects (e.g. NPCs and player). Scripting is used to create the other game objects (e.g. Reaction, Action, and Interaction) as shown in Figure 3c. This approach for creating the game level data is very common amongst game engines and as the surveys [7] showed 84% of the engines surveyed provide editors to create the game level.

Figure 4 shows the graphical user interface which has mood variables sliders on the top left corner of the screen and actions controller on the bottom left corner of the screen. The player can use the keyboard to navigate around and the mouse to select a NPC which reveals its mood variables and the actions. Torque has a GUI Editor to set most interface controllers. As with the game level data the interface can be created by other means such as scripting, and configuration files.

Figure 4: The interface created using GUI builder.

The third step is to create the object model to hold the structure for the game objects. The object model consists of five classes (see Figure 3a): Player, NPC, Action, Reaction, and Interaction. Torque has a default object model for the player and the AI player. These can be extended to add the properties that are specific to the game (i.e. mood variables to NPC). The extension and the creation of the other classes can be created using a static object model using either C++ or TorqueScript. The other game object models are created using scripting (see Figure 3b for an example). The final step is to create the game logic which controls how the NPC reacts to the player actions (see Figure 3d for an example).

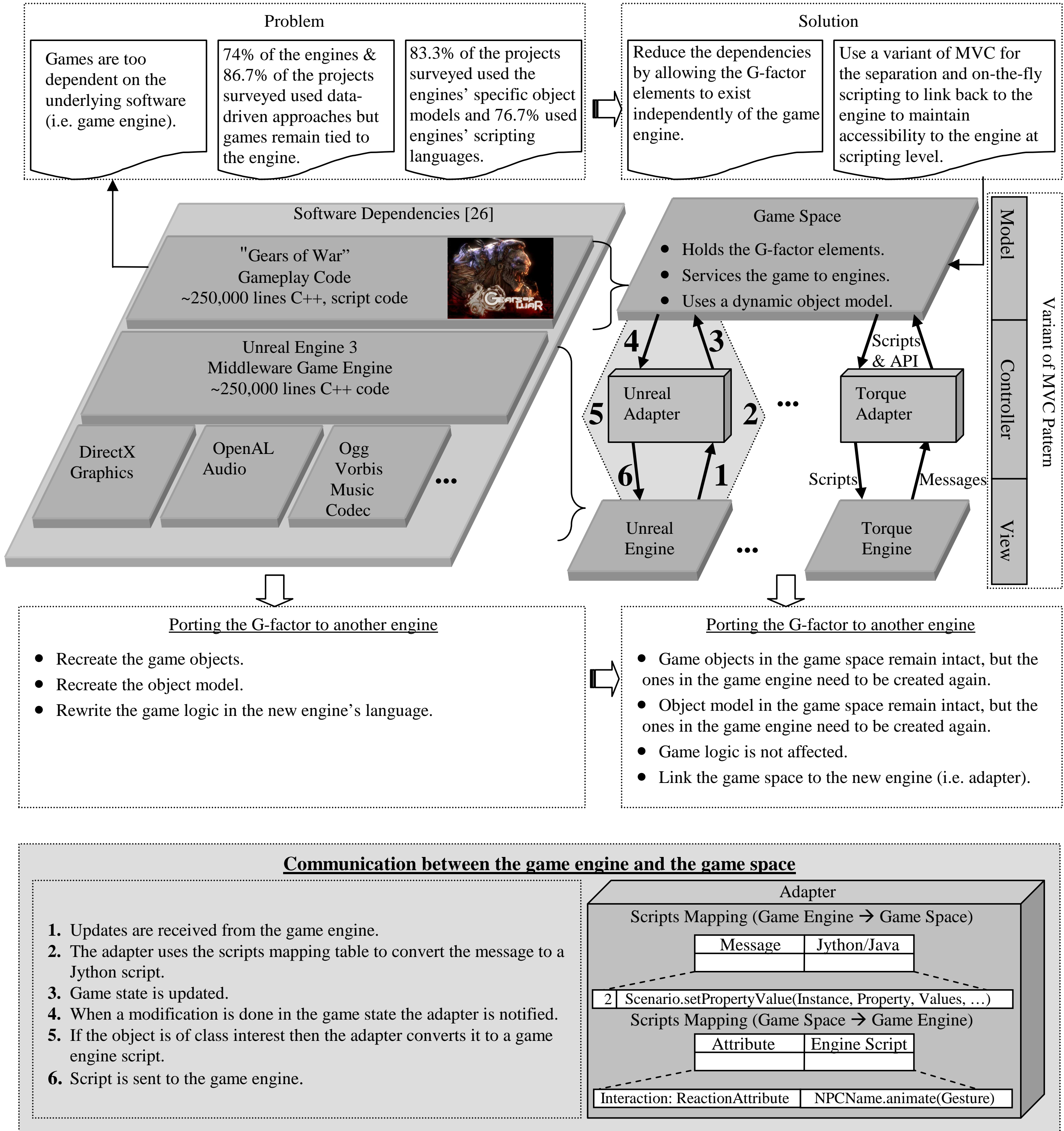


Figure 5: GSA overview.

3.2 GSA's Approach

Figure 5 illustrates the software dependencies problem the GSA is aiming to tackle. The example of dependencies is highlighted by the development of 'Gears of War' which is dependent on Unreal Engine 3 and the underlying software [26]. This is similar to the dependency the Moody NPCs game suffers from. Furthermore it is similar to the dependencies exhibited by the projects we surveyed [7].

GSA's objective is to reduce the dependencies by adopting a service-oriented design philosophy which enables the G-factor to exist independently of the game engine. The service-oriented approach has proved its practicality for achieving different types of portability such as platforms and languages [12]. The novel design approach employed in GSA combines a variant of the model-view-controller (MVC) pattern to separate the G-factor (i.e. model) from the game engine (i.e. view) with on-the-fly scripting to enable communication through an adapter (i.e. controller). The use of a variant of MVC rather than the normal MVC avoids a known liability which tight couples the view to the model [8]. The use of on-the-fly scripting is used to maintain the attractive attributes associated with a typical game development which uses data-driven mechanisms to modify the G-factor. Most notably modifiability is upheld in a typical game development approach using scripting which our surveys found to be very popular with game engines and projects that use game engines. To maintain this level of modifiability (i.e. scripting level access) to the game engine and the game space, GSA uses on-the-fly scripting to communicate with both via the adapter. For example a communication may begin with the game engine sending the updates to the adapter (step 1 in the communication protocol shown in Figure 5). The adapter converts them into scripts or direct API calls (step 2) which are then used to update the game space (step 3). When the game space needs to communicate with the game engine it notifies the adapter of the changes that need to be communicated (step 4). The adapter formats these into the engine's scripting language (step 5) and sends them to the engine to be executed (step 6). The separation and the communication mechanism allow the G-factor to exist independently of the game engine. The effect this has on portability means that when migrating to a new engine the elements in the game space (i.e. the game state, object model, and game logic) can stay intact. Contrasting this to migrating a game developed using a typical game development approach which often require all three to be created again shows the extent of the effort saved.

As was shown in Table 1 the first difference between this approach and the typical game development approach is the creation of the game objects which is split over the game engine and the game space due to the two types of game objects (see Figure 6). The first type are the game objects that have to have representations inside the game engine to provide visual representations such as the Player and the NPCs needed for the Moody NPCs game. These require real-time processing in the game engine and it is impractical to communicate every frame from the game space to the game engine. Therefore these objects have to be created in the game engine as well as the game space and only updates are communicated. The second type of the game objects are the ones that do not have representations inside the game engine such as the Action, Interaction, and Reaction objects. These objects can be created in the game space only. The object model creation is similarly split over the game engine and the game space (see Figure 7).

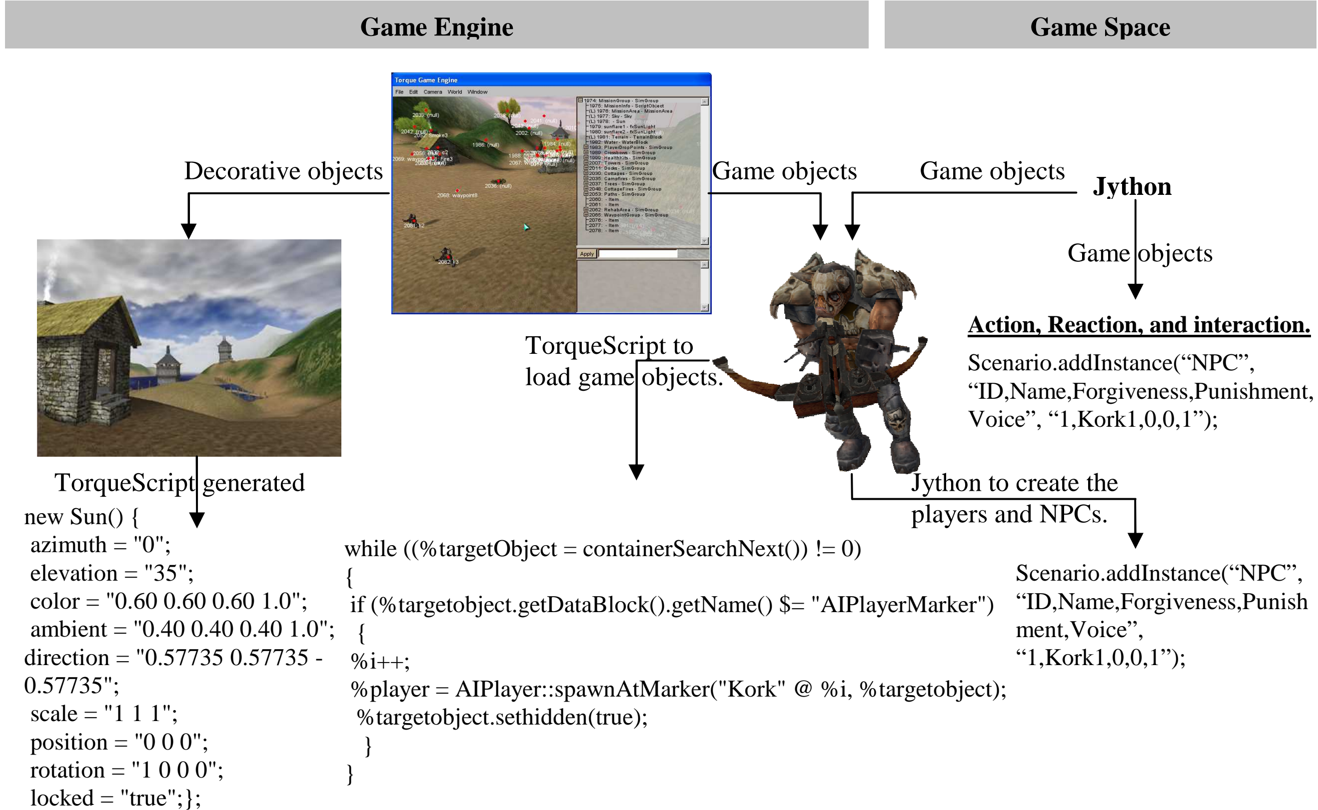


Figure 6: Creating the level data using GSA.

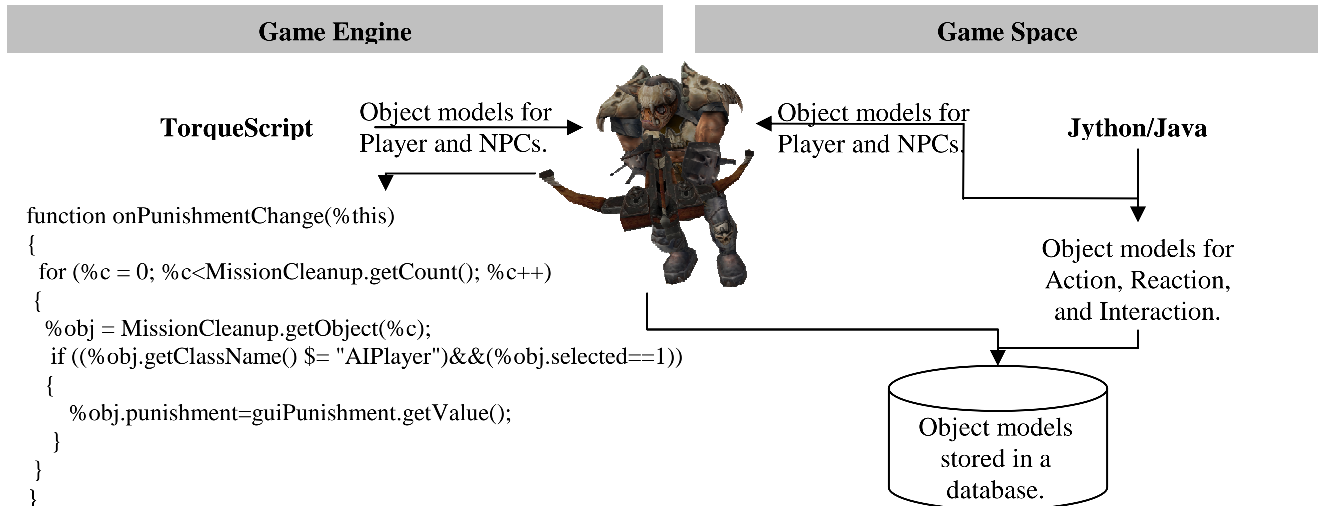


Figure 7: Creating the object models using GSA.

The second difference is creating the game logic in the game space rather than the game engine. The third difference is creating the adapter which handles the communication between the game space and the game engine.

4. EVALUATION AND DISCUSSION

To evaluate the architecture's ability to make the G-factor portable while maintaining the quality attributes of the typical game development (i.e. modifiability and performance) we conducted two types of evaluations: ad-hoc [4] and structured [5]. The ad-hoc evaluation showed that the same G-factor can be serviced to two different engines. The structured evaluation found the following issues associated with GSA:

- Performance is affected by the separation decisions made (MVC, scripting, and messaging) to achieve portability. Further performance tests on a small game showed that the average game space's CPU and memory overheads reported are: 1.51% and 16.49MB respectively. The game engine runs at almost the same CPU speed in both approaches.
- There is a danger if the message load increases that the game space becomes the bottleneck in the architecture. Further tests showed that the frames per second (FPS) reduces by 11.6% when using GSA's approach compared with the typical approach for a throughput of 55.49 messages per second.
- The data integrity across the different game states is at risk. Initial tests revealed no problems, but further tests are required before this can be established with certainty.

When contrasting the two development approaches it is obvious that the GSA approach requires extra work. Therefore, the choice between the two can be dependent on the project size. If the project is a prototype and if rewriting the game is not an issue then the typical development approach is more suitable for rapid prototyping. However if that is not the case, then the GSA development approach presents a better option due to combining the portability attribute with the modular design principle. This means that the need for portable G-factor can be considered as a whole or as a part. Projects that require the whole G-factor to be portable can adopt the whole architecture. But projects that are only capable of investing in part of the G-factor to be portable can choose to do so. Making only part of the G-factor portable may be the only option available for projects that choose an engine that does not provide some of the capabilities required by the architectural decisions. For instance the survey of the projects [7] that use game engines found that 51% of the projects chose Unreal engine for a variety of reasons despite the fact that it does not support on-the-fly scripting.

5. CONCLUSIONS

This work has examined portability in game engines and found three elements that are still lagging behind other portabilities which are object model, game logic, and game state which form the game's brain. The reasons behind their over dependencies on a game engine were demonstrated using a sample game development. The same sample game development was then contrasted with the GSA's approach to show how the migration process can be reduced. The effectiveness of this approach has been evaluated and the remaining challenges described. We have used this approach to develop a serious game which was used to train traffic accident investigators in Dubai police [6].

Dounis [11] predicts that gameplay is going to be the distinguishing factor between future games. This will generally mean an increase in the game size. Combined with the increased number of commercial licensees of game engines and the interest engines are receiving from outside the games industry (e.g. serious games community), this will increase the need for portable games which can become a very strong selling point due to two reasons. The first reason is because developers can keep the visual aspects of their game up to date with the latest game engine. The second reason is the security from having to face 'the RenderWare Problem'.

6. REFERENCES

[1] Adobbati, R., Marshall, A.N., Scholer, A., Tejada, S., Kaminka, G., Schaffer, S., Sollitto, C. Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research. Proceedings of the International Conference for Autonomous Agents, *Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada, (2001).

[2] Aha, D.W., Molineaux, M. Integrating learning in interactive gaming simulators. *Challenges of Game AI: Proceedings of the AAAI'04*

Workshop (Technical Report WS-04-04). San Jose, CA: AAAI Press, (2004).

[3] Berndt, C.; I.Watson; and H.Guesgen. OASIS: An Open AI Standard Interface Specification to Support Reasoning, Representation and Learning in Computer Games. *IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*. 31-July 2005, Edinburgh, 19-24.

[4] BinSubaih A., Maddock S. and Romano D.M. Game Logic Portability. *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology ACE 2005, Computer Games Technology session*, June 15-17th, Valencia, Spain, pp. 458-461, ISBN 1-59593-110-4.

[5] BinSubaih A., Maddock S. Using ATAM to Evaluate a Game-based Architecture. Workshop on Architecture-Centric Evolution (ACE 2006). Hosted at the *20th European Conference on Object-Oriented Programming ECOOP 2006* July 3-7, 2006, Nantes, France.

[6] BinSubaih A., Maddock S., Romano D.M. A Serious Game for Traffic Accident Investigators. *Special Issue of International Journal of Interactive Technology and Smart Education on "Computer Game-based Learning."*

[7] BinSubaih, A., Maddock, S., Romano, D. (2007). *A Survey of 'Game' Portability*. Department of Computer Science Technical Report CS-07-05, 2007, University of Sheffield.

[8] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Volume 1, (1996), John Wiley and Sons, ISBN: 0471958697

[9] Carless, S. (2007) Rise of the Game Engine. *Game Developer*, April, 2007, pp.2.

[10] Coyle, D. and Matthews, M. Personal Investigator: a Therapeutic 3D Game for Teenagers. *CHI2004* Vienna 25-29 April 2004. Presented at the Social Learning Through Gaming Workshop.

[11] Dounis, E. The Great Debate: Gameplay vs. Graphics, September 7th, 2006, http://www.gamersmark.com/articles/205/

[12] Erl,T. *Service-Oriented Architecture: Concepts, Technology, and Design*. ISBN: 0131858580 Publisher: Prentice Hall 8/2/2005

[13] Hussain, T.S. and Vidaver, G. Flexible and purposeful NPC behaviors using real-time genetic control. *Proceedings of the 2006 World Congress on Computational Intelligence,* July 16-20, Vancouver, BC.

[14] Jacobs, S., Ferrein, A. and Lakemeyer, G. Unreal Golog Bots. *In IJCAI'05 WS on Reasoning, Representation, and Learning in Computer Games*, 31-July 2005, Edinburgh, 19-24.

[15] Jankovic, L. Games Development in VRML. Virtual Reality 2000, 5. pp. 195-203.

[16] Kapolka, A. The Extensible Run-Time Infrastructure (XRTI): An Emerging Middleware Platform for Interoperable Networked Virtual Environments. *Proceedings of the Lake Tahoe Workshop on Collaborative Virtual Reality and Visualization*, October 2003.

[17] Khoo, A., Dunham, G., Trienens, N., Sood, S. Efficient, Realistic NPC Control Systems using Behavior-Based Techniques. *Proceedings of the AAAI 2002 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment*, Menlo Park, CA.

[18] Laird, J. It Knows What You're Going To Do : Adding Anticipation to a Quakebot. *Agents, 2001*, pp. 385-392

[19] Laird, J.E., Assanie, M., Bachelor, B., Benninghoff, N., Enam, S., Jones, B., Kerfoot, A., Lauver, C., Magerko, B., Sheiman, J. Stokes, D. Wallace, S. A Testbed for Developing Intelligent Synthetic Characters. *In Artificial Intelligence and Interactive Entertainment: Papers from the 2002 AAAI Spring Symposium*, Menlo Park, CA (2002).

[20] Lewis, M. and Jacobson, J. Game Engines in Scientific Research. Communications of the Association for Computing Machinery (CACM), NY: ACM 45(1), 2002.

[21] Nareyek, A., Combs, N., Karlsson, B., Mesdaghi, S., Wilson, I. The 2005 Report of the IGDA's Artificial Intelligence Interface Standards Committee. http://www.igda.org/ai/report-2005/report-2005.html. (accessed 5/5/2007)

[22] Oliveira, M., Crowcroft, J., Slater, M. An innovative design approach to build virtual environment systems. *Proceedings of the workshop on Virtual environments 2003*, ACM International Conference Proceeding Series; Vol. 39, Zurich, Switzerland Pages: 143 – 151, (2003) ISBN:1-58113-686-2.

[23] Smith, R. Essential techniques for military modeling and simulation. *Proceedings of the 30th conference on Winter Simulation*, (1998), 805 – 812, ISBN:0-7803-5134-7

[24] Smith, R. D. Strategic directions for distributed simulation. Simulation 2000 Series, 2, 1-9.

[25] Sweeney T. The Next Mainstream Programming Language: A Game Developer's Perspective. January 11-13, 2006 The 33rd Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages Charleston, South Carolina. www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt (accessed 5/5/2007)

[26] Vilhjalmsson, H., Samtani, and P. MissionEngine: Multi-system integration using Python in the Tactical Language Project. *PyCon 2005*, March 23-25, Washington, D.C. (2005).

[27] Wang, J., Lewis, M., and Gennari J. Emerging areas: urban operations and UCAVs: a game engine based simulation of the NIST urban search and rescue arenas. 35th Winter Simulation Conference, (2003), New Orleans, Louisiana, 1039-1045.

[28] Young, R.M., Riedl, M.O., Branly, M., Jhala, A., Martin, R.J., and Saretto, C.J. An architecture for integrating plan-based behavior generation with interactive game environments. *Journal of Game Development*, 1(1), 51-70, (2004).