Toward Quality-Driven Development of 3D Computer Games

T.C. Nicholas Graham and Will Roberts

School of Computing, Queen's University, Kingston, Canada, K7L 4L5
graham@cs.queensu.ca, wildwilhelm@gmail.com

Abstract. The development of video games is a complex software engineering activity bringing together large multidisciplinary teams under stringent constraints. While much has been written about how to develop video games, there has been as yet little attempt to view video game development from a quality perspective, attempting to enumerate the quality attributes that must be satisfied by game implementations, and to relate implementation techniques to those quality attributes. In this paper, we discuss desired quality attributes of 3D computer games, and we use the development of our own *Life is a Village* game to illustrate architectural tactics that help achieve these desired qualities.

1 Introduction

Gaming software sales grew to $24.5 billion world wide in 2004 [6], while in the United States alone, 228 million computer games were sold in 2005 [4]. The gaming industry has become a significant part of the software development world.

Games are challenging to develop. They involve complex algorithms in graphics, artificial intelligence, database and distributed systems, have stringent performance, usability and correctness requirements, and at the same time, are developed under aggressive delivery schedules. Game development teams are multidisciplinary, and for top titles include 100 or more people.

As yet, the software engineering literature has had little to say about how to develop games. In this paper, we discuss aspects of why developing games is different from developing other forms of software, and, motivated by a framework suggested by Bass et al. [1], we propose a set of architectural *tactics* that are helpful in game development. These tactics provide guidelines for how to structure games to address their quality requirements. The tactics are motivated and illustrated by our experience with the development of *Life is a Village*, a 3D computer-aided exercise game.

The paper is organized as follows. We first introduce *Life is a Village*, from which our examples will be drawn. We then discuss quality attributes of interest to games. Finally, we introduce our architectural tactics for game development and relate them to those quality attributes.

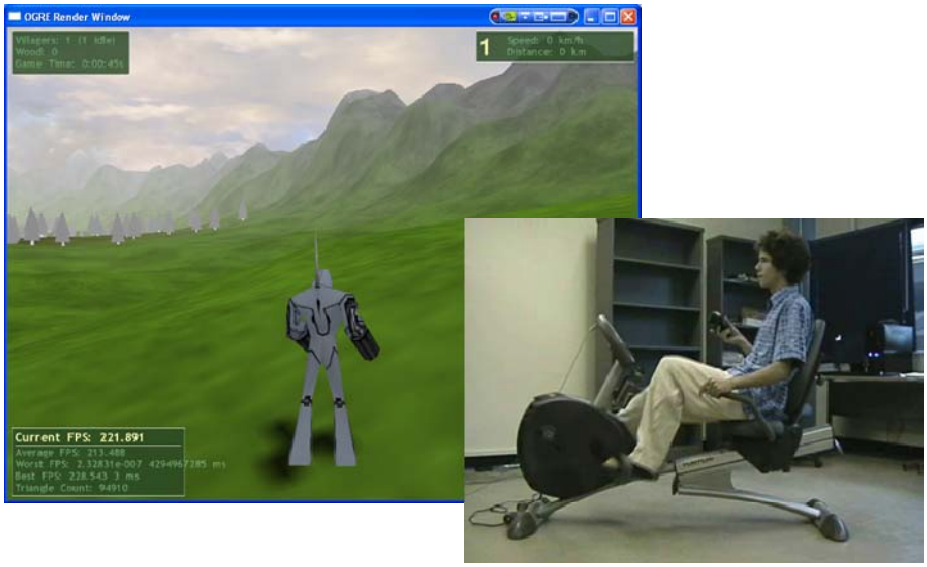

Fig. 1. Life is a Village game and player.

2 Life is a Village

Life is a Village is an experimental game testbed intended for exploration of computer-aided exercise [12], in which physical exertion is part of the game play.[1] The goal of the game is to gather resources from a large exterior landscape and use them to build an interesting village. The player traverses the landscape in search of resource nodes (such as wood, stone, etc.) When a node has been found, the player dispatches a villager from his/her village to start harvesting the resource. Once sufficient resources of the correct type have been collected, the player can add a new structure to the village.

The player uses a recumbent exercise bicycle to control the game (figure 1). Players navigate the terrain on their bicycle in the obvious way: pedaling moves forward; pedaling quickly moves forward quickly. Going uphill makes cycling harder; going downhill makes cycling easier. The player uses a handheld, wireless PS2 controller to steer, change gears, and provide button-based commands to the game. Exercise is an integral part of the game; the more players pedal, the faster they find resource nodes, the faster their villagers work, and therefore, the faster they can add to their village.

The core game framework has been implemented, but more work is to be done to make it a “fun” and playable game, such as adding additional village structures and

[1] More information on Life is a Village can be found at http://dundee.cs.queensu.ca/wiki/index.php/Life_is_a_Village

additional resource types. The development of this game motivates the tactics described in the remainder of this paper.

3 Quality Attributes for 3D Games

In this section, we review a number of quality attributes that are important to game development. This list is far from exhaustive, but serves as a representative set of qualities most important to game developers. Drawing from the presentation of Bass *et al.* [1], we select attributes divided into business time, development time and runtime. In section 4, we will then show how our architectural tactics address these quality attributes.

3.1 Business Qualities: Time to Market

Game developers face significant pressures to bring their products to market quickly. This pressure derives from a number of sources.

Games are often tied to events such as the release of a movie or the start of a sports season. For example, this year's Olympic Winter Games were accompanied by the Torino 2006 game; recent films such as King Kong, Spider-Man 2 and the Lord of the Rings trilogy have all been supported by video game releases. Each year sees the release of a profusion of football, hockey and soccer games featuring that season's players. Games must be released on schedule for the event with which they are associated, or risk losing their appeal.

Games that take a long time to develop risk falling behind the technology curve, leading to a spiral of further delays as artwork and special effects are updated to avoid appearing dated upon release. Additionally, console platforms have an expected lifetime of about five years, meaning that late releases risk catching their chosen platform on the decline.

Finally, the cost to develop a game for the next generation of consoles is estimated at $15-25 million [5]. Given such outlays, publishers face intense pressure to release quickly and begin recouping their investment.

3.2 Development-Time Qualities: Testability, Modifiability, Reusability

Modern computer games are complex and detailed, typically requiring tens of hours to complete. Games are highly graphical, and necessarily have non-deterministic behaviour. Some games have such complex artificial intelligence that their behaviour is "emergent", or unpredictable. All of these factors make games difficult to test. Games have stringent correctness requirements. Console games are distributed and played from a disk, so patches cannot be issued after the game's release. PC games, on the other hand, are routinely supported by patches, costing the publisher significant post-release development resources and distribution costs, as well as damaging its reputation. For example, the game Battlefield 1942, released in 2002, is currently

supported by over 270 MB of patches; Rome: Total War, released in 2004, requires over 130 MB of patches for correct play.

Modifiability is an important quality attribute of games. Games may evolve significantly during their development in the search for the elusive "fun" quality. Games are often extensively modified after their release as game expansions are developed. Massively multiplayer games evolve considerably over their online life, sometimes completely changing their character. Modifiability is a pre-cursor to reusability; the success of projects often relies on reuse of code from earlier projects.

3.5 Runtime Qualities: Usability, Performance

Usability of games differs significantly from that of other kinds of software. The main task of someone playing a game is to be entertained (or simply, to have *fun*.) Fun games routinely violate all normal rules for the design of usable systems. Games often provide players with information in inefficient forms, provide overly complex command interfaces and force players to perform low-level tasks that could quite reasonably be automated. However, a first-person shooter game that provided the player with the location of all enemies, a racing game that prevented players from losing control of their car, or a Tetris game that automatically chose the best location for a falling block would not be *fun*. Game usability must therefore balance the ease of learning and use of the game's interface with the fun that using the interface provides.

The primary performance metric in video games is *frame rate*, measured in frames per second (fps). A minimal value ensuring smooth animation is approximately 30 fps. A maximal value would match the refresh rate of the player's monitor; modern CRT monitors have a refresh rate of 75-85 Hz. Game players claim to be able to perceive the difference of frame rates up to 200 Hz, meaning that high frame rates may be necessary for marketing reasons even in cases where the benefit to game play is not clear.

Both measures of *average frame rate* and *worst frame rate* are important. Average frame rate gives a sense of how well the game is performing in general. Worst frame rate indicates how well the game does when under stress, perhaps the very time that player's require best performance.

4 Tactics for Game Development

Architectural choices can greatly influence a game's quality attributes. The trade literature provides diverse advice on how to architect games (e.g., for sports games [15], for massively multiplayer online games [13] and for real-time strategy games [11]). There is, however, little to help game developers choose broad architectural strategies in a principled manner. We advocate the use of *architectural tactics* [1] to help developers make informed architectural choices. Architectural tactics provide high-level advice for how to structure a software system. Tactics are not code or design patterns, but are higher-level, more generic techniques. Tactics influence quality attributes: a given tactic may improve one attribute while worsening another.

	Time to Market	*Testability*	*Modifiability*	*Reusability*	*Usability (Fun)*	*Performance (FPS)*
Create tools allowing non-programmers to engage in development	+			+	+	–
Decompose application into independent components	+	+	+	+		–
Structure application around existing components	+	+	⊙			⊙
Use scripting languages to allow rapid modification of game	+	+	+		⊙	+
Avoid blocking actions in main frame loop						+
Identify opportunities for parallel execution	–	–				+

Table 1. Architectural tactics for game development and quality attributes that they influence. '+' indicates a positive influence on the quality attribute, '-' a negative influence, and '⊙' a tuneable influence.

An architect can therefore analyze which tactics best meet the trade-offs required for his/her project. The approach of linking architectural tactics to software quality attributes has already been applied to human-computer interaction more broadly [2, 7], but not to game development.

Table 1 shows the tactics we propose for game development. This list should be viewed as a starting point; ultimately, our goal is to provide a rich set of tactics that developers can study before committing to a concrete architecture. These tactics were identified as a result of our experience with developing the *Life is a Village* game as well as consulting the game development trade literature. Developing a more complete set of will require the expertise of a wide group of game developers.

In sections 4.1 through 4.6, we review the six tactics presented in table 1, and show how they influence the quality attributes discussed in section 3. The tactics are illustrated with examples from the development of the Life is a Village game.

4.1 Tactic: Create tools allowing non-programmers to engage in development

To understand how games are developed, it is useful to consider the structure of game development teams. Table 2 shows the composition of the teams that developed five popular video games between 2002 and 2005. (The table was produced by consulting

	Battlefield 1942 (2002)	World of Warcraft (2004)	Civilization 4 (2005)	Battle for Middle Earth (2005)	F.E.A.R. (2005)
Producer	2	6	3	8	14
Designer	3	29	2	11	9
Writer			5	2	
Artist	12	41	34	39	18
Programmer	11	29	18	33	24
Audio	3	14		6	9
Video		35		7	3
Quality Assurance	51	114	26	73	56
Actor	18	36	1		16
Total	***100***	***304***	***97***	***179***	***149***

Table 2. Breakdown of development teams for five popular computer games.

the credits released by the games' publishers. For consistency, all people appearing in development roles in the credits are included in the table; no attempt was made to distinguish between part-time and full-time roles.)

Table 1 reveals three interesting points. First, game development teams for premiere (known as "AAA") games are large, involving upwards of 100 people. Second, these teams are highly interdisciplinary, involving design, story writing, creation of artwork, music, sound effects, voice acting, creation of video cut-scenes, programming and quality assurance. Programmers represent only 10%-20% of the development team. Third, the role of quality assurance is enormous, ranging over 25%-50% of the team's personnel.

The tasks of artists and designers include creating and animating entities that appear in the game world, designing the physical structure and appearance of game "levels" (interior or exterior), and scripting encounters between the players and the environment.

Since all of these tasks involve programming-like activities, one approach is to have the artists/designers specify the behaviour they desire, leaving programmers realize the specification. It is far better to allow non-programmers on the development staff to perform these tasks directly, without the involvement of programmers: artists can get faster turn-around on their ideas, and programmers cease to be a bottleneck in the process. All game development studios purchase at least some commercial tools to help empower artists, for example tools for modeling entities (e.g., Maya and SoftImage XSI) and tools for animation (e.g., Alias MotionBuilder). Larger studios can afford to build custom tools helping with other aspects of development.

This tactic helps with *time to market* by allowing artists/designers to be more productive. It helps *reusability*, since once developed, the tools can be used in future projects. *Usability* is enhanced, since designers can more quickly iterate between development and testing. Performance may be hindered as high-level tools may produce less optimized output than hand-crafted code.

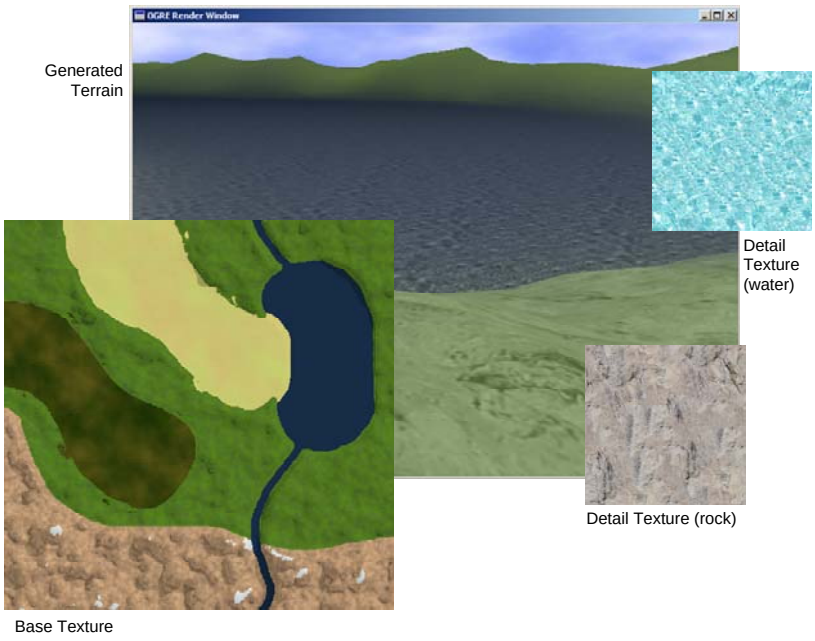

Fig. 2. A terrain consists of a polygon mesh with an overlaid base texture and detail texture

4.1.1 Illustration: Landscape Generation in Life is a Village

We applied this tactic in Life is a Village by developing a tool for procedural generation of landscapes. This allows people without programming skill to quickly develop rich 3D worlds. In 3D games, exterior landscapes are typically represented as a polygon mesh covered in a texture. The polygon mesh is covered in a *base texture*, an image that is stretched over the terrain's area. Often, a *detail texture* is blended with the base texture to give additional detail in the neighborhood close to the camera, reducing blurriness (figure 2).

Terrains can of course be created manually by a programmer by writing the appropriate DirectX or OpenGL commands to create and texture the terrain geometry. More realistically, artists use tools such as Leveller[2] and Terragen[3] to draw the 3D model of the terrain and to paint it with the desired texture. Such tools export a *heightmap* and a texture. The heightmap is a matrix specifying the height *y* of the terrain at each (*x*, *z*) point, and is used to generate the features of the terrain during the game's runtime. Terrain modeling tools such as these can lead to beautiful results, at the cost of significant manual labour.

[2] Leveller: http://www.daylongraphics.com/products/leveller
[3] Terragen: http://www.planetside.co.uk/terragen

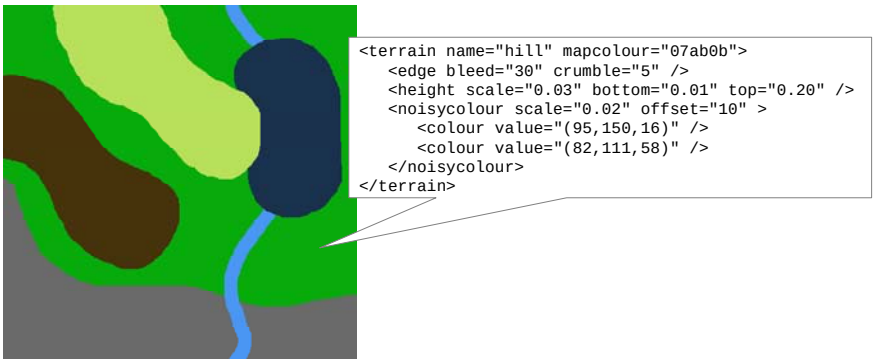

Fig. 3. Landscapes are generated from a terrain map, a simple bitmap image showing where each type of terrain is located.

For Life is a Village, we took an alternative approach of generating landscapes procedurally from a high-level description. This approach allows developers to quickly generate landscapes of arbitrary size, reducing time to market. Landscapes consist of numerous terrain types (e.g., hills, mountains, forest), each with differing properties such as height and coloration.

Figure 3 shows the inputs that a developer must provide to the terrain generation tool. The developer uses a paint program to create a bitmap representing where each terrain type appears. In the bitmap, terrain types appear representing mountains, hills, forest, plain, river and lake.

The properties of the terrain types are defined in XML. (Future plans involve building a simple GUI editor for terrain types.) Attributes of terrain types include the range of colours that can appear in the terrain, the height range of the terrain, the "noisiness" of the terrain (e.g., smooth, rolling hills vs jagged peaks), and properties allowing shadows to be pre-computed. The result of running the tool is a heightmap and a base texture. Figure 2 shows the result of running the inputs shown in figure 3, and an example of the rendered terrain.

4.2 Tactic: Decompose application into independent components

This tactic represents one of the fundamental lessons of software engineering, that it is important to decompose software system into components with well-defined interfaces that can be developed by different people. While this tactic is important to all large software products, it is of particular interest to the development of games, where large teams work under intense time pressure. Adopting this tactic, most modern games are based on a well-understood set of core components.

This tactic aids *time to market* by allowing parallel work, *testability* by providing hooks for unit testing, *modifiability* through localization of change, and *reusability* through the provision of components that may be modified for use in other games. *Performance* may be negatively impacted by rigid component interfaces or by components' information hiding, but may also be improved by algorithmic insights afforded by separation of concerns.

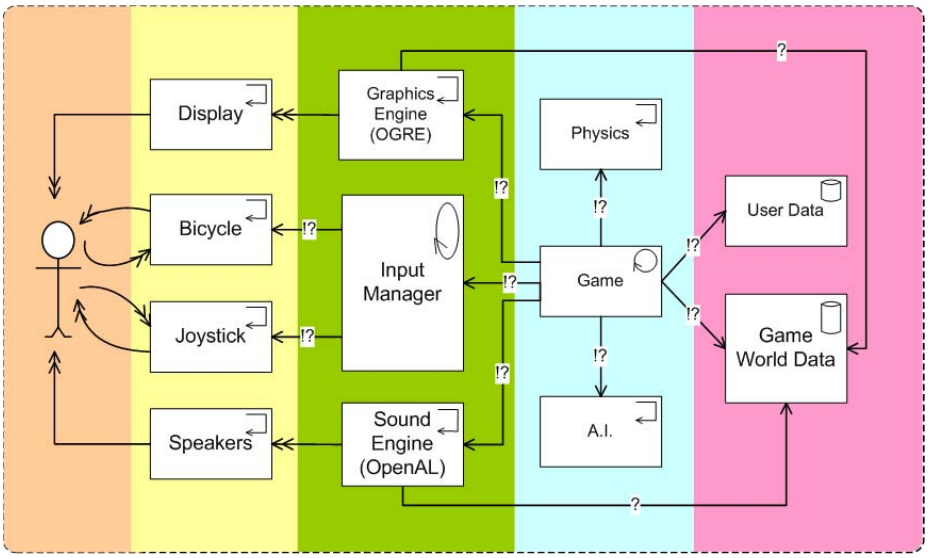

Fig. 4. The architecture of Life is a Village, in Workspace Architecture notation [9].

4.2.1 Illustration: Architecture of Life is a Village

Figure 4 shows the architecture of the Life is a Village game. This shows how the game is decomposed into high level components that can be given to different development teams. The components present in this architecture are typical of modern 3D games. The architecture is expressed in Workspace Architecture notation [9]. The core of the application is the *Game*, which runs in its own thread. The game takes input from various input devices, such as the bicycle and joystick. The input manager runs asynchronously in its own thread. Output is provided by calls to a *Graphics Engine*, which in turn updates the *Display*, and to a *Sound Engine*, which sends data to the *Speakers*.

The *AI* component is responsible for villager behaviour. The *Physics* component deals with collision detection and realistic behaviour of the player and non-player characters when jumping and falling.

The *User Data* and *Game World Data* components represent data about the player's state and the state of the game world itself.

4.3 Tactic: Structure application around existing components

A critical strategy for quickly developing complex games is the re-use of components from other projects, or the purchase of third-party components. Examples of highly successful third-party components include the Unreal game engine[4] and the Havok physics engine[5]. Reuse is critical to game development due to the importance of time

[4] Unreal Engine: http://www.unrealtechnology.com/html/technology/ue30.shtml
[5] Havok Physics Engine: http://www.havok.com

to market; there simply isn't time to build all components of a game from scratch. A significant part of the value of game development companies is the base of software they have available allowing them to develop new games quickly.

Reuse of components can help *time to market* by reducing code that has to be written, but can also increase time to market if the time to adapt the component to its new use is excessive, or if the component ultimately is a poor match with its requirements. Reuse helps with *testability* if the component has already been extensively tested in other contexts. As above, *modifiability* may be helped through localization of change, and *performance* may be either improved or worsened depending on the details of the components. Reuse may negatively impact usability through locking the developers into a particular style of gaming, or may improve usability by supporting varied and complex interaction styles that would be prohibitive to program from scratch.

4.3.1 Illustration: Use of open-source components

Life is a Village relies heavily on third-party components:

- The Object-Oriented Graphics Rendering Engine (*OGRE*)[6] is an open-source 3D graphics rendering engine that clearly illustrates the trade-offs of component use. While OGRE abstracts the low-level details of DirectX and OpenGL, dramatically reducing the effort of developing 3D graphics code, it has an incomplete feature set, third part add-ons of mixed quality, and difficulty integrating with commercial modeling tools.
- The Open Dynamics Engine (*ODE*)[7] is an open-source physics engine. ODE supports collision detection and correct physical behaviour of objects acting under force.
- *OpenAL*[8] is an open-source 3D sound engine adopted by such well-known titles as Doom 3 and Quake 4.

The gaming world has seen a strong convergence on what predefined components should be used and a perhaps surprisingly strong list of open-source tools. Additionally, an increasing number of companies have created strong niches in the development of third party tools for game development.

4.4 Tactic: Use scripting languages to allow rapid modification of game

Scripting languages have become a common technique for reducing the time to develop games and for reducing the skill level required of game developers. Games almost uniformly use C/C++ for core graphics, low-level AI and networking. Scripting languages such as Python or Lua [8] can then be used to encode the game play itself. Development of custom languages may be appropriate when domain information can be encoded in the language [10], but the cost of developing and maintaining custom languages may exceed their value [14]. Some games open their scripting languages to their player base, leading to a profusion of game enhancements produced and made available by players.

---

[6] OGRE 3D Graphics Engine: http://www.ogre3d.org
[7] Open Dynamics Engine: http://www.ode.org
[8] OpenAL: http://www.openal.org

```
IF at_tree AND NOT chop AND NOT drop_off_wood
  THEN chop AND NOT move

IF at_tree AND chop AND chop_timeout
    THEN NOT chop AND switch_targets
        AND reverse_path AND drop_off_wood AND move

IF drop_off_wood AND at_wood_drop_off
    THEN NOT drop_off_wood AND drop_wood
        AND switch_targets AND reverse_path
        AND go_to_tree AND move

IF NOT chop AND NOT drop_off_wood
    THEN go_to_tree AND move
```

Fig. 5. AI rules for a villager chopping wood and returning it to the village.

Scripting languages may improve *time to market*, as it is quicker to write and debug code in high-level languages. Time may be lost, however, to working around an awkward or poorly designed scripting framework, or one that is poorly supported by debugging tools. The *testability* of code may be improved, as scripting languages typically provide more runtime checking than raw C++ code. Scripts are typically high-level and interpreted, therefore more *modifiable* than low-level code. Since they support a fast code-execution cycle, scripts allow quicker refinement of gameplay mechanics, which may increase the *usability* of the final product. Scripting languages are typically slower than compiled code, so excessive use of scripting in time-critical areas may reduce *performance*.

4.4.1 Illustration: AI Scripting

Life is a Village uses a simple scripting language (adapted from Champandard [3]) to define villager behaviour. This allows behaviour to be quickly defined and changed, supporting rapid, experimental development. Figure 5 shows the rules specifying the behaviour of a villager whose job is to walk from the village to a tree, chop wood until his bag is full, then return to the village and drop off the wood.

The language is based on rules specified using propositional logic. A rule is triggered if its antecedent holds. Once triggered, the rule engine ensures that the rule's consequent holds. For example, the rule

```
IF NOT chop AND NOT drop_off_wood
    THEN go_to_tree AND move
```

will be triggered if the villager is not currently chopping wood or dropping off wood in the village. If triggered, the rule ensures that the villager is walking to the tree.

Rules are bound to the application via semantic actions; atoms in the antecedent query the game state, while atoms in the consequent may modify game state in order to make the consequent true.

This language helps collect AI decisions into one place, and allows villager AI to be modified without recompilation of the program. It also, however, illustrates problems with the scripting approach. When developers attempted to add more resources to the game, they discovered difficulties in generalizing the rules, since

there is no facility for parameterizing the resource being collected. The possible solutions included making many slightly modified copies of the rules, or burying the problem in the application through more powerful semantic actions. Neither approach was satisfactory, so the scripting language itself must be modified.

4.5 Tactic: Avoid blocking actions in main frame loop

Games are driven via a main loop responsible for computing the display for the next frame. The time taken to compute each frame is directly related to the time required to compute each iteration of this loop; e.g., to maintain a frame rate of 20 frames per second, frames must be computed within 50 ms. To optimize worst frame rate, this 50 ms must be treated as a soft real-time bound for each frame rather than an average to be achieved over the execution of the program.

In order to increase the game's frame rate, it is important to architect the main frame loop to contain no excessively lengthy computations, and particularly, no computations of unpredictable length.

4.5.1 Illustration: Input Handling

In traditional graphical user interfaces, input is handled via an event mechanism, where user inputs such as keystrokes and mouse button clicks are transmitted to the application via a callback mechanism (e.g., as provided by Java Swing's listener architecture.) Continuous inputs such as mouse motion are converted into a discrete set of events. In 3D games, inputs are instead handled by polling the input devices within the main frame loop. Thus if a game controller button is depressed or a joystick moved, the game will be able to react to the input within the main frame loop, and modify the game state appropriately. This approach of course requires a sufficiently high frame rate that the devices are polled often enough to provide responsive input.

In Life is a Village, one of our input devices is a Tunturi E6R recumbent bicycle. The bicycle can be polled for inputs representing the speed at which the user is cycling, the current tension of bicycle, what (if any) buttons the user is pushing, and the user's heart rate. Polling is performed via a proprietary protocol via a COM port link between the bicycle and computer.

Polling the bicycle takes a variable amount of time, ranging between 5 ms and 20 ms. Assuming the bicycle is polled once per frame, this time is added to the frame computation cost, unacceptably impacting frame rate. The solution, as shown in figure 4, is to run the input manager in its own thread. The input manager continuously polls the bicycle (and other input devices) in its own thread. When the main frame loop checks the input state, the input manager provides the last value obtained from the input device. Values from the bicycle may therefore be a few milliseconds out of date, but the result can be provided without blocking, and therefore without impacting frame rate.

4.6 Tactic: Identify opportunities for parallel execution

Modern gaming platforms support extensive parallelism. Microsoft's Xbox 360 game console provides three 3.2 GHz dual core PowerPC processors, or six cores in total, in a shared memory environment. Sony's forthcoming PlayStation 3 is built around a 3.2 GHz Cell Processor consisting of seven Synergistic Processing Elements (SPE's), each a 128 bit SIMD RISC processor, all connected by a 10 GBps bus. Desktop PC's are following the trend towards parallel architectures, with both Intel and AMD having scheduled quad-core CPU's for release in 2007. The challenge of programming this next generation of consoles is how to distribute the computation required in the game amongst these many processing elements.

The benefit of parallelism is a potential improvement in *performance.* Parallel programs are harder to write and debug, and therefore may negatively impact *time to market* and *testability.*

4.6.1 Illustration: Pathfinding

Pathfinding involves finding a reasonable path for agents in the game world that have to move from one location to another. For example, if a villager has to move from the village to a tree selected by the player, the game needs to first compute the route that the villager will follow. Path computations can be time-consuming, especially if there are many to do at the same time, and so make a good candidate for parallel execution. Additionally, path computation is not time-sensitive, in that a brief delay in computation will simply cause the villager to wait, playing an idle animation, before moving towards the tree. Pathfinding is mediated via a *CAXVillagerPathManager* component, which maintains a pool of threads that are assigned to a queue of path computation requests.

The six tactics presented in this section have shown how high-level approaches to architecting games can help meet quality requirements. The tactics each address one or more of the quality attributes identified in section 3, sometimes positively, and sometimes negatively. Relating tactics to quality attributes helps developers make reasoned architectural decisions.

5 Conclusion

In this paper, we have discussed quality attributes of interest to 3D video games, and proposed six tactics for addressing these quality attributes. The collection of tactics allows game developers to consider broad approaches to development in the context of how design choices affect game qualities. We illustrated the tactics through examples drawn from the development of the Life is a Village computer-aided exercise game.

Future work includes expanding the list of tactics and the quality attributes addressed. For example, we plan to consider tactics useful in the development of multi-player games.

Acknowledgements

We gratefully acknowledge the support of the National Science and Engineering Research Council in performing this work. The Life is a Village game benefited from the hard work of Irina Skvortsova, Rob Fletcher, Kevin Grad, Kevin Kassil, Joseph Lam, Banani Roy, Paul Schofield, and Sean Richards.

References

1. Len Bass, Paul Clements and Rick Kazman, Software Architecture in Practice, second edition, Addison-Wesley Professional, 2003.
2. Len Bass, Bonnie E. John, Natalia Juristo Juzgado, Maria Isabel Sánchez Segura, Usability-Supporting Architectural Patterns, in *Proceedings of the International Conference on Software Engineering*, pp. 716-717, 2004.
3. Alex J. Champandard, *AI Game Development,* New Riders Publishing, 2003.
4. Entertainment Software Association, *Top 10 Industry Facts*, 2005, Available at http://www.theesa.com/facts/top_10_facts.php
5. John J. Geoghegan, The Console Transition: A Publisher's Perspective, *BusinessWeek Online*, December 14, 2005.
6. Ronald Grover, Cliff Edwards, Ian Rowley and Moon Ihlwan, Game Wars, *BusinessWeek Online*, February 28, 2005.
7. Bonnie E. John, Len Bass, Maria Isabel Sánchez Segura and Rob J. Adams, Bringing Usability Concerns to the Design of Software Architecture. In *Proceedings of EHCI/DSVIS*, pp. 1-19, 2004.
8. Matthew Harmon, Building Lua into Games, in *Game Programming Gems 5*, pp. 115-128, Charles River Media, 2005.
9. W.G. Phillips, T.C.N. Graham and C. Wolfe, A Calculus for the Refinement and Evolution of Multi-User Mobile Applications In *Proceedings of Design, Specification and Verification of Interactive Systems*, Lecture Notes in Computer Science, pp. 137-148, 2005.
10. Falco Poiker, Creating Scripting Languages for Nonprogrammers, *AI Game Programming Wisdom*, Charles River Media, pp. 520-529, 2002.
11. Bob Scott, Architecting an RTS AI, *AI Game Programming Wisdom*, Charles River Media, pp. 397-401, 2002.
12. Brian K. Smith, Physical Fitness in Virtual Worlds, *IEEE Computer*, pp. 101-103, October 2005.
13. Shea Street, Massively Multiplayer Games using a Distributed Services Approach, in *Massively Multiplayer Game Development 2*, Charles River Media, pp. 233-241, 2005.
14. Paul Tozour, The Perils of AI Game Scripting, *AI Game Programming Wisdom*, Charles River Media, pp. 541-554, 2002.
15. Terry Wellmann, Building a Sports AI Architecture, in *AI Game Programming Wisdom 2*, Charles River Media, pp. 505-514, 2004.