Editor: Christof Ebert
Vector Consulting Services
christof.ebert@vector.com

Practices and Technologies in Computer Game Software Engineering

Walt Scacchi

Computer games are at the forefront of software engineering. Games software engineering, although often neglected in curricula, poses huge challenges such as time to market, complexity, collaborative development, and performance. Game development ranges from entertainment, with games being more sophisticated and complex than movies, to serious games used for education in universities and industry. Here, Walt Scacchi introduces us to computer game software engineering with technologies and a hands-on case study. I look forward to hearing from both readers and prospective column authors. —*Christof Ebert*

COMPUTER GAMES ARE RICH, complex, and frequently large-scale software applications. They're a significant, interesting, and often compelling domain for innovative research in software engineering (SE) techniques and technologies. Computer games are progressively changing the everyday world in many positive ways. Game developers, whether focusing on entertainment market opportunities or game-based applications in nonentertainment domains such as education, healthcare, defense, or scientific research (that is, serious games), thus share a common interest in how best to engineer game software.

Here, I examine aspects of contemporary computer game SE. To supplement this description, the sidebars present a brief look at game development technologies and a case study in applying computer game SE techniques.

What Game Developers Should Know

There are many different and distinct types of games, game systems, and gameplay, much like there are many dif-

GAME DEVELOPMENT TECHNOLOGIES

Computer games might well be the quintessential domain for computer science and software engineering R&D. Why? Modern multiplayer online games must address core issues in just about every major area of computer science research and education. Such games entail the development, integration, and balancing of software capabilities drawn from many areas. These areas include algorithm design and complexity, AI, computer graphics, computer-supported cooperative work or play, database management systems, human–computer interaction and interface design, OSs and resource or storage management, networking, programming- or scripting-language design and interpretation, and performance monitoring. Few other software application arenas demand such technical mastery and integration skill. Yet game development is expected to rely on such mastery and provide gameplay that most players find satisfying, fun, and engaging.

Computer games are thus an excellent domain for which to research and develop new ways and means for software engineering. Accordingly, there are many kinds of commercial or open source software development kits, engines, services, and approaches for producing, delivering, and evolving computer games of different genres.

Table A provides a small sample of possibilities that serve as a starting point. Interested software professionals and students should also go online and search for the software technologies that best match their interests in, constraints on, and enthusiasm for developing computer games.

TABLE A

Some technologies for computer game software engineering.*

Game SDK or game engine motif	Commercial examples	Game development features	Open source software alternatives	Development or target platforms	Common game genres
HTML5 or web	Construct 2, GameSalad	Rule-based, UI event processing	EaselJS, GDevelop, Kiwi.js, Phaser	Computers or devices with web browsers	2D web-browser-based games
Game-genre-specific	Adventure Game Studio, Minecraft, RPG Maker, SAGE	Genre-based UI, user experience	Freeciv, Minetest, Ren'Py, Quest, Stratagus	Networked PCs	Adventure and role-playing games, real-time strategy games, visual novels
Library, framework, or runtime environment	GameMaker, libGDX, Microsoft XNA	Game programming primitives, open APIs	ANX, Cocos2d, OGRE	PCs	2D or 3D single-user or multiuser games
Game modding (modifying)	Half-Life, Neverwinter Nights, Unreal	Modification or reuse of working games	Doom, Quake, Quake Arena	Networked PCs	Depends on the originating game
Game IDE	CryEngine, Source, Unity, Unreal Engine, UDK	Production quality workflow	Blender, Torque 3D	PCs	Mass-market games, 3D first-person action and shooter games
Cloud-based or MMOG service	Amazon Lumberyard, Facebook, Steam, Twitch	Scalable services and secure e-commerce	OpenSimulator, Worldforge	PCs, consoles, Internet-connected smartphones	eSports, free-to-play games, MMOGs

**SDK stands for software development kit; MMOG stands for massively multiplayer online game.*

THE *BEAM* GAME

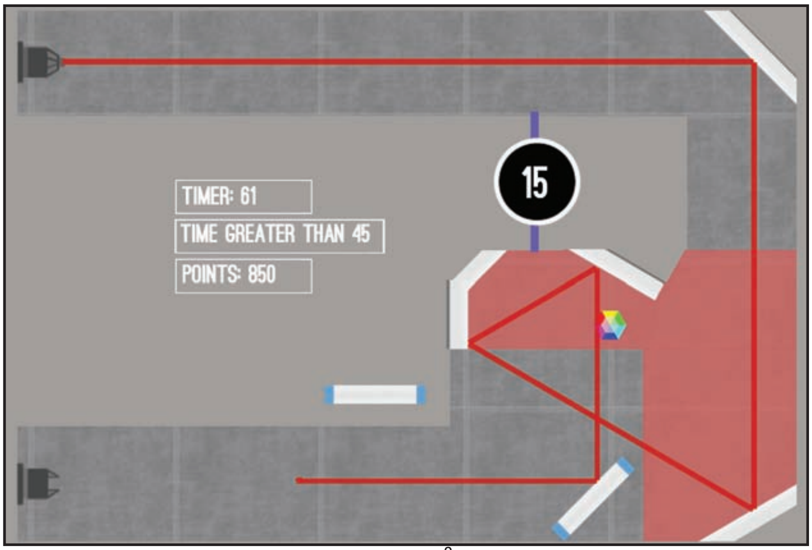

FIGURE A. A screenshot of the *Beam* game.[2] The player is placing mirrors to route an optical beam from the upper-left source to the lower-left target.

Case studies can help elucidate how to apply current practices and technologies in computer game software engineering (SE). In five previous case studies, I focused on software reuse and game repurposing.[1] Here, I focus on a study investigating whether a STEM-literate high-school student who was an avid gamer could learn basic SE concepts and practices.[2] (STEM stands for science, technology, engineering, and math.)

First, I had a student, Mark Yampolsky, identify a new game he would develop and demonstrate that could help his fellow students learn a challenging STEM topic—beam physics. Beam physics is central to modern physics—for example, in the design of simple optics as presented in high-school physics and as the basis for advanced particle accelerators and quantum teleportation devices.

The study proceeded in an agile incremental manner whereby the student identified functional or nonfunctional requirements that could be translated into game mechanics that could be realized using an event-driven, rule-based-system architectural framework. Such a framework is supported, for example, by 2D software development kits, such as Construct 2 and GameSalad (see the sidebar "Game Development Technologies"), that support rapid prototyping of interactive media or games for deployment with web browsers. By starting with an architectural framework and a software development kit, rather than a programming language, the student could focus on identifying play input and display output events and event types (mouse clicks, object drag-and-drop, game start and end, and so on). These events and event types could then trigger reactive rules that would update the gameplay (display) space and points earned (or lost).

Figure A shows a screenshot of the resulting *Beam* game. *Beam*'s multilevel goals entail finding either the shortest path or a path routed to achieve certain outcomes. One outcome could be to minimize where to place and how to orient (rotate) optical devices such as mirrors and lenses to ensure beam routing from the source to the target.

As the student developed *Beam*, issues or tradeoffs surfaced regarding how best to structure and refactor different rule sets. Overall, this version of *Beam* uses seven rule sets entailing more than 180 event-update rules.[2] Developing a new game with a rule-based system such as *Beam*'s presents a classic software engineering problem: refining and evaluating architectural alternatives. The *Beam* case study illustrates how SE concepts can mediate computer game development.

References

1. W. Scacchi, "Repurposing Gameplay Mechanics as a Technique for Developing Game-Based Virtual Worlds," *Computer Games and Software Engineering*, K.M. Cooper and W. Scacchi, eds., CRC Press, 2015, pp. 241–260.
2. M. Yampolsky and W Scacchi, "Learning Game Design and Software Engineering through a Game Prototyping Experience: A Case Study," *Proc. 5th Int'l Workshop Games and Software Eng.* (GAS 16), 2016, pp. 15–21.

ferent and distinct types of software applications, information systems, and systems for business. Understanding how to develop games for a particular platform requires identifying what types (genres) of games are commercially available. Popular genres include action games, first-person shooters, adventure games, role-playing games, fighting games, racing games, simulation games, sports games, strategy and real-time strategy games, music and rhythm games, parlor games (board and card games), puzzles, educational or training games, and massively multiplayer online games (MMOGs).

This suggests that knowledge about one type of game (for example, role-playing games such as *Dungeons & Dragons*) doesn't subsume, contain, or provide the gameplay experience, user interface, gameplay scenarios, or player actions in other types of games. So, being skilled in software development for one type of game (for example, a turn-taking role-playing game) doesn't imply ability or competence in developing software for another type of game (for example, a continuous-play twitch or action game). (Twitch games test players' reaction time.) This is analogous to saying that just because a developer is skilled in payroll and accounting software doesn't mean that the developer is skilled in enterprise database management or e-commerce systems. The differences can be profound, and the developer skills and expertise can be narrowly specialized.

Conversely, common games such as card or board games raise the obvious possibility of developing one game engine that can be shared or reused to support multiple games of a single type—a game product line. For example, checkers and chess are played on an 8 × 8 checkerboard, and player actions in both games are basically the same (picking up a piece and moving it to a square allowed by the game rules), even though the game pieces, rules, and gameplay differ.

So, being skilled in developing a checkers game can suggest having the skill to develop a similar game such as chess, especially if both games can use the same game engine (the game's runtime environment[1]). However, this is likely only when the game engine allows for distinct sets of game rules and the distinct appearance of game pieces. That is, the game engine must be designed for reuse or extension, which isn't always an obvious engineering choice and which increases game engine development's initial cost. Subsequently, developing the software for different kinds of games that are of the same type or use the same game engine requires a higher level of technical skill and competence than designing an individual game of a given type.

Understanding how game software operates on a game platform requires understanding gameplay and player actions. Understanding a game platform entails understanding an embodied game device (for example, the Apple iPhone, Microsoft Xbox One, or Nintendo Game Boy) and the internal software runtime environment that enables its intended operation and data communication. Developers must also understand the game's architectural structure, how the game functions, how the player controls the game device through its interfaces (keyboard, buttons, stylus, and so on) and video and audio output, and how the interfaces and output affect game data transmission and reception in a multiplayer game network.

Requirements Engineering

Understanding how best to elicit and engineer computer game requirements is unsurprisingly a fertile area for computer game SE research and practice, much as it has been for mainstream SE. However, relatively few game development approaches employ SE requirements development methods such as use cases and scenario-based design.

Many game developers in industry have reviewed the informal game postmortems that first appeared in *Game Developer* magazine in the 1990s and currently appear on Gamasutra.com. Austin Grossman's edited collection of 50 or so postmortems revealed common problems in game development projects.[2] These problems cluster around project software and content development scheduling, budget shifts (generally budget cuts), and other nonfunctional requirements that drift or shift in importance during game development projects. None of this should be surprising to experienced SE practitioners or project managers, although it might be new knowledge to SE students and new self-taught game developers.

Similarly, software functional requirements for computer games most often come from the game producers or developers, rather than game players. However, nonfunctional requirements (for example, the game should be fun to play but hard to master, and it should run on mobile devices and the web) dominate computer game development and thus marginalize the systematic engineering of functional game requirements. Nonetheless, the practice of openly publishing and sharing postproject descriptions and hindsight rationalizations might prove valuable as another kind of empirical SE data for further study, as well as something to teach and practice in SE education project courses (as I discuss later).

Architecture Design

Computer games often represent configurations of multiple software components, libraries, and network services. Consequently, computer game software must have an architecture,[1] and ideally this architecture is explicitly represented and documented as such. The architecture might be proprietary and thus protected by its developers as intellectual property covered by trade secrets and user license agreements. However, there's substantial educational value in having access to such architectural renderings as a means for quickly grasping key system design decisions and the modules that participate in gameplay event processing. This is one reason for interest in games that are open to modding (modifying) or free or open source software extensions.

But other architecture concerns exist. For example, networked multiplayer games employ at least four kinds of software or information architectures:

- the static and dynamic runtime architectures for a game engine,
- the architecture of the game development frameworks or software development kits that embed a game's development architecture together with its game engine,
- the architectural distribution of software functionality and data-processing services for the games, and
- the informational and geographical architecture of the game levels as designed play spaces.

Game system architecture can have different configurations. For instance, for the architectural distribution of software functionality and data-processing services, five system configurations are common:

- a single server for multiple interacting or turn-taking players;
- peer-to-peer networking;
- client-server networking for user clients and play-space data exchange servers;
- distributed, replicated servers for segmented user play sessions through sharding; and
- cloud-based game content delivery, a play event or score database, a game forum and chat or voice services, analytics, and commerce services.

In contrast, the focus on computer games as interactive media often sees little or no software architecture as being relevant to game design. This is especially true in the design of games that assume a single-server architecture or PC game runtime environment, rather than in game systems in which distributed services must be provided and system architecture is critical.[3] My point here is not to focus on the gap between game design and game software (architecture) design as consisting of alternative views but to draw attention to the need for computer game SE to finds ways to span the gap.

Playtesting

Computer games that potentially involve millions of players will consistently and routinely manifest bugs. Again, this is part of the puzzle of any complex SE effort; games are no exception. However, the user experience, and thus user satisfaction, might be key to driving viral social media that help promote game sales and adoption. So, paying close attention to bugs and features in game development and usability might be key to a game development studio's economic viability.

Furthermore, as decades of developer experience with large-scale software applications have shown, most users can't articulate their needs or requirements in advance but can assess whether what's provided meets their needs. So, the development of large-scale, high-cost computer games that take years to produce and person-decades (or person-centuries) of developer effort could change from monolithic product development lifecycles to ones that are much more agile, incremental, and driven by user feedback based on progressively refined or enhanced game version (or prototype) releases.

Early and ongoing game playtesting will likely become a central facet of computer game SE, as will tools and techniques for collecting, analyzing, and visualizing game playtesting data.[4] This is one area in which computer game SE might substantially diverge from early computer game development approaches, much like agile methods often displace the waterfall method. So, computer game developers, much like mainstream software engineers, are moving toward incremental development, rapid release, and user playtesting to drive new product versions.

Reuse and Repurposing

Systematic software reuse could be considered in multiple SE activities (requirements, architecture, design, code, build and release, and test cases) for a single game or game product line. For example, many successful games become franchises through the production and release of extension packs (that provide new game content or play levels) or sequels (for example, *Quake II* and *Quake III*). Whether or how to employ software-product-line concepts and methods in widespread computer game business models is unclear and underexplored. A new successful computer game product might have been developed and released in ways that sought to minimize software production costs. In that way, the software company could avoid the investment necessary to make the software architecture reusable and extensible and the component modules replaceable or upgradable without discarding much of the software developed up to that point. This means that SE approaches to computer game product lines might be recognized in hindsight as missed opportunities, at least for a given game franchise.

Reuse could reduce development costs and improve quality and productivity, as it often does in mainstream SE. Commercial computer game development often relies on third-party software components (for example, game engines) or middleware products (for example, AI libraries for nonplayer characters), which are perhaps its most visible forms of software reuse. Game software development kits, game engines, procedural game content generation tools, and game middleware services are all undergoing active R&D in industry and academia. (For additional details, see the sidebar "Game Development Technologies.")

Game engines are perhaps the best success story for computer game software reuse. However, commercial game development studios and independent game developers sometimes avoid adopting commercially available game engines because they believe that the engines' characteristic patterns or mechanics would overly restrict their game's development patterns or gameplay mechanics. If that happened, players might feel that such games offer a derivative play experience rather than an original one.

Nevertheless, the future of reusable game development techniques might include catalogs of patterns or antipatterns for game requirements, architecture and design patterns for game software product lines, and online repositories of reusable game assets organized by standardized ontologies. You can find other approaches to reuse in free or open source software for computer game development and in game software repurposing. Additional sources are the emerging AI or computational-intelligence methods for automated or semiautomated content generation and level design that appear in *IEEE Transactions on Computational Intelligence and AI in Games*.

> Computer games are a growth-oriented domain at the forefront of software engineering.

Runtime Services and Scalability Infrastructures

Computer games range from small-scale, standalone smartphone apps to large-scale, distributed, real-time MMOGs. They're sometimes played by millions of users, so large-scale, big data approaches to gameplay analytics and data visualization become essential for engineering sustained gameplay and deployment support.[4] Prior knowledge of the development of multiplayer-game software systems and networking services might be essential for SE students learning to develop social or mobile MMOGs.

To engage players and promote the adoption and ongoing use of such large and upward- or downward-scalable applications, computer game SE techniques have significant potential but require further articulation and refinement. Questions are just emerging regarding the integration of game playtesting and user play analytic techniques with large-scale, big data applications. Similarly, the question of how best to design back-end game data management capabilities or remote middleware gameplay services also points to SE challenges for networked-software-systems engineering, as has been recognized within the history of networked-game software development.

The ongoing emphasis on computer games that realize playful, fun, social, or learning experiences across gameplay platforms leads naturally to interdisciplinary approaches to computer game SE. In such approaches, psychologists, sociologists, anthropologists, and economists could provide expertise on defining new gameplay requirements and experimental designs to assess user experience quality.

Furthermore, the emergence of online fantasy sports, along with eSports (for example, team-versus-team or player-versus-player competitions for prizes or championship rankings) and professional-level tournaments for games such as *Counter-Strike*: *Global Offensive*, *Dota* 2, or *League of Legions*, points to other computer game SE challenges. These challenges include cheat prevention, latency equalization, statistical scoring systems, complex data analytics,[4] play data visualization, and streaming video broadcasts (for example, through MLG.TV or Twitch) that support balanced game systems with performance (monitoring) equalized for professional-level tournaments. Similarly, the emergence of games developed for VR or augmented-reality UIs and user experiences, such as *Pokemon Go*, suggests opportunities for engineering game software that exploits the new devices and sensors available through the UI, along with the engagement affordances these user experiences offer.

Computer Games and SE Education

SE faculty who teach project-oriented SE courses increasingly have sought to better motivate and engage students through game software development projects, as most CS students are literate in computer games and gameplay. The use of game development projects for SE capstone project courses is now widespread.

For educators teaching software engineering education (SEE) project courses, it might be valuable for their students to become engaged with computer game SE through exposure to the history of computer game software development or by reviewing recent advances in computer game SE fundamentals and SEE.[5] For example, C. Shaun Longstreet and Kendra Cooper, Alf Wang and Brian Wu, and others have incorporated contemporary SE practices such as software architecture and model-driven development within game-based SE project courses.[5] In addition, Tao Xie and his colleagues at Microsoft and others have developed game-based software-testing competitions.[5]

Similarly, whether to structure projects as massively open online courses or competitive, interteam game jams also merits consideration. Such competitions can be testbeds for empirical SE (or SEE) studies—for example, when project teams consist of students who take on different development roles, with each team comprising members with comparable roles and experience.

Computer game SE is a growth-oriented domain at the forefront of software engineering. A new generation of software engineers will take on the technical challenges involved in facilitating the development, deployment, and evolution of computer games as complex software systems that support global cultural-media practices. Readers interested in further exploring computer game SE practices and technologies might also find the cited references helpful for learning about game design practices,[3] common approaches and mistakes in game production,[2] game engine and runtime environment architectures,[1] gameplay data analytics and visualization techniques,[4] or recent advances in computer game SE research and education.[5]

References

1. J. Gregory, *Game Engine Architecture*, A K Peters / CRC Press, 2009.
2. A. Grossman, ed., *Postmortems from* Game Developer*: Insights from the Developers of* Unreal Tournament, Black and White, Age of Empires, *and Other Top-Selling Games*, Focal Press, 2003.
3. J. Schell, *The Art of Game Design: A Book of Lenses*, CRC Press, 2008.
4. M. Seif El-Nasr, A. Drachen, and A. Canossa, eds., *Game Analytics: Maximizing the Value of Player Data*, Springer, 2013.
5. K.M. Cooper and W. Scacchi, eds., *Computer Games and Software Engineering*, CRC Press, 2015.

WALT SCACCHI is a senior research scientist and research faculty member at the Institute for Software Research and the director of research at the Institute for Virtual Environments and Computer Games, both at the University of California, Irvine. Contact him at wscacchi@ics.uci.edu.