Computer games (CG) are rich, complex, and often large-scale software applications. CG are a significant, interesting, and often compelling software application 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 non-entertainment domains like education, healthcare, defense, or scientific research (i.e., “serious games”), thus share a common community of interest into how best engineer game software.

There are many different and distinct types of games, game systems, and game play, much like there are many different and distinct types of software applications, information systems, and system used for business. Understanding how games as a software system are developed to operate on a particular game platform requires identifying what types of games (i.e., game genre) are available in the market. Popular game genre include action/first-person shooters, adventure, role playing game (RPG), fighting, racing, simulations, sports, strategy and real-time strategy, music and rhythm, parlor (board and card games), puzzles, educational/training, and massively multiplayer online games. This suggests that knowledge about one type of game (e.g., role playing games like Dungeons and Dragons) does not subsume, contain, nor provide the game play experience, player control interface, game play scenarios, or player actions found in other types of games. So being skilled in the art of one type of game software development (e.g., building a turn-taking RPG) does not imply ability or competent skill in developing another type of game software (e.g., a continuous play twitch/action game). This is analogous to saying that if a software developer is skilled in “payroll and accounting” software application systems, this does not imply such a developer is also competent or skilled in the development of “enterprise database management” or “E-commerce product sales over the Web” systems. The differences can be profound, and the developer skills and expertise narrowly specialized.

Conversely, games of a common type, like card or board games, raise the obvious possibility for a single game engine to be developed and shared/reused to support multiple game kinds of a single type—a game product-line family. For example, the games Checkers and Chess are played on an 8X8 checkerboard, though the shape and appearance of the game play pieces differs, and the rules of game play differ, though the kinds of player actions involved in playing either Chess or Checkers are the same (picking a piece and moving it to a square allowed by the rules of the game). So being skilled in the art of developing a Checkers game can suggest the ability or competent skill in developing a similar game like Chess, especially if both games can use the same game engine. However, this is likely only when the game engine (the game's run-time environment [2]) is designed to allow for distinct sets of game rules and distinct appearance of game pieces—that is, the game engine must be designed for reuse or extension, which is not always an obvious engineering choice, and it is one that increases the initial cost of game engine development. Subsequently, developing the software for different kinds of games of the same type, or using the same game engine, requires a higher level of technical skill and competence in software development than designing an individual game of a given type.

Understanding how game software operates on a game platform requires understanding game play and game player actions. Understanding a game platform entails understanding an embodied game device
(e.g., Apple *iPhone*, Microsoft *Xbox One*, Nintendo *GameBoy*) and the internal software run-time environment that enables its intended operation and data communication capabilities. It also requires how the game is designed in terms of its architectural structured and how it functions, as well as how the game player controls the game device through its interfaces (keyboard, buttons, stylus, etc.) and video/audio displays, and how they affect game data transmission and reception in a multiple player game network.

A sample of techniques and technologies in CGSE that inform contemporary CGSE studies include the following. A Case Study in the application of CGSE techniques, and a sample of Game Development Technologies appear in respective sidebars in this article.

Game software requirements engineering

Understanding how best to elicit and engineer the requirements for CG is unsurprisingly a fertile area for CGSE research and practice, much like it has been for mainstream SE. However, there are still relatively few game development approaches that employ SE requirements development methods like use cases and scenario-based design.

Many game developers in industry have reviewed the informal game “post mortems” that first began to be published in *Game Developer* magazine in the 1990’s, and more recently on the *Gamasutra.com* online portal. Grossman's [3] edited collection of fifty or so postmortem best reveal common problem that recur in game development projects, which cluster around project software and content development scheduling, budget shifts (generally development budget cuts), and other non-functional requirements that drift or shift in importance during game development projects. None of this should be surprising to experienced SE practitioners or project managers, though it may be “new knowledge” to SE students and new self-taught game developers. Similarly, software functional requirements for CG most often come from the game producers or developers, rather than from end-users. However, non-functional requirements (e.g., the game should be fun to play but hard to master; game should run on mobile devices and the Web) dominate CG development efforts, and thus marginalize the systematic engineering of functional game requirements. Nonetheless, the practice of openly publishing and sharing post-project descriptions and hindsight rationalizations may prove valuable as another kind of empirical SE data for further study, as well as something to teach and practice within SE Education project courses (see below).

Game software architecture design

CG as complex software applications often represent configurations of multiple software components, libraries, and network services. As such, CG software must have an architecture [2], and ideally such an architecture is explicitly represented and documented as such. While such architecture may be proprietary and thus protected by its developers as intellectual property covered by trade secrets and end-user license agreements, there is substantial educational value in having access to such architectural renderings as a means for quickly grasping key system design decisions and participating modules in game play event processing. This is one reason for interest in games that are open to modding or free/open source software extensions. But other software architecture concerns exist. For instance, there are at least four kinds of CG software architecture that arise in networked multiplayer games: (a) the static and dynamic run-time architectures for a game engine; (b) the architecture of the game development frameworks or SDKs that embed a game’s development architecture together with its game engine; (c) the architectural distribution of software functionality and data processing services
for networked multiplayer games; and (d) the informational and geographical architecture of the game levels as designed play spaces. For example, for (c) there are four common alternative system configurations: single server for multiple interacting or turn-taking players; peer-to-peer networking; client-server networking for end-user clients and playspace data exchange servers; and distributed, replicated servers for segmented user play sessions via “sharding.”

In contrast, the focus on CG as interactive media often sees little/no software architecture as being relevant to game design, especially for games that assume a single server architecture or PC game run-time environment, or in a distributed environment that networking system specialists are assumed will design and provide [4]. Ultimately, our point is not to focus on the gap between game design and game software (architecture) design as alternative views, but to draw attention to the need for CGSE to finds ways to span the gap.

**Game software playtesting**

CG as complex software applications for potentially millions of end-users will consistently and routinely manifest bugs. Again, this is part of the puzzle of any complex SE effort, so games are no exception. However, as user experience (UX) and thus user satisfaction, may be key to driving viral social media that helps promote retail game sales and adoption, then paying close attention to bugs and features in CG development and usability may be key to the economic viability of a game development studio. Further, again from knowledge of decades of experience in developing large-scale software applications, most end-users cannot articulate their needs or requirements in advance, but can assess what is provided in terms of whether or not it meets their needs. This in turn may drive the development of large-scale, high cost CG that take calendars to produce, and person-decades (or person-centuries) of developer effort away from monolithic product development life cycles, to ones that are much more incremental and driven by user feedback based on progressively refined/enhanced game version (or prototype) releases. Early and ongoing game playtesting will likely come to be central facet of CGSE, as will tools and techniques for collecting, analyzing, and visualizing game playtesting data [5]. This is one activity where CGSE efforts going forward may substantially diverge from early CG software development approaches, much like agile methods often displace “waterfall” software life cycle development approaches. So CG developers, much like mainstream software engineers are moving towards incremental development, rapid release, and user playtesting to drive new product release versions.

**Game software reuse and repurposing**

Systematic software reuse could be considered within multiple SE activities (requirements, architecture, design, code, build and release, test cases) for a single game or a product-line of games. For example, many successful CG are made into “franchise brands” through the production and release of extension packs (that provide new game content or play levels), or product line sequels (e.g., *Quake, Quake II*, and *Quake III*). Whether or how software product lines concepts and methods can be employed in widespread CG business models is unclear and under-explored. A new successful CG product may have been developed and released in ways that sought to minimize software production costs, thus avoiding the necessary investment 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 CG product lines may be recognized in hindsight as missed opportunities, at least for a given game franchise.
Reuse has the potential to reduce CG development costs and improve quality and productivity, as in it often does in mainstream SE. Commercial CG development relies often on software components (e.g., game engines) or middleware products provided by third parties (AI libraries for NPCs), as perhaps its most visible form of software reuse practice. Game software development kits (SDKs), game engines, procedural game content generation tools, and game middleware services all undergoing active R&D within industry and academia. See Game Development Technologies sidebar for additional details.

Game engines are perhaps the best success story for CG software reuse, but it is often the case that commercial game development studios and independent game developers avoid adoption of such game engines when they are perceived to overly constrain game development patterns or choice of game play mechanics, to those characteristic of the engine. This means game players may recognize such games as offering derivative play experience, rather than as original play experiences. However, moving to catalogs of pattern/anti-patterns for game requirements, architecture and design patterns for game software product lines, and online repositories of reusable game assets organized by standardized ontologies, may be part of the future of reusable game development techniques. Other approaches to software reuse may be found in free/open source software for CG development and in game software repurposing, as well as emerging methods in AI/computational intelligence methods for (semi) automated content generation and level design advances found in the IEEE Transactions on Computational Intelligence and AI in Games.

Game run-time services and scalability infrastructures

CG range from small-scale, standalone applications for smartphones (e.g., app games) to large-scale, distributed, real-time, applications massively multiplayer online games (MMOGs). CG are sometimes played by millions of end-users, so that large-scale, “big data” approaches to game play analytics and data visualization become essential techniques for engineering sustained game play and deployment support [5]. Prior knowledge of the development of multiplayer game software systems and networking services may be essential for CGSE students focusing on development of social/mobile MMOGs. In order to engage the users and promote the adoption and ongoing use of such large and upward/downward scalable applications, CGSE techniques have significant potential, but require further articulation and refinement. Questions on the integration of gameplay playtesting and end-user play analytic techniques together with large-scale, big data applications are just beginning to emerge. Similarly, how best to design backend game data management capabilities or remote middleware game play 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 CG that realize playful, fun, social, or learning game experiences across different game play platforms leads naturally to interdisciplinary approaches to CGSE, where psychologists, sociologists, anthropologists, and economists could provide expertise on defining new game play requirements and experimental designs to assess the quality of user play experiences. Further, the emergence of online fantasy sports, along with eSports (e.g., team/player vs. team/player competitions for prizes or championship rankings) and commercial/international endeavors for professional level game play tournaments for games like CounterStrike: Global Operations, DOTA2 or League of Legions point to other CGSE challenges like cheat prevention, latency equalization, and statistical scoring systems, complex data analytics [5], play data visualizations, and streaming video broadcasts (e.g., via MajorLeagueGaming.tv or Twitch) that support game systems that are balanced and performance (monitoring) equalized for professional-level tournaments. Similarly, the emergence of games developed for virtual reality or augmented reality user interfaces (UI) and user experiences (UX) like Pokemon Go, also point to emerging opportunities for engineering game software that takes
advantage of the new devices and sensors available through the UI, along with engagement affordances that these UX offer.

**Computer games and software engineering 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 worldwide are literate in CG and game play. The popularity of encouraging game development projects for SE capstone project courses is now widespread.

For those who teach SEE project courses, it may be valuable for students to become engaged with CGSE through exposure to the history of computer game software development, or by review of recent advances in CGSE fundamentals and SEE [1]. Other emerging SEE practices are articulated by Cooper and Longstreet, Wang, and others expand their visions for SEE by incorporating contemporary SE practices like software architecture and model-driven development into software project courses, or game-based software testing competitions developed by Xie, Tillman and others [1]. Similarly, whether to structure the SEE coursework projects as massively open online courses (MOOCs), or around competitive, inter-team game jams also merits consideration. Such competitions can serve as testbeds for empirical SE (or SEE) studies, for example, when project teams are composed by students who take on different development roles, and each team engages members with comparable roles and prior experience.

In conclusion, the emerging field of computer games and software engineering represents a growth-oriented domain where a new generation of software engineers will take on 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 practices and technologies for CGSE may also find the suggested references helpful for learning about game design practices [4], common approaches and mistakes in game production [3], game engine and run-time environment architectures [2], game play data analytics and visualization techniques [5], or recent advances in CGSE research and education [1].

**References**


**Sidebar #1: Game development technologies**

Computer games may well be the quintessential domain for Computer Science and SE research and development practice. Why? Modern multi-player online games (MMOG) must address core issues in just about every major area of CS research and education. Such games entail the development, integration, and balancing of software capabilities drawn from algorithm design and complexity, artificial intelligence, computer graphics, computer-supported cooperative work/play, database management systems, human computer interaction and interface design, operating systems and resource/storage management, networking, programming or scripting language design and interpretation, performance monitoring, and more. Few other software system application arenas demand such technical mastery and integration skill. Yet game development is expected to rely on such mastery, and provide a game play user experience that most users find satisfying, fun, and engaging. Computer games are thus an excellent domain for which to research and develop new ways and means for (game) software engineering. Accordingly, there are many different kinds of commercial/open source software development kits (SDK), engines, services, and approaches for producing, delivering, and evolving computer games of different genre. The table provides a small sample of possibilities that serve as a starting point. But the interested software professional or student should go online and search for the software technologies that best match their interests, constraints, and enthusiasm for participating in the development of computer games today and into the future.

<table>
<thead>
<tr>
<th>Game SDK or game engine motif</th>
<th>Commercial Examples</th>
<th>Game development features</th>
<th>Open source software alternatives</th>
<th>Development or target platforms</th>
<th>Common game genres</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HTML5/Web</strong></td>
<td>Construct 2, GameSalad</td>
<td>Rule-based, UI event processing</td>
<td>EaselJS, GDevelop, Kiwi.js, Phaser</td>
<td>Computers or devices with Web browsers</td>
<td>2D web-browser based</td>
</tr>
<tr>
<td><strong>Game genre specific</strong></td>
<td>Adventure Game Studio, Minecraft, RPG Maker, SAGE</td>
<td>Genre-based User Interface and user experience</td>
<td>FreeCiv, Minetest, Ren'Py, Quest, Stratagus</td>
<td>Networked personal computer</td>
<td>Adventure, role-playing games, RTS, visual novel</td>
</tr>
<tr>
<td>Library, Framework, or RunTime Environment</td>
<td>Gamemaker, LibGDX, Microsoft XNA</td>
<td>Game programming primitives, open APIs</td>
<td>ANX, Cocos2d, OGRE</td>
<td>Personal computers</td>
<td>2D/3D single/multi user</td>
</tr>
<tr>
<td>Game modding</td>
<td>Half-Life, NeverWinter Nights, Unreal</td>
<td>Modify/reuse working games</td>
<td>Doom, Quake and Quake Arena</td>
<td>Networked personal computers</td>
<td>Depends on originating game</td>
</tr>
<tr>
<td>Integrated Game IDE</td>
<td>CryEngine, Source, Unity, Unreal Engine and UDK</td>
<td>Production quality workflow</td>
<td>Blender, Torque3D</td>
<td>Personal computers</td>
<td>Mass market, 3D first-person action/shooter games</td>
</tr>
<tr>
<td>Cloud-based or MMOG services</td>
<td>Amazon Lumberyard, Facebook, Steam, Twitch</td>
<td>Scalable services and secure online commerce</td>
<td>OpenSimulator, World Forge</td>
<td>PCs, consoles, smartphones connected to Internet</td>
<td>eSports, free-to-play, MMOG</td>
</tr>
</tbody>
</table>
**Sidebar #2: Case Study in CGSE—The BEAM game**

Case studies can serve to help elucidate how current practices and technologies in CGSE can be applied. A sample of five case studies that focus on software reuse and game repurposing practices can be found elsewhere [1]. Here we focus on a single study where the core CGSE challenge was determining whether a STEM-literate high school student who was an avid gamer could learn basic SE concepts and practices [2]. The case study began with the student being tasked to identify a new game that he would develop and demonstrate that could help his fellow students to learn a challenging STEM topic, which in this case was beam physics. Beam physics is central to modern physics, such as the design of simple optics as presented in high school physics, as well as the basis of advanced particle accelerators. This study highlights development of the BEAM game, with more details elsewhere [2].

The study proceeded in an agile incremental manner whereby (non)functional requirements were identified, 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 SDKs like Construct 2, GameSalad, or others (see Game Development Technologies) that support rapid prototyping of interactive media/games for deployment with Web browsers. Starting with an architectural framework and SDK, rather than with a programming language, allowed the student to focus attention to identifying play input and display output events and event types (mouse clicks, object drag-and-drop, game start/end, etc.) that could then trigger reactive rules that would update the game play (display) space and points earned (lost). Figure 1 shows a view of the game space during a sample BEAM game, whose multi-level goals entail finding either the shortest path, or a path routed to achieve certain outcomes, like minimizing where to place and how to orient (rotate) optical devices like mirrors and lenses to insure beam routing from source to target. Along the way, issues/trade-offs in how best to structure and refactor different rule-sets were surfaced. Overall, this version of BEAM utilizes seven rule sets entailing more than 180 event-update rules [2]. As such, developing a new game with a rule-based system presents a classic problem in SE: refining and evaluating architectural alternatives. The development of BEAM serves as a case study in how SE concepts can mediate development of a CG.

Figure 1. A screen display of BEAM during play, where the user must route the beam moving from the upper left source to the lower left target while placing mirrors to establish a working beam path [2].

Biography
Walt Scacchi is senior research scientist and research faculty member in the Institute for Software Research, and also Director of Research at the Institute for Virtual Environments and Computer Games, both at University of California, Irvine. He served as Co-Chair for Software Engineering in Practice Track at the 33rd Intern. Conf. Software Engineering, General Co-Chair for 8th Intern. Conference on Open Source Systems, and Co-Chair of 3rd Games and Software Engineering Workshop at the 35th Intern. Conf. Software Engineering.