Agile Game Development

Brandon Koepke, Brett Pelletier, David Adair, Rajvir Jhawar, Ian Macaulay, and Tom Bielecki

Abstract—There are many facets of game development that make it uniquely challenging. Video games tend to have a much larger emphasis on performance than traditional applications and have much more subjective system requirements. Video game design is also more difficult than traditional software system design due to the complex interactions between objects. Since a large amount of the object interaction in video games happens in the user interface, there are also unique challenges in testing video games. These constraints make video game framework selection much more critical to the success of the game. The creative design process and the ability of the game to immerse its user is also very critical to the success of the game. In this paper we further discuss the unique challenges that game design poses to the software development community.

Index Terms—game, development, software.

I. Introduction

The video game development industry is in a unique position within the software industry due to the numerous requirements and demands that are required to produce successful, profitable video games. The video game industry has been shifting from the legacy waterfall development methodology to more agile methods such as Scrum which allow the demands of performance, expansive content, quick and efficient release cycles and immersive content to be fulfilled in an efficient manner.

The video game development industry must apply unique and innovative solutions to each step of the software process due to the nature of the agile development process and the unique demands described previously. Furthermore, companies must adapt their creative processes in order to survive in the highly competitive industry.

The sudden influx of mobile technologies has amplified the need for more efficient agile techniques in game development. The game industry's development process has had to adapt to relatively short lifecyles and unique input, size, and performance constraints in order to survive. We present a subset of these adopted methods and technologies as they pertain to agile development in the video game industry.

II. Requirements

A. Background

Requirements elicitation is a very difficult part of software development. In order to understand what requirements elicitation is, we first have to further define requirement [1, p. 62]:

> (1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed

documents; (3) a documented representation of a condition or capability as in (1) or (2).

In game design, many of the users requirements are vague and subjective. For example, the requirement that a game must be *fun* and *absorbing* is not a very well understood requirement.

The exact breakdown of what requirements elicitation entails is still being debated. One source separates requirements into function and nonfunctional requirements [2]. They then further classify the nonfunctional requirements into performance/reliability, interfaces, and design constraints.

The process for requirements elicitation is decomposed into three activities [3]. The first is the elicitation of requirements from various individual sources. Then you need to insure that the needs of all users are consistent and feasible, and finally you need to validate that the requirements are an accurate reflection of the users needs.

There are many general problems with elicitation including [4]:

1) Problems of scope
   - The boundary of the system is ill-defined
   - Unnecessary design information was given
2) Problems of understanding
   - Users have incomplete understanding of their needs
   - Users have poor understanding of computer capabilities
   - Analysts have poor knowledge of problem domain
   - User and analyst speak different languages
   - Ease of omitting "obvious" information
   - Conflicting views of different users
   - Requirements are often vague and untestable, e.g. "user friendly" and "robust"
3) Problems of volatility
   - Requirements evolve over time

The understanding and volatility issues will still be present in game design, but scope issues may not be as large of a problem as they are in other software systems.

B. Requirements Documents

The requirements elicitation process usually requires a game design document, written by the game design team. The design document is usually less formal than a typical software design document [5]. This is partially due to the fact that the imposition of too much structure can be detrimental to the creative process. The game design document is usually used as the requirements document as defined by the preproduction team [5].

A game design document should describe several things [6]
- *Define the game:* Articulate what the game is as clearly as possible

- *Core Gameplay:* Describe the main view, player activity, the user interface
- *Contextual Gameplay:* Shell menus, tutorial mechanics, multiplayer mechanics
- *Talk Story:* Describes the world backstory, character backgrounds, the levels, missions
- *Game Assets:* Design of the 3D models, voices, motion capture, sound effects, music

The requirements documents are usually written along side the design documents. Requirements can take several forms including [6] frame rate requirements, the size of the resulting binary, the length of time it should take to produce the product, having few defects, being able to process 100,000 polygon characters, etc. Once as many of the requirements as possible have been identified, the requirements are usually modelled using UML and use case diagrams as in standard software engineering. Prototyping is also very common, (Figure 1) shows a mockup in early production, and (Figure 2) shows the resulting screen.

Fig. 1.   Mockup early in production [6]

Fig. 2.   Production screen [6]

C. Emotional Requirements

One of the most difficult parts of the requirements elicitation process is the specification of the emotional requirements. Video game designers are concerned with the players experience which is rather difficult to define. One definition describes the player experience as "the means by which the player's consciousness is cognitively engaged while simultaneously inducing emotional responses" [7]. The emotional requirements can then be further broken down into the emotional intent of the designer and the means by which the designer expects to induce the target emotional state [7]. This makes the nonfunctional requirements much more paramount to the success of the software system. In order to successfully capture the emotional requirement, it must capture "the intent of the designer, and the means by which the designer expects to induce the target emotional state" [7].

There are several different methods for representing emotional requirements. The *emotional terrain* method captures artistic expression with spatial qualifiers. A range of colors are used in order to represent emotion at a specific point in time in a game. A newer method requires creating an *emotion intensity map* (Figure 3).
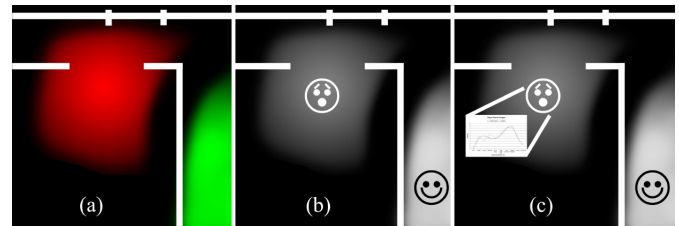

Fig. 3.   Emotion Intensity Map [7]

The emotion intensity map uses luminance to quantify intensity, and emotions are mapped by either a Chernoff Face [8], or Ekman's Facial Action Coding System [9].

Temporal qualifiers are also unique to game design. For example, "if a player reaches a room within 5 minutes of the start of gameplay, the player is very skilled so increase the intensity" [7]. Focus groups and post-gameplay interviews can also be used to gauge the emotional response of the users during gameplay. This can be used to ensure that the desired emotion is induced in the user at the correct time in a game.

There are also several software requirements that are never captured in an official manner [10]. In practice, usability, immersion, and motivation are usually considered in user surveys.

III. DESIGN

A. Background

As with most software industries, when the paradigm of programming of software designs shifted from imperative to object-oriented programming, the game development industry adopted this change to take advantage of the many benefits that it offered. These benefits included data encapsulation, easier mapping of concepts to programming structures, and the ability to group and share behavior amongst objects of

the same functional type. These benefits allowed additional functionality to be inserted without the need to duplicate a lot of code, thus greatly reducing the time, effort, and cost of developing large-scale software products.

```
CEntity
■CMoveable
    ■CVehicle
        ■CCar
        ■CTank
        ■CJetPack
    ■CHuman
        ■CPedestrian
        ■CPlayer
        ■CAlien
    ■CRigid
        ■CRock
        ■CGrenade
    ■CMissile
    ■CGun
    ■CMedkit
```
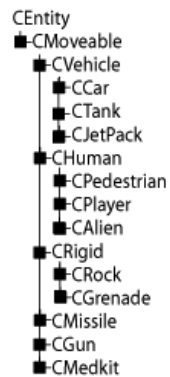
Fig. 4.   A typical game object hierarchy. [11]

In a typical object-oriented software product, the objects form a hierarchical tree [11] that has width (many objects with the same direct ancestor, such as there are many weapon-types that are each modeled as an object, and they would all share the same ancestor of 'Weapon'), and height (set of objects that form a list of direct ancestors for each other, such as how a 'DwarfMale' is a 'Dwarf' that is a 'PlayableCharacter' that is a 'Character' that is an 'Entity', and so on). This tree-like structure describes the term inheritance, where one object such as 'DwarfMale' may inherit functionality from 'Dwarf'. In small-to-medium sized products, inheritance can make development both simpler and much more efficient. However, as projects grow and changes are made, designing objects around inheritance can lead to problems due to relationships between objects in the tree. Changing or adding code at the root of the tree may cause the changes to cascade down through the rest of the tree, resulting in undesired behavior, as described by Herb Marslas, a developer for Age of Empires II:

> . . . functionality can be added or changed in a single place in the code to affect many different game units. One such change inadvertently added line-of-sight checking for trees [12].

With the development archetype shifting from the waterfall method, where projects are defined at the beginning and are less adaptable to changes, to a more agile and iterative-based approach, the ability to change the project on both small and large scales becomes a requirement of the design of the project. As teams in the game development industry began to conduct agile development processes, they began to realize that the object-oriented design paradigm was not as flexible and adaptable as it needed to be, largely in part due to the inheritance structure and requirement of defined interfaces between objects.

To solve the issues with large inheritance trees, architects, designers, and programmers discovered [13] the benefits of composition over inheritance, where it is better and/or easier to consider a has-a relationship over an is-a relationship between objects. An example of this can be described as the difference between a 'Character' object having a position in the world, rather than a 'Character' object being a 'PositionalObject'. The distinction is small conceptually, but rather large in terms of design and implementation of the project. In order to change the 'Character' to not having a position, a developer would simply have to remove the position with respect to the composition method, while the inheritance method would require the object to be placed somewhere else in the tree, potentially leading to side effects and/or undesired behavior.

The realization of the benefits of composition over inheritance began a movement of the game development industry to a design pattern known as 'Entity-Component Systems', where objects are modelled by what they are composed of or contain, rather than their position in the hierarchical inheritance tree.

B. Entity-Component Design

Fig. 5.   Entity component modelling in Crysis. Source: http://piemaster. net-content.s3.amazonaws.com/wp-content/uploads/crysis-components.jpg

The entity-component design paradigm is primarily focused on three primary objects in the application: entities, components, and system-managers (or component-managers). Entities are the primary conceptual objects in the game, such as 'Player', 'Target', and 'Map', among others. Components, on the other hand, are essentially atomic parameters that can be attributed to an entity. For example, a 'Player' entity may have a 'Position' component, a 'Render' component, a 'Velocity' component, a 'Physics' component and anything else that describes the behavior of a 'Player' entity. The other object type, system-managers are objects responsible for the management of entities and their components, and the coordination and synchronization of each respective type. For example, a 'RenderSystemManager' may be responsible for iterating through every entity with a 'Render' component, and rendering that entity to the display.

There are many benefits that this type of design provides. The primary benefit, however, may be the fact that this type of design is inherently agile in nature. Changing the system is simply the process of adding, removing, or editing the desired entities and/or components. For example, to create a hidden trap object, an invisible object in the game that reacts

in some way to the player, a developer may simply have to create the 'Trap' entity, provide it with a 'Position' component, and a 'Script' component that provides some behavioral functionality based on the 'Position' of the entity relative to the player, and register the entity with the correct system-manager that can update the 'Script' component whenever the player moves. Or to make the player invisible, a developer could simply remove the 'Render' component from the 'Player' entity. Additionally, once a component has been created, it is possible to reuse the component across the rest of the system or in future projects as it is simply a set of parameters containing no side-effects or requiring any interfaces for any other objects.
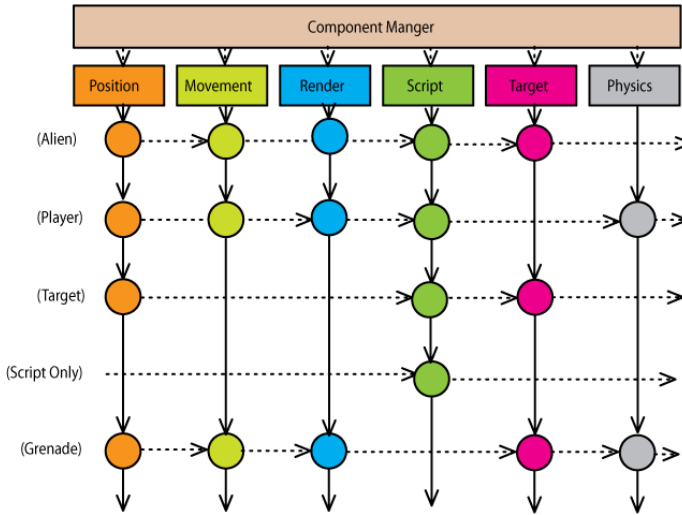

Fig. 6.   Entity-component relationship. [14]

Looking at this type of design, it is clear that the system and all of its entities are data-driven in nature. Rather than an object inheriting from some 'Renderable' object, the desire to render the object is encapsulated into whether or not the entity has a 'Render' component. With this in mind, the complimentary relationship between agile development and entity-component design actually increases due to the fact that rather than having to receive requirements from a customer (the game designer, in respect to the game developer), the developer may actually allow the customer themselves to implement the desired functionality through a scripting engine, such as Dungeon Seige's scripting language Skrit.

Incorporating scripting languages allows the game designers–that is, the department whose job it is to define the functionality of the game and its entities–to actually define the functionality of an entity. Not only does this align perfectly with the agile manifesto, where interaction with the customer is paramount, but it also (if implemented correctly) allows different entities and components to be injected, removed, or edited at run-time to provide instant feedback to both the developer and the designer. This allows for an efficient iterative approach to be taken when developing the game as a whole.

The combination of entity-component systems with built-in

```
[t:chicken_red,n:0x837FD928]
{
    [placement]
    {
      p position = 1.3,0,1.8,0x1738FFDB;
      q orientation = 0.3828,0.2384,-0.7772,0.98;
    }
    [common]
    {
        screen_name = "Super Chicken";
    }
    [body]
    {
        avg_move_velocity = 18.000000;
    }
}
```

Fig. 7.   The Skrit scripting language for describing components in Dungeon Seige II [15].

scripting engines has become the primary design paradigm of modern game development, as showcased by the primary free, open source, and production game engines in the market today such as the Unreal Development Kit with Unrealscript scripting, and Unity3D with Mono, C#, and Javascript scripting.

C. Future of Game Design

With the increased practice of agile development methodologies, and the realization of its complimentary nature with entity-component design with embedded scripting, it is likely that entity-component systems will continue to be the primary design archetype for the foreseeable future. So where does the industry go from here? A good modification candidate to the current methodology appears to be incorporating functional programming considerations within the game design. Functional programming is a programming method that strives for pure, non-state mathematical functions. Simply speaking, a pure function is a function that receives some input, performs some calculations, and returns a result or a modified copy of the input. A pure function is never able to access any state outside of the function's local scope. This is in stark contrast to the current paradigm of imperative programming, where typically it is the state of the application or an object that determines how the application behaves, with the goal of most functions to put the object or application into some known state.

Functional programming was originally introduced with LISP in the 1950s, but was slowly overshadowed by imperative programming due to the lack of computational resources available to facilitate pure functional programming in performance-critical applications. However, as innovations in the number of transistors able to be packed onto a single processing chip have begun to slow, processors have begun to incorporate additional parallel cores to increase processing power capabilities. With this plateau in processing power per core, and an increasing number of available cores, it is necessary for software to take advantage of parallel, multithreaded computing. However, the current method of imperative programming where application behavior is primarily driven by objects modifying state begins to break down in parallel environments as you begin to require transaction-based state modifications which can impose performance-limiting overhead. Thus, with parallel architectures becoming much more common, a paradigm-shift

back to functional programming begins to provide tantalizing benefits.

While functional-versus-imperative programming is primarily an implementation-level decision, understanding the advantages and limitations of functional programming at a design-level may provide a better design in which to implement the specific programming archetype. Additionally, understanding that pure-functions do not interact with program state and do not produce any side effects allows functionality to be created iteratively in an agile approach. It is simply a matter of replacing a pure-function with a new implementation–say a better algorithm or different behavior altogether–and the rest of the code (in terms of program state) is unaffected by unsuspected side effects. With the nature of the code being simpler due to the reduced complexity of program state, the application becomes much easier to maintain and extend.

IV. TESTING

The video game market is a multibillion dollar business. Game developers spend thousands of dollars into game testing to have a product that is designed and tested well enough it doesn't frustrate the players who buy the game. They do this so that the developers can successfully penetrate the market and get your name known. Testing in a video game is quite different from testing a conventional piece of software. In the game development the software is more visual then most other type of software, and therefore small errors in the code can be seen easily by the user. As a developer you want the user to get immersed into your game and have the most enjoyable experience possible. This can easily be ruined by small bugs in the software or hardware. Described below are some of the most important tests conducted throughout the design cycle of the product. Functionality Testing: The most common type of testing and as such can be done by a person with little or no technical knowledge. This is released to individuals to do the testing for the publisher. These test check for general problem in the game or the user interface. The main problems these testers looks for, is any issues with the stability, game mechanic, and integrity of the game. This is used when the game is playable in some form before being release.

A. Compliance testing

This is the general standards for each platform, such as x-box or Wii, which a game would like to be released on. Such standards include handling of error messages, copyright material, trademarks, and save data handling. If the game does not meet the standard guidelines as laid out by the platform developer then game will be delayed and cost will increase as a result.

B. Compatibility testing

This tests the game to see if its compatible with the software and hardware supplied by the publishers. Compatibility testing is used early in the beta version and at the end of the production of the game. This ensures that the hardware includes brands of different manufactures and assorted input such as joysticks. They also test the performance and results when the hardware is ran at the advertised minimum requirements.

C. Multiplayer testing

Used when testing a game with some type of multiplayer feature. Testing is done to ensure all the connectivity methods, such as modem, LAN, and internet, are working efficiently and correctly. Online player interaction and proper game performance while online contributes to the majority of this type of testing.

D. Load Testing

Tests the limits of a system, on the number of threads that can run at a particular time in the program. This will test the number of players on a MMO server, or the number of sprites active on the screen. This, being one of the harder testing types, requires a large group of testers or software that emulates heavy activity.

E. Localization testing

Deals with checking proper translation of in-game text or dialogue. These testers are usually person's who speak multiple languages and understand regional differences between countries.

F. Soak testing

Tests the how the game is running on the system for prolong periods of times in different modes of the operation or when repetitive actions. These modes of operation include, pausing the game in various modes of play, remaining on the title screen, or just idling. Usually since this type of testing requires no interaction besides the initial set-up automated tools are used to do any repetitive tasks such as mouse-clicking or button pressing. Used to find any memory leaks or rounding errors.

G. Beta testing

Initial versions of the software are released to the general public or select individuals so that they can physically play the software and find any bugs the testers have missed. This increases the chances of finding bugs in the game itself and is a cheap and efficient way to create hype for the game. Since the game is in beta the players understand that there will be bugs in the game.

H. Regression testing

Once a bug has been found and potentially fixed QA testers must verify that the bug has actually fixed the problem. They must also determine if the fix had a detrimental effect on the game or caused a new bug to suffer.

V. CREATIVE INTERACTION

Video games are an interactive form of entertainment media, so it follows that the interactivity is crucial to the success of the game. Different genres of games create a certain expectation from users about the characteristics of the interactivity. Failing to meet these expectations can frustrate users greatly, or even prevent them from being able to play [16]. The interactive aspect of the medium, often referred to as gameplay, is composed of user control and feedback.

A. Control

The extent of the user control capabilities are generally limited by the platform and accessories available to the users of your target demographic. Specific control devices and sensors include keyboard, mouse, joysticks, gamepads, touch screens, accelerometers, gyroscopes, cameras, depth cameras, floor mats and weight sensors, GPS, and microphones. Further work is being done to use bio sensors and even neurological control.

The last decade has introduced a number of large changes in the way that users interact with games. An explosion of capacitive multi-touch screens on mobile devices, as well as entirely new form factors for gaming consoles, controllers, and accessories has vastly expanded the domain of interaction design. Feedback has also progressed rapidly, with high definition 3D screens, haptic feedback mechanisms, cinematic-quality 3D sound design, and virtual-reality goggles.

B. Feedback

Without feedback, control capabilities would be useless. The design of feedback in games is something that not many gamers realize until it goes horribly awry. One such feedback flaw is the concept of lag, which is introduced in concurrent multiplayer Client-Server and adhoc games, where latency causes temporal artifacts and even race conditions in code between competing threads and processes. Lag is often an issue in Real-Time Strategy (RTS) and First Person Shooter (FPS) games. These type of games typically use multiple strategies to mitigate the risks associated with lag. Certain anti-lag techniques include kicking players off a server if their ping time is too high, as well as checking for character hits on the gamers local machine.

Feedback mechanisms need to be seamless and intuitive to the user, so that interactions with the user control devices feel directly connected to the output of the games. User feedback is also important in terms of progress and failure to complete the objectives of the game. Designers for FPS games such as Battlefield and Call of Duty have attempted to display the loss of health of a player by overlaying different effects in the field of view of the player, such as beating heartbeats, blurred vision, and dusty effects. Success and progress is often rewarded with sound clips designed to make the user feel good, as well as special effects with the same value. Games like Guitar Hero use a computer-generated audience that cheers for the player when they have done well or after completing a level/song.

VI. FRAMEWORKS

A. Background

In the game development industry today, game engines provide the necessary frameworks, APIs, and SDKs for developers to build games for various platforms. The purpose of a game engine is to abstract out the details of commonly used tasks and separate them into reusable components for the developers. Some of the functionality provided by game engine's include rendering, physics, user input, collision detection, and animation. As there are hundreds of different game engines available on the market, functionalities provided by each game engine will differ depending on the type of engine the developers choose to use. There are both commercial and open source game engines but the majority require a commercial license but may include a free non-commercial version with some limitations.

B. Types of Game Engines

There are various types of game engines available to developers. At the lowest level, developers will combine many different APIs to develop their own game engines. While this gives developers the flexibility of choosing their own components, it usually comes with a drawback of high costs. There can also be technical issues that can arise when trying to use many different libraries together for the first time. The drawbacks associated with this approach make it less attractive in the gaming industry.

The mid-level game engines include many features out of the box. These game engines are usually ready for rendering, physics, user interface, collision detection, and much more with less setup required than that of the low level approach. Developers will still need to do a bit of programming to have a complete game up and running. The Unreal Engine 3 is an example of a mid-level engine.

At the highest level are game engines that allow for point and click game development with very little programming required. Unity3D and GameMaker are examples of this type of game engine which makes the development process as user friendly as possible. Although these are easy to use engines, they come with a drawback. Most of them are limiting in the types of games they can make or the quality of graphics they can produce. The advantage of quickly developing games have made high level engines more common today.

C. Unreal Engine 3

Unreal Engine 3 is a cross platform game development toolkit that was developed by Epic Games. This toolkit is widely used by many game developers including BioWare (EA), 2K Games, Epic Games (Microsoft), Arkane Studios (Bethesda Softworks), and many more. It has been used to produce hit games including Gears of War, Borderlands, and BioShock. Epic Games describes the Unreal Engine 3 to have been designed with

> ...ease of content creation and programming in mind, with the goal of putting as much power as possible in the hands of artists and designers to develop assets in a visual environment with minimal programmer assistance, as well as giving programmers a highly modular, scalable and extensible framework for building, testing, and shipping games in a wide range of genres. [17]

Features include tools for animation, artificial intelligence, audio, physics, rendering, networking, and more. Although Unreal Engine 3 is a commercial-use game engine, Epic Games released the Unreal Development Kit (UDK) which includes many of the features of the Unreal Engine 3 for free non-commercial use.

Fig. 8.   Rendering produced using the Unreal Engine 3. Source: http://www.unrealengine.com/files/features/rendering1.JPG

D. CryENGINE 3

CryENGINE 3 was developed by Crytek and is used to develop games for Xbox 360, Playstation 3, Windows, and Wii U. CryENGINE 3 was used to develop the first person shooter franchise titled Crysis developed by Crytek and publish by Electronic Arts. Similar to Unreal Engine 3, CryENGINE 3 is a commercial game engine that offers a free version of the toolkit for non-commercial use only. It also includes a large number of tools for visuals, characters, AI systems, physics, performance, and more.

Fig. 9.   Crysis 3 was develop by Crytek using CryENGINE 3. Source: http://www.crysis.com/us/screenshots/crysis-3-concept-art-warehouse

E. Unity3D

Unity3D is a cross platform game engine that can be used for developing games for Xbox 360, Playstation 3, Windows, iOS, Android, Mac, and more. Unlike CryENGINE 3 and Unreal Engine 3, Unity has a free version that can be used for developing games for commercial use (with certain conditions). The Pro version of Unity3D has a licensing fee but includes a larger set of features than that of the free version. The developers of Unity call it a

> game development ecosystem: a powerful rendering engine fully integrated with a complete set of intuitive tools and rapid workflows to create interactive 3D content; easy multi-platform publishing; thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing Community. [18]

Although Unity has not been used to produced big name titles like Unreal Engine 3 has, it is still a popular choice especially for mobile application development. Of the featured games provided on the Unity website, a majority of the titles were developed for iOS and Android devices.

Fig. 10.   "Escape Plan" Developed using Unity3D for the Playstation Vita. Source: http://en.wikipedia.org/wiki/File:Escape-plan-screenshot.jpg

F. Cocos2D

Cocos2D is an open source 2D game framework released under the MIT License making it free to use and develop for commercial purposes. It has been ported to many different devices including iOS, Android, Blackberry 10, and Windows. There have been thousands of titles produced for mobile devices using Cocos2D. Unlike Unreal Engine 3, CryENGINE 3, and Unity3D, Cocos2D is not considered a game engine. Its goal is to abstract the details of the low level graphics packages and make game development fast and easy to use for mobile developers.

Fig. 11.   "Zombie Smash" Developed for iOS and Android using Cocos2D. Source: http://www.blogcdn.com/www.tuaw.com/media/2010/03/zombie-smash.jpg

VII. Conclusion

Video game development presents several unique challenges. The requirements documents have to take the intent of the designer into account, and many of the requirements

are very difficult to quantify. Emotional requirements are one unique challenge, and effectively quantifying them is a current research area.

The game industry has already moved from hierarchical tree based inheritance to composition for larger projects. This lead to the entity-component design paradigm which focuses on entities, components, and system-managers. High performance constraints and the move to multicore systems is currently pushing the game industry towards functional programming as well. By adopting functional programming, video games will be able to take full advantage of multicore systems.

Developers want to make sure that the users of their games become immersed in them, and very small bugs can break that immersion. This means that video games must have very few bugs. In order to reduce the number of bugs, video games will typically have functionality, compliance, compatibility, multiplayer, localization, soak, regression and beta testing.

Video games also have unique user control devices including the keyboard, mouse, joysticks, gamepads, touchscreens, etc. Feedback mechanisms in video games are also very important, and "lag" in video games can completely destroy the user experience.

Finally, there are several different frameworks that developers can choose from when developing video games. Some larger companies make their own, whereas smaller companies usually choose from those that have already been built. Each of the frameworks have pros and cons and developers must choose those frameworks that complement the work they are doing the most. All of these challenges make video game development very difficult and distinct from typical software development.

REFERENCES

[1] A. September, "Ieee standard glossary of software engineering terminology," 1990.

[2] K. Southwell, K. James, B. Clarke, B. Andrews, C. Ashworth, M. Norris, and V. Patel, "Requirements definition and design," *The STARTS Guide,*, vol. 1, pp. 177–313, 1987.

[3] W. E. Rzepka, "A requirements engineering testbed: Concept, status and first results," in *System Sciences, 1989. Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, vol. 2. IEEE, 1989, pp. 339–347.

[4] J. McDermid, "Requirements analysis: Problems and the starts approach," in *Requirements Capture and Specification for Critical Systems, IEE Colloquium on*. IET, 1989, pp. 4–1.

[5] D. Callele, E. Neufeld, and K. Schneider, "Requirements engineering and the creative process in the video game industry," in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*. IEEE, 2005, pp. 240–250.

[6] E. Bethke, *Game development and production*. Wordware Publishing, 2003.

[7] D. Callele, E. Neufeld, and K. Schneider, "Emotional requirements in video games," in *Requirements Engineering, 14th IEEE International Conference*. IEEE, 2006, pp. 299–302.

[8] B. Flury and H. Riedwyl, "Graphical representation of multivariate data by means of asymmetrical faces," *Journal of the American Statistical Association*, vol. 76, no. 376, pp. 757–765, 1981.

[9] P. Ekman, W. V. Friesen, and J. C. Hager, "Facial action coding system," *A Human Face*, 2002.

[10] T. Bentley, L. Johnston, and K. Von Baggo, "Putting some emotion into requirements engineering," in *The Seventh Australian Workshop on Requirements Engineering*, 2002, pp. 227–244.

[11] M. West, "Evolve your hierarchy?refactoring game entities with components," *Teoksessa Game Developer Magazine, osa*, vol. 13, 2006.

[12] K. Wilson, "Game object structure." GDC, 2003.

[13] M. Haller, W. Hartmann, and J. Zauner, "A generic framework for game development," *Proceedings of the ACM SIGGRAPH and Eurographics Campfire*, 2002.

[14] M. West, "Evolve your hierarchy," January 2007, available: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy.

[15] S. Bilas, "A data-driven game object system," in *Game Developers Conference Proceedings*, 2002.

[16] E. Folmer, "Designing usable and accessible games with interaction design patterns," available: http://www.gamasutra.com/view/feature/1408/designing_usable_and_accessible_.php.

[17] U. Engine, "Game development tools for unreal engine 3," available: http://www.unrealengine.com.

[18] Unity, "Game engine, tools, and multiplatform," available: http://unity3d.com/unity/.