# Agile Game Development

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

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam in iaculis neque. Suspendisse id vestibulum metus. Quisque felis massa, commodo nec iaculis sit amet, pretium vel ligula. Maecenas vel ultricies justo. Curabitur aliquam euismod pulvinar. Donec ultrices est eu lectus feugiat mollis. Nullam tempus dictum metus sed dignissim.**

***Index Terms*—game, development, software.**

## I. Introduction

ALIQUAM bibendum gravida augue. In ut turpis tellus, sit amet egestas lectus. Morbi arcu magna, hendrerit ut auctor sed, viverra a tortor. Phasellus a dolor nec metus porta pharetra. Sed non bibendum ligula. Nam dui mi, condimentum vel molestie sit amet, cursus ac ipsum. In sodales aliquet urna sed imperdiet. [**?**].

## II. Requirements

## III. Design

### A. Background

As with most software industries, when the paradigm of programming of software designs shifted from procedural and function-oriented programming 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.

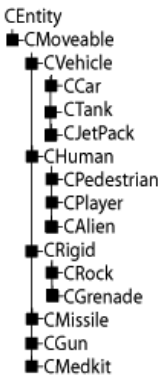


Fig. 1. A typical game object hierarchy. [1]

In a typical object-oriented software product, the objects form a hierarchical tree 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. [2]

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 [6] 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. 2. Entity component modelling in Crysis. Source: http://piemaster.net-content.s3.amazonaws.com/wp-content/uploads/crysis-components.jpg

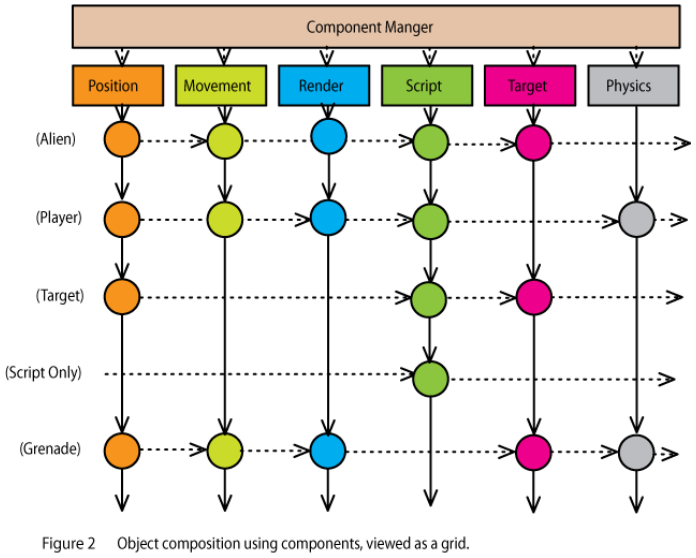


Fig. 3. Entity-component relationship. [1]

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.

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.

```
[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. 4. The Skrit scripting language for describing components in Dungeon Seige II. [3]

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 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

## V. Creative Interaction

## VI. Frameworks

### A. Background

In the game development industry today, Game Engines provide the necessary frameworks, API's, and SDK's 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.

### B. Types of Game Engines

There are various types of game engines available to developers. At the lowest level, developers will combine many different API's 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 mid-level engines.

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 the high level engines more common today.

### C. Unreal Engine 3

The 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. [8]

Features include tools for Animation, Artificial Intelligence, Audio, Physics, Rendering, Networking, and many more. Although the 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. 5. Rendering produced using the Unreal Engine 3. Source: http://www.unrealengine.com/files/features/rendering1.JPG

### D. CryENGINE 3

The 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 the 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 many more.

### 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;


Fig. 6. Crysis 3 was develop by Crytek using the CryENGINE 3. Source: http://www.crysis.com/us/screenshots/crysis-3-concept-art-warehouse

> thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing Community. [10]

Although Unity has not been used to produced big name titles like the 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. 7. "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.

## VII. Conclusion

(The conclusion is dependant on the body of our paper, which has not been completed for this draft.)

Fig. 8. ”Zombie Smash” Developed for iOS and Android using Cocos2D. Source:http://www.blogcdn.com/www.tuaw.com/media/2010/03/zombie-smash.jpg

## REFERENCES


[1] M. West. (2007). *Evolve Your Hierarchy* [Online]. Available: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

[2] K. Wilson. (2002). *Game Object Structure: Inheritance vs. Aggregation* [Online]. Available: http://gamearchitect.net/Articles/GameObjects1.html

[3] S. Bilas. (2002). *A Data-Driven Game Object System* [Online]. Available: http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf

[4] J. Carmack. (2012). *Functional Programming in C++* [Online]. Available: http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/

[5] J. Carmack. (2012). *Functional Programming in C++* [Online]. Available: http://www.altdevblogaday.com/2012/04/26/functional-programming-in-c/

[6] M. Haller, W. Hartmann, and J. Zauner. (2022). *A generic framework for game development*. *Proceedings of the ACM SIGGRAPH and Eurographics Campfire*

[7] J. Ward. (2008). *What is a Game Engine?* [Online]. Available: http://www.gamecareerguide.com/features/529/what_is_a_game_.php

[8] Game Development Tools for Unreal Engine 3, Unreal Engine. [Online]. Available: http://www.unrealengine.com/en/features/

[9] MyCryENGINE, Crytek. [Online]. Available: http://www.mycryengine.com/

[10] Game Engine, tools, and multiplatform, Unity. [Online]. Available: http://unity3d.com/unity/

[11] Cocos2D for iPhone, Cocos2D. [Online]. Available: http://www.cocos2d-iphone.org/about