ECS Game Engine Design

Daniel Hall, *Student, Cal Poly SLO,* Zoe Wood, *Advisor, Cal Poly SLO,*

Abstract—Game programming design and organization can be difficult and complicated. To simplify the development process, frameworks with an array of tools and utilities known as game engines are used. The main goal of this project is to explore game engine designs and develop a design for a modular and expandable game engine. The designs covered in this paper are Object Oriented Programing (OOP) and two Entity Component System (ECS). OOP designs, commonly used in computer science, use a hierarchy of objects to share functionality. ECS designs are based off of the concepts Composition over inheritance in which objects contain features instead of inheriting them. However, designs have their own weaknesses, such as expandability in OOP design. The issue comes from the tightly coupled nature of a hierarchy where changes made near the root of the hierarchy requires significant code reconstruction. ECS solves the coupling issue, however, problems exist with cross system communication and shared components across systems. The two ECS designs used, Cupcake and Artemis, aim to solve these issues. By writing simple game applications, the functionality of each design was tested and analysed. Using the two strengths of Cupcake and Artemis, this paper proposes a new ECS design to minimize architectural issues.

1 Introduction

Game development can be a complicated process. It can require the use of graphics, sounds, physics, networks, artificial intelligence, and input. A common practice is to use existing libraries that implement a technology to save time and effort. Even so, tying several technologies together into one system can prove to be a difficult task. Game engines answer this problem by providing both the framework and technology needed to make a game. This allows game developers to skip the struggle of implementing technologies and focusing on game development.

The largest problem encountered when developing game engines is how to represent game objects. Game objects can range from a simple 2D image with no control or interaction to a highly complex 3D object with controls, sounds, animations, and AI. Conceptually, it is easy to understand that game objects represent entities within a game with several features. The issue arises when trying to organize an architecture that can handle a combination of features.

A common approach is to use an Object Oriented Programming (OOP) architecture. However, due to the nature of inheritance and hierarchies, difficulties arise when representing a game object. Entity Component Systems (ECS) became widely popular as the answer to OOPs problems with game objects.

This paper will go over the disadvantages of OOP, benefits of ECS, variations of ECS, and a proposed ECS design to promote modularity.

2 Object Oriented Programming

Object Oriented Programming is a programing design that is meant to be reusable and modular. It represents information as objects containing data and logic. A unique and practical feature of OOP is inheritance. This allows the architecture to create a hierarchy of shared features and structure which helps promote code reuse and organization.

What would a hierarchy look like with game objects? OOP sounds exactly what game engines need for code reuse and organization. *Figure 1* is an example of what a hierarchy may look like representing a vehicle and a

- *Daniel Hall is with the Computer Engineering Department, California Polytechnic State University, San Luis Obispo, CA, 93407.*
  E-mail: daniel.ma.hall@gmail.com
- *Zoe Wood is with Computer Science Department.*

Manuscript received June 14, 2014

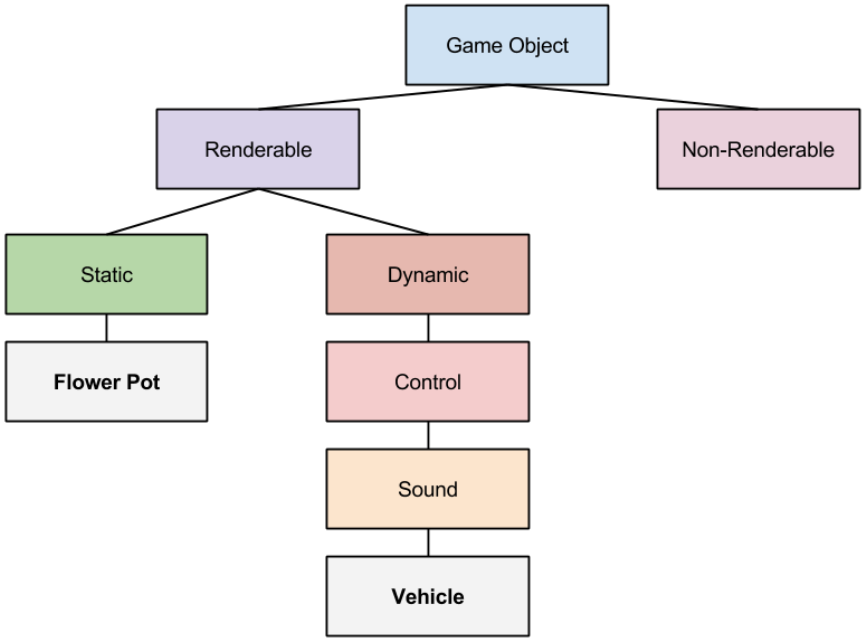

Fig. 1. OOP Hierarchy

flower pot. This works out when paths are simple such as the previous example. However, problems arise when game objects begin to share features.

2.1 Problems of OOP

When it comes to shared features in a hierarchy, designing an architecture becomes really difficult. How does *Figure 1* change with a static object with sound or perhaps a non-renderable object with sound and control such as a player look? *Figure 2* shows a potential hierarchy tree with the added game objects. Our tree begins to show redundancies on features and paths. The more combination of features added to game objects, the more complex the hierarchy will become. This causes the code to be difficult to manage and organize.

A main issue of this complex hierarchy is processing game objects. It would be difficult to create a single function that accepts the various types of game objects holding a specific functionality. Solutions such as function overloading are difficult to maintain because every type of object containing a specific functionality needs it's own function. This forces either a non intuitive hierarchy structure or code reuse for all the different combinations.

The worse issue of this complex hierarchy is expandability. Every child is heavily depended on its parents structure. Adding a new feature or changing a current feature can be difficult or impossible without extensive code reconstruction.

3 Entity Component Systems

Composition over inheritance is the proposed method to solve the hierarchy issues caused by OOP. This concept adds functionality through composition instead of inheritance making systems more modular and independent. As an example, using an ECS, the vehicle object will contain an object for features such as sound or control instead of creating a hierarchy chain. This makes the vehicle object more dynamic since altering features is as simple as adding or removing objects which require no major code reconstruction.

3.1 Architecture

ECS are composed of three fundamental parts, entities, components, and system. Entities are

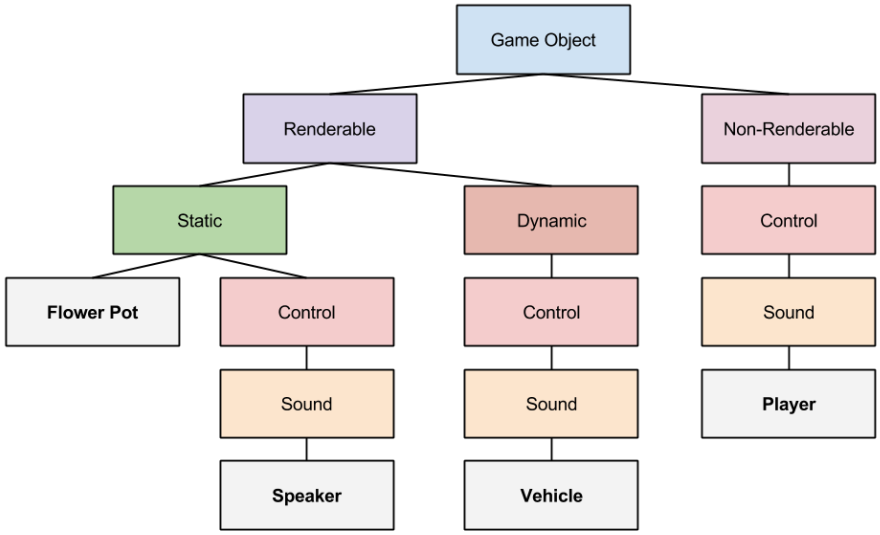

Fig. 2. OOP Hierarchy Redundancy

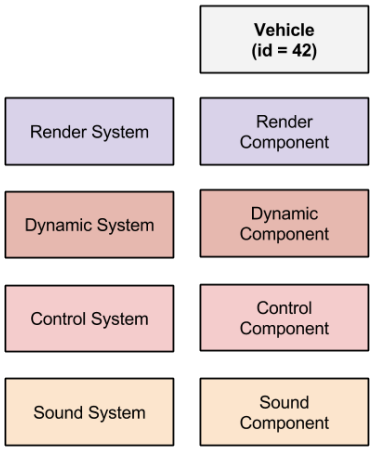

Fig. 3. Game Object in ECS

unique ids associated to objects. Components are data representing information about entities. Systems run logic on entities to utilize or alter the data from components.

In reference to game development, game objects are referenced as entities and features are represented as components and systems. This can be viewed like a database in which entities are the unique ids, systems are the database tables, and components are the data within the database tables.

Figure 3 shows how a game object may look like in an ECS architecture. Vehicle is associated to a unique id. Using this unique id, the caller attaches features by creating components associated to the desired feature. This allows systems to process entities and apply actions using the provided data in the components.

3.2 Benefits

The benefits of an ECS architecture is its isolated nature. Each component is nothing more than simple data points, therefore inherently being isolated from other components. Systems require specific components in order to run its logic. This makes systems relatively decoupled since the number of different components contained in an entity does not alter how the system interprets the entity. As long as the component requirements are met, the system will run its logic on an entity.

Figure 4 shows the previous hierarchical design converted to an ECS. In respect to the Render System, what difference does the flower pot and speaker have? Does the Render System care if the speaker has a sound component? Is the sound component required to render an object to the screen? This is an example of how the ECS system promotes isolated features and strong code organization. Instead of following a long complicated tree, a simple table can be used to view what features a game object contains.

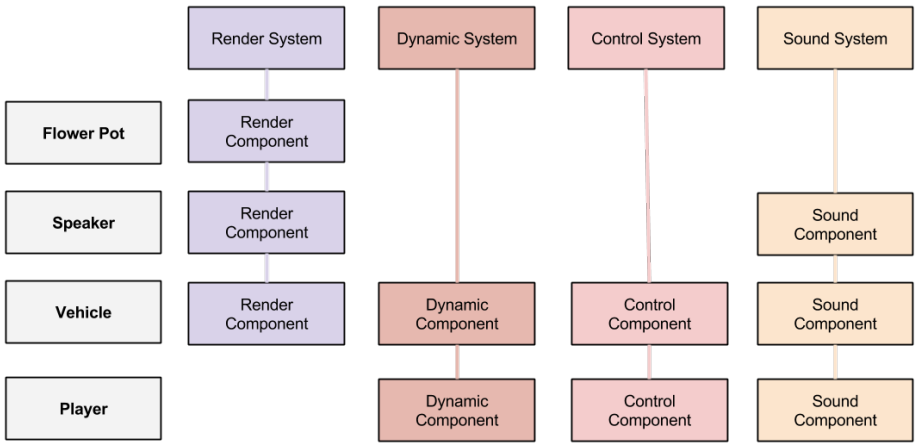

Fig. 4. Entity Component System

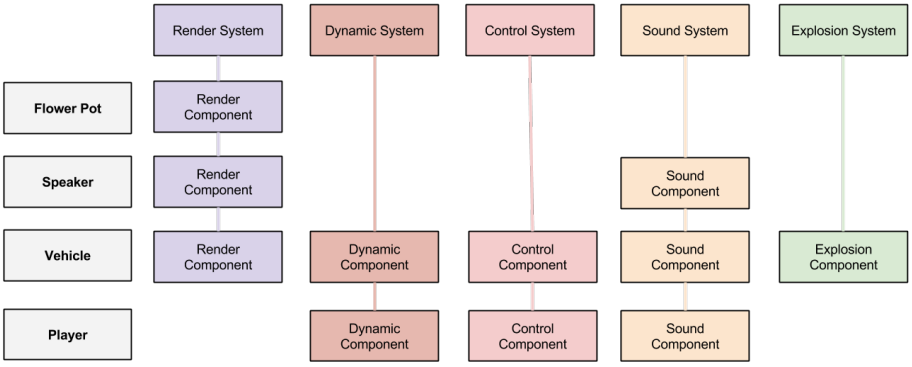

Fig. 5. Entity Component System Added Functionality

An additional benefit of an ECS architecture is the modularity and expandability of systems. Since systems and components are relatively isolated and independent, it easily allows new features to be added. For example, adding sound to a flower pot is as simple as adding a sound component. Adding a new feature such as explosions can be as minimal as creating a new system and component. An updated diagram on *Figure 5* shows how new new features have low impact on the overall structure of the engine.

3.3 Disadvantages

Although ECS systems may address several problems brought by OOP, they have their own set of problems. Isolation and independence are both the positive and negative aspects of an ECS system. The negative aspect occurs when the design requires shared components and cross system communication.

The issue with shared components is containment and process order. The concept promotes coupled systems which goes against the independent design of ECS. Depending on the design, the location of shared components is vague. Additionally, the order of processing entities can play a vital role. For example, in an engine with movement, collision, and render, the process order will alter the visible outcome. Such as, an order of movement, render, and collision could create a case where an object would draw clipped into a wall because colli-

sion was not processed yet.

Cross system communication is another aspect of ECS that is difficult. Cross system communication is necessary for cases in which an event happens. For example, a ball colliding with the ground should make a noise. The issue with an ECS is the collision and sound system are different and isolated from each other. A couple methods of solving this isolated issue is adding states to components or implementing a messaging system. Either way, designing a method to connect systems while keeping them independent is difficult.

4 CUPCAKE ECS

The Cupcake ECS architecture is designed to be a modular ECS system in which the goal is to easily add or remove systems without affecting the architecture. This requires systems to be completely independent from each other and never use or have reference of any other system. This ideally will create an engine where a user can plug and chug new features and hopefully promote code reuse and sharing among users.

An example of how a user would use Cupcake is to start with the base ECS framework. This would consist of utilities to manage systems, entities, and a framework to make new systems and components. Next, the user would choose from a list of plugins (systems) for any required functionality. For example, a user can pick a list such as, FMOD for sound, OpenGL for 3D rendering, and Bullet for physics. Lastly, an install script would compile the plugins and framework into a static library and set of header files. The result would be a highly customizable, user specific engines.

4.1 Architecture

Cupcake architecture is split into several parts, engine, managers, and systems. *Figure 6* shows the basic architecture of Cupcake. The engine is the top layer interface to all the systems, managers, and entities. It is responsible for running the game loop and managing systems and entities. The engine manages entities by providing the ability to add and remove entities while maintaining a unique id for each entity. Systems are managed by providing two list for processing and idle systems. Systems contained in processing are processed by the engine every game loop.

Systems are responsible for managing components and implementing several abstract methods. The abstract methods consist of init, release, and update (for processing systems). An example of a system, such as FMOD, to play sounds is a system with a list of components for sounds and position. The init function would initialize the FMOD library file. The release function would release all resources tied into the system such as components and the FMOD library. The update function is responsible for updating the sound position for moving objects.

4.2 Cupcake Solutions

Cupcake contains the same generic issues with any ECS systems with shared components and cross system communication. To solve the shared components issue, Cupcake has an external set of components outside the engine. These external components are passed to Systems upon creation. This allows multiple systems to have the same set of components without requiring any method to keep the data in sync. Additionally, it maintains the independence of systems from each other. An example of sharing a component is position for the rendering and physics system. By referencing the same list of components, any changes made to position by physics will automatically update for the renderer.

The cross communication of systems is solved with an external messaging system. This messaging system consist of handlers, triggers, and messages. Triggers is a piece of logic executed each loop for specific conditions. Based on the conditions, a message would be sent to the messaging system. Message handlers is an object created for systems to catch and execute logic based on messages. An example of a player moving requires three parts. A trigger to execute based upon player input. When the trigger registered a player input, it would send the appropriate move message into the system such as move forward. The forward message

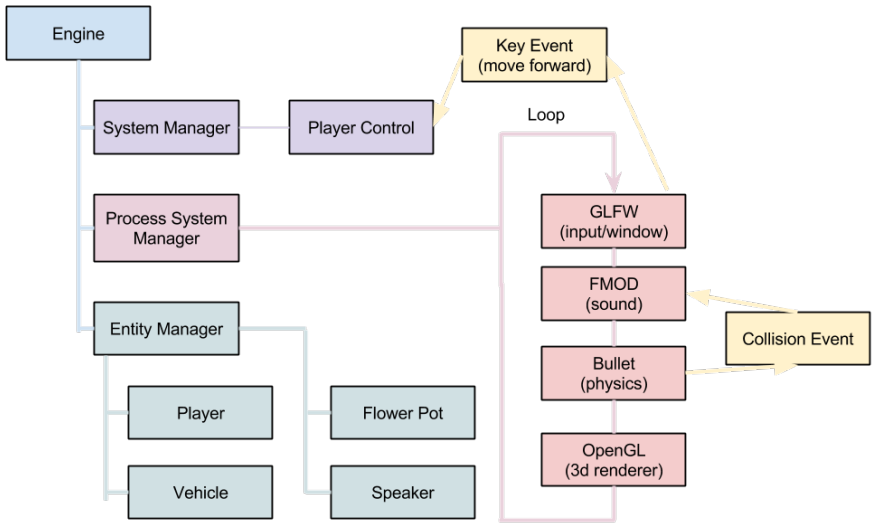

Fig. 6. Cupcake Architecture

is then caught by a handler for the physics system. The physics system would then be called to move the player by the handler.

4.3 Analysis

Cupcake system maintained the general essence of an entity component system by promoting system independence and data driven architecture. The approaches to the common ECS problems is sufficient, however could be improved upon.

Shared components is a relatively simple solution to the shared data problem. The issue with external shared components is the vague representation of what contains the shared component. It is not clear where the caller should make or get the shared component unless instructed. By requiring the caller to be instructed where this external list should come from exposes a weak design. It allows callers to make duplicate lists of the same component which would negate the purpose of having shared components.

The external messaging system proved to work relatively well. The external aspect of the messaging system helps promote system independence and allows the caller to make customized behavior for messages. The problem with the external messaging system is its weak integration into the engine. It would be much more useful to create a standard on how cross system communication should be handled. By integrating messaging into the engine, systems would be more likely to be designed around a standard messaging system.

Lastly, a flaw with Cupcake is removing entities from the engine. Due to components being stored within systems themselves, there is no simple method of removing entities from the systems. The caller would have to release components from each system manually in order to delete the entity. This leaves issues with allowing systems to be designed without implementing a function to release data on a specific entity.

5 Artemis ECS Framework

Artemis ECS Framework created by Gamadu is designed for game development. The design is relatively similar as Cupcake. Entities are unique ids, components are data, and systems process logic using the data. Artemis Framework is originally written in Java, however this analysis will go over the Vinova Mobile C++ port of the Artemis Framework. From the Java to C++ port, several features were not implemented so this analysis is only accurate to the C++ port.

5.1 Architecture

The Artemis Framework contains a central World object which is the interface of the

framework. The World object contains all managers and runs the game loop. The managers used in this framework are for systems, entities, groups and tags. The managers for systems and entities are similar to Cupcake in which they manage what entities or systems exist. The group and tag manager are simple systems used to organize entities together and add a tag to a specific entity. This is done by associated some string to a group of entities or a single entity.

The entity manager in Artemis is unique because it stores all components in the engine. This is done by a two dimensional array in which the first identifier is the component type and the second identifier is the entity id. The entity manager also manages unique ids for each active entity.

The systems in Artemis manages a list of entities to process. This is done by assigning an identifier for each component type. Using the component type, entities can be filtered out by the system so only valid entities are processed. To access data, the caller must create component mappers to get components from the entity manager.

5.2 Analysis

Artemis has a unique solution to fix the shared components issue. By attaching components to the entity manager, this allows all systems to have access to components. This allows components to automatically syncs across systems using them. Additionally, it removes the ambiguous effect of whom contains shared components since all components are stored in the same location. This is useful when the caller needs to remove an entity because the entity manager can search through the list of components and remove all data associated to the entity.

An issue with Artemis is the lack of cross system communication. The only thing shared between systems are components which could be used as a message. However, this can potentially be an expensive method of messaging since creating and deleting components is expensive.

Lastly, the largest limitation of Artemis is the component and system type limit. The unique ids associated to components and systems are done with bits. The bits are limited to a count of thirty two. This create an issue where filtering becomes broken when the user exceeds the component or system limit. The new components are assigned undefined type that can allow invalid entities to be passed to systems.

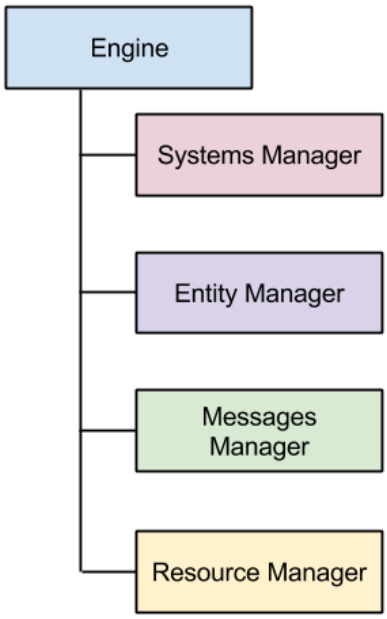

Fig. 7. Engine Object

6 Proposed Architecture

The proposed architecture for an ECS system is a combination of Cupcake and Artemis. Cupcake solved cross system communication and maintained system independence. Artemis solved the issue of shared components by localizing all components in the entity manager. The combination of both technologies solve issues with ECS while maintaining modularity.

6.1 Architecture

This architecture like Cupcake and Artemis will have a central interfacing object called engine. The engine will be responsible for managers and the game loop. The contained managers are systems, entities, resources, and messages. The engine is also be responsible for

using the system manager to process processing systems every game loop.

Systems and entity manager will work identically as Artemis in which components are stored within the entity manager. The component types will have a unique id to allow systems to filter out component types. Using the component ids, systems can create a list of valid entities that have the proper components and process them.

A new addition is the resource manager which store information that components may share. Object meshes is an example of shared data for multiple components. For example, if fifty identical enemies exist within a game, it would be expensive to create duplicate mesh data for each entity. The resource manager would handle cases in which it is ideal to share the exact same data among entities.

The message manager will be responsible for cross system communication. Like Cupcake, it will consist of handlers that the caller creates that catches messages and executes logic into systems. Instead of triggers, systems will be responsible for sending messages to the message handler.

7 Projects Using ECS

For this project, two difference ECS systems were implemented and used in a simple game context. The Cupcake engine written in C++ and used to make the test application Cron. The Artemis Framework C++ port was used to developed an engine used for the CPE 476 project Carrota.

7.1 Cron

Developing the Cupcake engine, a test application was created in order to test the functionality of the engine. *Figure 8* shows the example test application. The sphere represent the player where the user may move the object around with controls. Additionaly, positional audio is based off of the position of the sphere. The cubes represented test objects in which physics inacted upon them and they played a constant audio. This was used to ensure 3D audio worked properly.

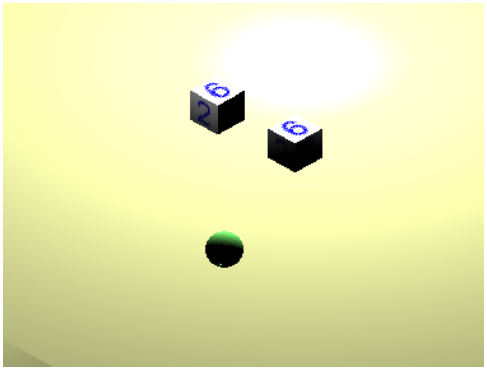

Fig. 8. Cupcake Test App

Fig. 9. Carrota Bunny

Fig. 10. Carrota Shop Menu

7.2 Carrota

Using the Artemis Framework, an ECS Engine was developed for a CPE 476 project. With this engine, an FPS defense game was created. The features in this engine included: 2D/3D/Text rendering, sound, collision, physics, animations, framebuffer objects, and shadows. Each feature was relatively independent from each other.

8 Future Work

In future work, the proposed ECS implementation will be developed and refined. The framework will initially be built followed by feature implementations. Features to support 3D

Fig. 11. Carrota Doughnut Gun

rendering and real time graphics are hoped to be accomplished. The end goal is to make a verbose framework with multiple features than can be shared across engines.

REFERENCES

[1] *Entity-Component-System-Revisited*, http://flohofwoe.blogspot.com/2013/07/entity-component-system-revisited.html?m=1: July 6th, 2013.
[2] Randy Gaul, *Component Based Engine Design*, http://www.randygaul.net/2013/05/20/component-based-engine-design/: May 20th, 2013.
[3] Ted Brown, *Fast Entity Component System*, http://www.openprocessing.org/sketch/18023: January 18th, 2011.
[4] Bob Nystrom, *Component*, http://gameprogrammingpatterns.com/component.html
[5] Gamadu, *Artemis Entity System Framework*, http://gamadu.com/artemis/
[6] Alec Thomas, *EntityX*, https://github.com/alecthomas/entityx/tree/master/entityx: May 15th, 2014.
[7] *Entity System Wiki*, http://entity-systems.wikidot.com/