Swinburne University of Technology
COS20007 - Object-Oriented Programming

Data-Oriented, Entity-Component-Systems Within The Context of Object-Orientated Programming

JACOB MILLIGAN
100660682

November 13, 2016

Contents

1. Overview

Object-Oriented programming languages are amongst the most commonly used for developing large software applications (stackoverflow.com 2016). Specifically, within the context of video games, the use of C++ as the language of choice is ubiquitous. Game engines commonly have a high demand on resources, frequently operate an a huge variety of data, and encapsulate a large spectrum of disciplines, and often an Object-Oriented approach to design is the most obvious choice for managing complexity. However, as these applications and tools become larger and the demand they place on computer hardware increases, the focus on traditional Object-Oriented design has recently fallen under criticism, not only within the game industry but other software engineering disciplines, as a source of both inefficient run-times and difficult to understand code-bases (Llopis 2009).

In response to these criticisms, many developers within the game industry have turned to a different approach to software architecture - Entity-Component-Systems (ECS) and a 'Data-Oriented' approach to design, both of which aim to decouple program logic from the data it operates on, increase runtime efficiency by maximizing smart CPU caching behavior, and build more flexible, easier to maintain solutions for primarily video-game related, large-scale systems.

This report will explain how Object-Oriented programming is traditionally used within game engines, describe an implementation of an alternate Data-Oriented and Entity-Component based system, and analyze whether the use of such systems and design practices provide a more desirable solution than traditional Object-Oriented design or if a combination of the two approaches is most ideal in relation to run-time efficiency, flexibility, and maintainability.

2. Background

2.1. Object-Oriented Programming in Games

Within the game industry C++ is the programming language of choice (Gregory 2015, p. 4) and its Object-Oriented features are often heavily utilized for reducing code complexity and encapsulating data. Game objects may have hundreds of variations of their base representation, leading to developers employing inheritance and polymorphic designs that accurately describe the way the game world appears to the user. For example, within the context of Unreal Engine 4, all objects that can be placed within a level inherit from a child of the `AActor` class which itself is derived from the `UObject` class (*UE4 Programming Guide* 2016), allowing the programmer derive from these classes for their own types and obtain access to a large number of pre-defined behaviors. From the perspective of a user, this can often be the simplest way to express a game engine, however such an approach to design can lead to overly complex code bases as variations in game object types become wide and numerous, resulting in structures similar to Figure 1. If not managed carefully by both developers building and using the engine, such structures can become difficult to debug and reason about, performance bottle-necks hard to identify, and modifications to the structure can be challenging as these large hierarchies with many co-dependencies often prove inflexible.

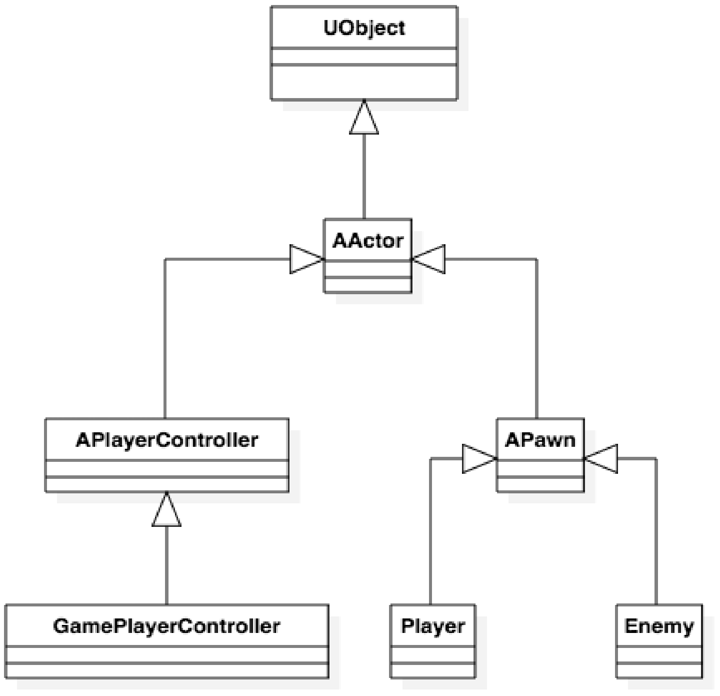


Figure 1: A possible Unreal Engine 4 class hierarchy.

It's important to note that while overly complex and coupled OOP architecture can result in the above-mentioned challenges, relationships between the level of usage of OOP and programmer productivity have been identified (López, Chavoya, and Campaña 2012) (Comstock, Jiang, and Naudé 2007), demonstrating that an appropriately decoupled and well-designed Object-Oriented architecture tends to lead to quicker development times. Moreover, this isn't to say that existing engines are poorly designed, only that creating a well-structured solution within them may often be challenging.

2.2. Data-Oriented Design

CPU Caches Typically, a CPU will employ a hierarchy of caches of differing speeds and sizes which store recently accessed data for further use in fixed-size blocks known as *cache lines*. Before retrieving some set of data from main memory, the CPU checks for its existence in ascending order in the L1 cache, the smallest and fastest cache, through to the L2 cache and so on before proceeding to fetch it from main memory. Furthermore, a CPU's L1 cache is typically split into a dCache (data) and an iCache (instructions), which allows commonly reused *instructions* to be fetched faster. The speed of each cache is highly dependent on its size and as such, the amount of memory each cache level can store is carefully balanced between specific CPU architectures - for example the highest selling current CPU chip according to both Amazon (Amazon 2016) and Newegg Business (Newegg 2016), Intel's i7-6700K, has a 256kb L1 cache, split 128kb/128kb between data and instructions, a 1024kb L2 cache, and an 8mb L3 cache. Furthermore, the way in which CPU's *predict* how data will be accessed and how they replace the currently cached data based on this prediction can affect performance. These *replacement policies* can involve replacing oldest data first, or newest data first in the case where a memory access pattern resembles an ABCDABCD sequence, or a combination of both of the two alongside other policies (Al-Zoubi, A. Milenkovic, and M. Milenkovic 2004).

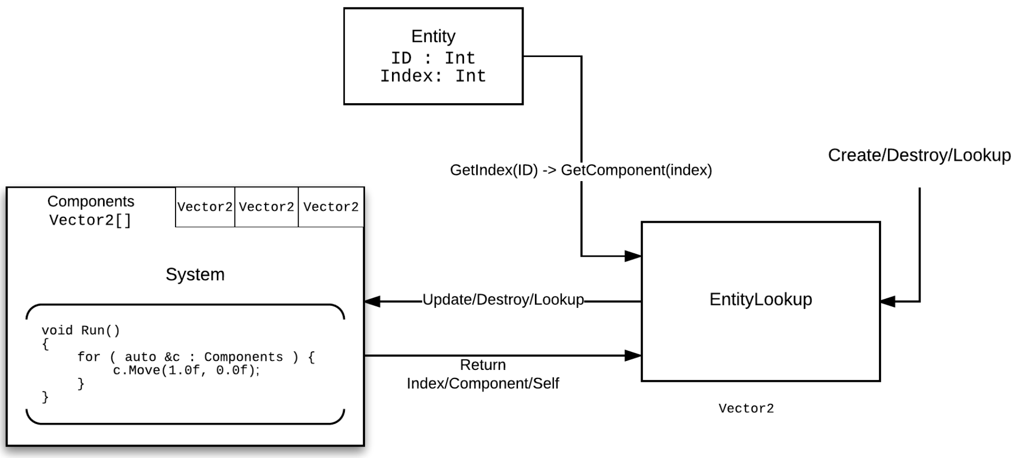
Data-Oriented Design Often referred to as a programming paradigm, Data-Oriented Design can rather be defined as a thought process to apply when designing data structures and algorithms. Primarily this design approach involves reasoning about *how data is used by a computer at the hardware level*, by arranging structures and systems so that their associated data is laid out and operated on in a way that the specific behaviors of different computer architectures are fully taken advantage of and penalties caused by factors such as branch-predictions, cache and TLB (Translation Lookaside Buffer) misses, virtual function and container lookups are minimized or eliminated completely. While appearing at first to be a case of premature optimization, games and their engines operate on frametime-limited deadlines and a Data-Oriented approach to design can lead to significant performance gains (Dice 2013). Often this approach to software design places a large emphasis on operating on data in bulk, stating that it's relatively rare that any piece of data exists as a unique permutation operated on in complete isolation (Dice 2010). This is often implemented by storing data in contiguous containers, such as arrays or packed plain-data structs, where program systems often consist of a single `for` loop with minimal branching. As game systems very rarely operate on objects individually, this approach can result in repeatedly accessed data to be present in a single CPU cache line for extended periods of time.

2.3. Entity-Component Architecture

Entity-Component architectures were initially developed in order to create flat inheritance structures in Object-Oriented languages and to facilitate highly data-*driven* (Not to be confused with Data-Oriented design) game engines and tools - that is, to allow games to be described using tools, data file formats such as JSON and XML, and databases rather than through code (Leonard 1999).

In a pure ECS, a game object (entity) is an integer ID which acts as an index into a series of arrays of **components** which store only plain data and contain no game logic. All logic within such an application is then executed through **entity systems** which iterate over these component arrays using each entities id as an index for accessing properties to manipulate. Generally the access and modification of entities, systems, and components is controlled via an *entity lookup table or map* which handles creation, destruction and association of entities and components, storing systems and handing them out as needed, **Figure 2** shows an overview of this behavior. In practice this allows entities to be modified and their behavior altered with ease as components are attached and detached as needed and provides numerous other benefits as a side-effect of this behavior, such as:

1. Reducing the number of cache misses, and therefore CPU stalls, during game play due to contiguous storage of components and entities.
2. An entirely flat inheritance structure, creating easily understood and modularized code bases able to be debugged and tested more flexibly.
3. A game able to be described primarily through data files, databases, and scripting languages able to be modified by non-programmers with ease.



```
void Run()
{
    for ( auto &c : Components ) {
        c.Move(1.0f, 0.0f);
    }
}
```

Figure 2: A high-level overview of basic ECS functionality.

3. Entity-Component-System Implementation

This section describes an implementation for a complete Entity-Component-System but should in no way be considered as the ideal approach. Many game and application developers will have wide and varied requirements and their goals will be very different to the goals specified here, some may require pure performance, while others may place more importance on readability and interface. As such, this ECS can be considered as an overview of possible implementations, a stepping-stone for further exploration into the subject by interested readers.

3.1. Requirements

In the design of the ECS, implemented using C++, three factors were of primary concern:

Run-Time efficiency As mentioned previously, performance of a program is one of the possible major gains from using an ECS. In the implementation outlined in this report, run-time efficiency is considered as *the lowest possible average time to render a frame.*

Flexibility Another possible advantage of using an ECS is the flexibility that a programs architecture gains - in fact the earliest proponents of Entity-Component style architectures weren't concerned at all with run-time speeds, just the ability for their software to be driven by data description files and tools (Bilas 2002). In the context of the implementation outlined in this report, flexibility is related to the number of classes, functions, variables, and control flow statements required to alter in order to add and remove entire application systems, essentially - how easy it is to modify a system.

Maintainability Maintainability is considered an important part of building large-scale software with long life-spans. While games tend to have short lifespans, the engines they use are generally much more mature and are often iterations of much older engines with new names, sometimes having decades worth of technical history and refactoring, such as in the case of id Software's id-Tech (Griliopoulos 2011). In order to mitigate the impact of these issues, a core system should aim to maximize maintainability from the point of design. In regards to the ECS implementation high maintainability is considered as isolation of logic and state, the speed at which one can step through call graphs and class trees to discover bugs, and the ease at which a test can be written without needing to expose any internals to the testing suite.

3.2. Implementation

Components In order to guarantee efficient run-times and maximize cache hits, components are stored so that the sequence in which their members are accessed and modified is *contiguous* and kept as such when entities are destroyed, added, and modified.

Many implementations will often store components on the entities themselves in a form of hash map for querying an entities contents, resulting in contiguous access to the *entities* but not necessarily to the components.

While this is a much simpler solution for querying components, it can negatively impact run-time speeds when executing systems, not only due to non-contiguous storage of components and therefore cache-unfriendly design, but also due to the overhead of a hashing function and the case where entities being removed and inserted leaves fragmentation of memory resulting in lookup time penalties and more cache problems.

Therefore the solution can be to store a component as an `std::vector<T>` and query them by indexing into the vector using each components' owning entity id, effectively creating a **component pool** that allows each component to be reused amongst entities.

However, this can result in instances where, as components are reassigned, their index no longer matches their associated entities id, demonstrated in **Figure 3**.

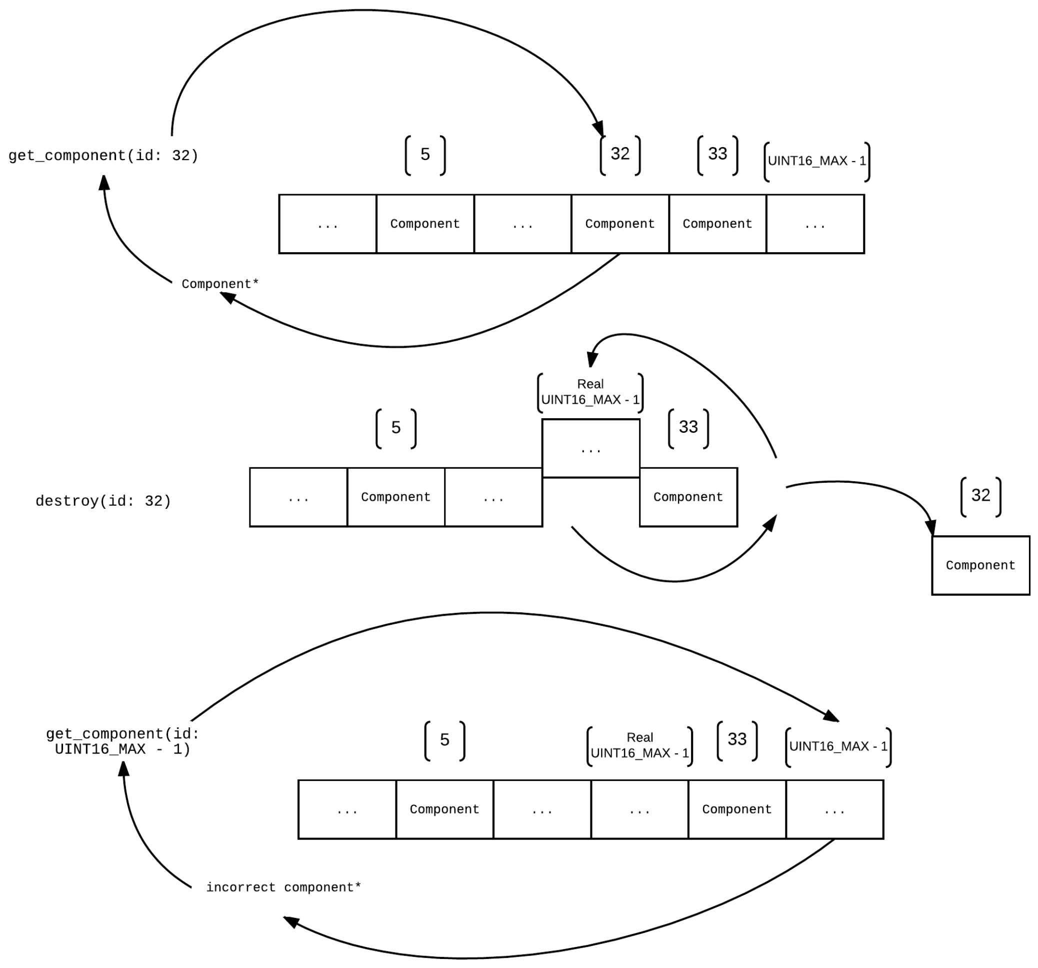


Figure 3: The problem of maintaining indexes in an ECS

In order to resolve this problem, an indirection table can be employed for querying an entity by id, implemented in the `ComponentData<T> struct`. Essentially this consists of an array of size `UINT16_MAX` (65535) containing a series of `Entity structs` that acts as a level of indirection, much like the index in the back of a book allows you to quickly find *the page* (or index) a particular topic (or piece of data) is located at. This indirection array then returns the index (or page, sticking with the book analogy) that the data (or topic) is located at within the actual `std::vector<T>` that contains the component instances. As the indirection array always contains the maximum number of possible entities, the id's will always be in their correct position, **Listing** 1 shows the code for this.

```cpp
/// ComponentData is a container of components with garunteed
 ↪ contiguous
/// storage that can be queried with an Entity id
/// \tparam Type of component to store
template <typename T>
struct ComponentData {

  /// All instances of this component mapped to an entity
  std::vector<T> instances;

  /// Initializes the container and lookup table with all entities
  ComponentData()
  {
    for ( int e = 0; e < UINT16_MAX; ++e ) {
      lookup_[e].id = e;
      lookup_[e].index = UINT16_MAX;
      lookup_[e].generation = UINT16_MAX;
    }
  }

  /// Gets an entities associated component as a pointer.
  /// Uses a pointer rather than a reference to allow for better
  /// auto compatibility and checking against nullptr
  /// \param entity The entity whose component is being retrieved
  /// \return Pointer to the component instance if found, nullptr
   ↪ otherwise
  inline T* get(const Entity& entity)
  {
    auto lookup = lookup_[entity.id];
    if ( lookup.generation != entity.generation )
      return nullptr;

    return (lookup.index < UINT16_MAX) ? &instances[lookup.index] :
     ↪ nullptr;
  }

private:
  /// Lookup array used to index into the actual data
  Entity lookup_[UINT16_MAX];
};
```

Listing 1: Using a lookup array as a level of indirection for maintaining indexes

Each element of the indirection array contains an `Entity struct` where its index is either a valid index in the actual component vector, or if the entity is not present, will have its index set to `UINT16_MAX`. Whenever an entity is added, a new component is pushed to the back of the component vector and the entity in the indirection array is updated with its new index, when an element is swapped, the two elements involved in the swap are updated in the indirection array with their new indexes and any elements being removed are assigned `UINT16_MAX` to reflect non-existence, demonstrated in **Figure 4**. The code for the implementation can be seen in **Appendix A**, **Listing 2**.

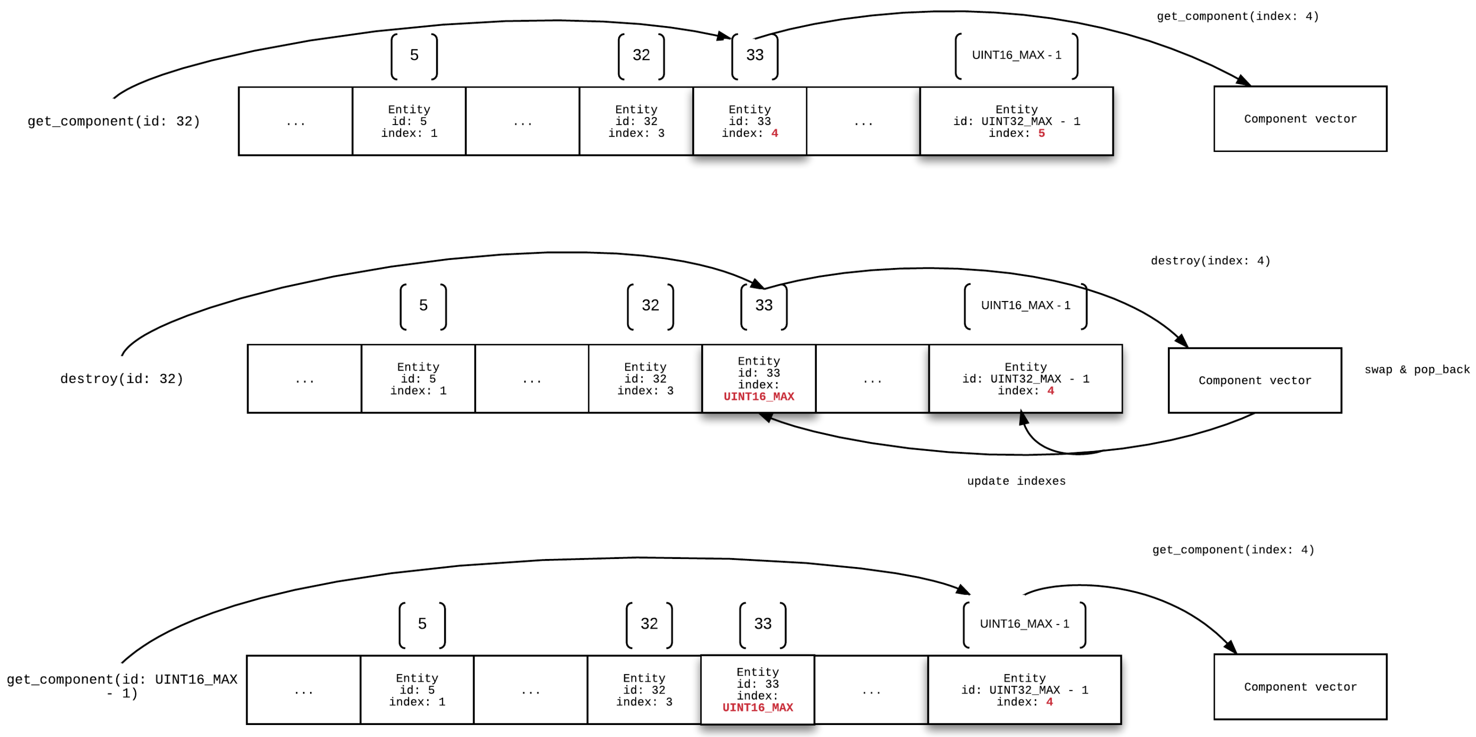


Figure 4: Using a layer of indirection to solve the inaccurate id problem

Entities Entities, shown in **Listing 2**, are designed to be lightweight and passed around the program in much the same fashion that primitive data types are. These are defined as `struct`s with a single id and index field alongside a `generation` field which is incremented whenever `UINT16_MAX` is reached and the `nextId_` field in the EntityMap (see **Section 3**) wraps to 0. This allows for a potential of 2^{256} total possible entities over the lifespan of the program while still keeping the entity lightweight - limited in size to 48 bits yet still containing lots of information about their state.

```cpp
/// Entity is an id and index pair that map this instance
/// to a series of systems and components, tying them all together
struct Entity {
  uint16_t id;
  uint16_t index;
  uint16_t generation;
};
```

Listing 2: Entity definition

Systems Using the above mentioned structures, systems are not required to be formally defined within the ECS and can be as simple as free functions declared in-line. However, part of a good ECS implementation is a good interface to match the usability of traditional OOP designs. Systems are designed to be entirely flexible with the only caveat be that they inherit from the base `System` class and implement the `add()`, `remove()`, and `has_entity()` abstract member functions to operate on their unique sets of data. These abstract functions are useful as each system may have more than one `ComponentData` container which may have unique requirements. Moreover, any potential performance hit caused by a single v-table lookup per system retrieval (once per frame) is minimal compared to the hit to programmer productivity that may occur without a defined interface for the whole ECS.

As a result, systems can be stored in homogeneous containers, allowing entities to have components attached and detached by simply iterating over all systems of interest and calling these polymorphic functions. Furthermore, as each system contains only its data of interest, the length of iteration over entities is limited to the length of its set of components and cache misses can be minimized. A trivial example of one such system is outlined in **Listing 3**.

```cpp
struct MovementSystem : public System {

  /// Movement vectors to operate on
  ComponentData<Vector2> moveVectors;

  /// Adds a new entity to this systems component data.
  /// \param entity Entity to add
  inline void add(const Entity& entity) override
  {
    moveVectors.attach(entity);
  }

  /// Removes an entity from this systems component data.
  /// \param entity Entity to remove
  inline void remove(const Entity& entity) override
  {
    moveVectors.detach(entity);
  }

  /// Checks if this system contains a specified entity
  /// \param entity Entity to check for
  /// \return True if has entity, false otherwise
  inline bool has_entity(const Entity& entity) override
  {
    return moveVectors.has_component(entity);
  }

  /// Moves all entities 1px down and to the right
  void move()
  {
    auto size = moveVectors.instances.size();
    for ( int e = 0; e < size; ++e ) {
      moveVectors.instances[e].x++;
      moveVectors.instances[e].y++;
    }
  }

};
```

Listing 3: An example system which moves all entities

Entity Map As is, systems, components and entities can be operated on in complete isolation without the need for some kind of centralized control or container class as long as careful manual management of entity id's occurs. However, as previously discussed, an approachable and centralized interface is desirable. Therefore, an `EntityMap` was implemented which maps systems, components, and entities to one another and handles the creation and destruction of entity id's, taking note of current generations and keeping track of the amount of live entities in existence. The EntityMap also stores each system within an `std::unordered_map<std::type_index, std::unique_ptr<System>>` shown in Listing 4 which, although consists of the previously mentioned overheads and negative cache impact, is useful in this situation as systems aren't expected to be removed and added dynamically often and are expected to be accessed only once per frame.

```
template <typename T>
inline T* get_system()
{
  return static_cast<T*>(systems_[typeid(T)].get());
}
```

Listing 4: Retrieving a system - Validation and error checking omitted for the purpose of a succinct example

Systems are registered to the entity map using `entityMap.add_system<T>()` to allow storage, retrieval and execution of each one from a central location, the implementation of which is outlined in **Listing** 5.

```cpp
template <typename T>
inline T* add_system()
{
  systems_.emplace(typeid(T), std::make_unique<T>());
  return get_system<T>();
}
```

Listing 5: Adding a system to the map - once again good programmers should implement some validation here

In regards to creation and destruction of entities, the EntityMap stores an integer, `nextId_`, which increments each time an entity is created but doesn't decrement when the entity is destroyed to avoid issues with conflicting id's. Instead, each time the id field reaches `UINT16_MAX` and wraps to zero, a second `currentGeneration_` field increments which, in combination with the entities id, creates a totally unique identifier, shown in **Listing 6**. It's important to note that the way this has been implemented, with separate id and generation fields, is for instructional purposes but in reality this would likely be implemented using a bit-mask over the id field with a certain amount of bits given to the generation and a certain amount to the id. Finally, an `std::unordered_map<std::string, Entity>` mapping between strings and entities allows for tagging and looking up entities by name. The full implementation of the EntityMap can be found in **Appendix A Listing 3**.

```cpp
inline Entity get_entity() {
  nextId_++;

  // Check for next generation
  if ( nextId_ == 0 )
    currentGeneration_++;

  Entity e;
  e.id = nextId_;
  e.index = UINT16_MAX;
  e.generation = currentGeneration_;

  return e;
}
```

Listing 6: Generating an entity

4. Testing Methods

The implementation outlined above was unit tested for correctness with the Google Test framework (Google 2016) (see **Appendix A.2**) and all performance metrics such as cache misses were measured using Apple's *Instruments* performance-analysis tool (Apple 2016). For opening and managing a window alongside drawing sprite, the popular media library SFML was used (SFML

2106). All tests were run on an early 2015 MacBook Pro Intel Core i5-5257U CPU 2.70GHz with 8GB RAM.

The Traditional OOP Comparison A typical OOP object hierarchy was built for comparisons, taking inspiration from the UE4 model outlined in **Figure 1** resulting in the hierarchy outlined in **Figure 5**. Each `GameObject` encapsulates a position vector, scale vector, while `Drawable` contains a sprite and draw/move functions, and `Character` contained collision rectangles and a collision function.

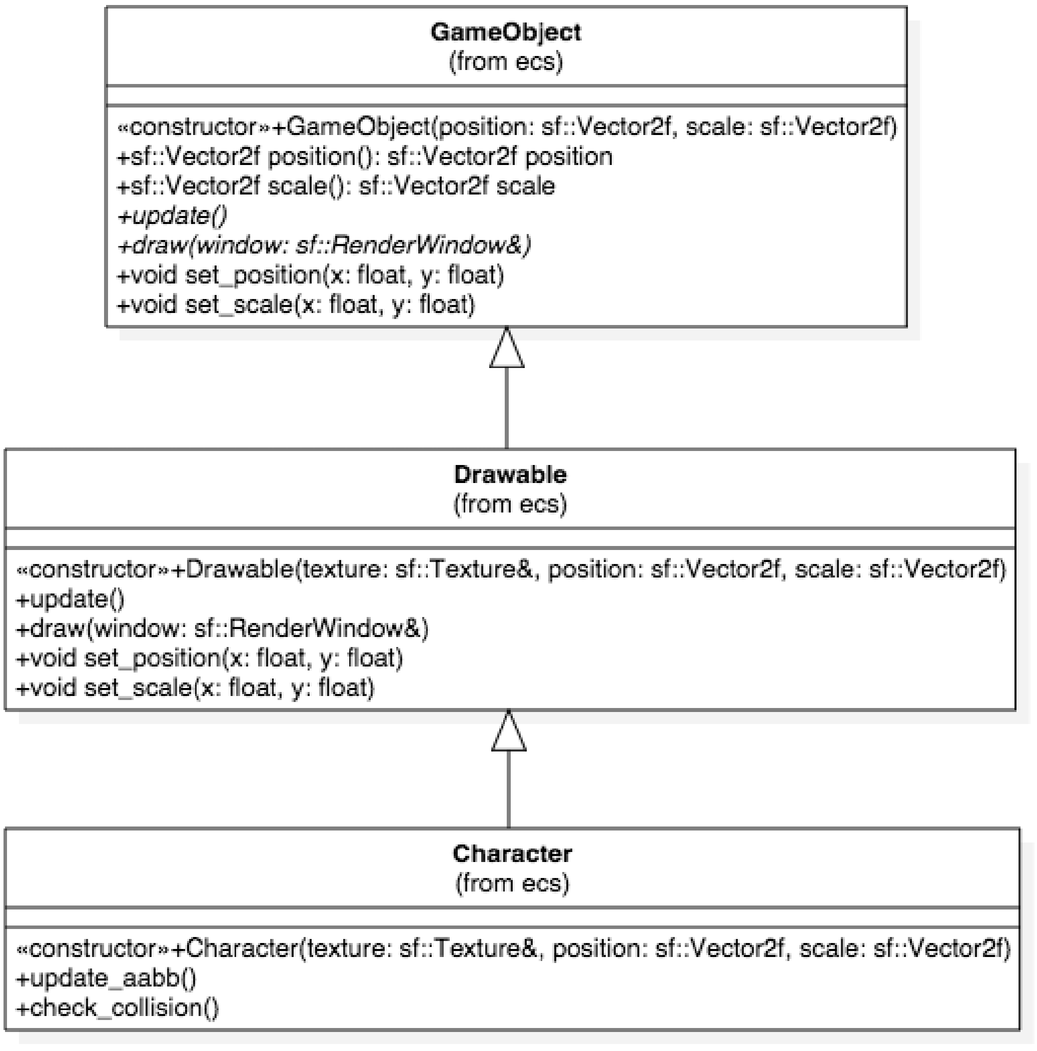

GameObject
(from ecs)
«constructor»+GameObject(position: sf::Vector2f, scale: sf::Vector2f)
+sf::Vector2f position(): sf::Vector2f position
+sf::Vector2f scale(): sf::Vector2f scale
+update()
+draw(window: sf::RenderWindow&)
+void set_position(x: float, y: float)
+void set_scale(x: float, y: float)
Drawable
(from ecs)
«constructor»+Drawable(texture: sf::Texture&, position: sf::Vector2f, scale: sf::Vector2f)
+update()
+draw(window: sf::RenderWindow&)
+void set_position(x: float, y: float)
+void set_scale(x: float, y: float)
Character
(from ecs)
«constructor»+Character(texture: sf::Texture&, position: sf::Vector2f, scale: sf::Vector2f)
+update_aabb()
+check_collision()

Figure 5: The traditional OOP game object comparison

Benchmark system A Benchmark was used in the form of a `std::vector` containing the relevant data (sprites, vectors etc.), and iterated over inside a free function in the same way as the above-mentioned methods.

Method A total of 65535 (`UINT16_MAX - 1`) objects were created for each system to operate on. For the traditional OOP objects a `std::vector<Character>` was used, polymorphism wasn't utilized to eliminate any overheads created by pointer dereferencing and casting, however it should be noted that polymorphic game object would commonly be used within a real OOP game engine. Each system was then executed in the sequence **MoveSystem → CollisionSystem → RenderSystem** with each one containing a single `for` loop and no branching. The full code for each system can be found in **Appendix A Listing 4**.
The systems were executed once per frame over 60 frames a total of twenty times each per system and their average execution time per 60 frames, number of iCache, L1, L2, and L3 cache misses were recorded alongside the total average execution time per system for the course of the test.

5. Running the Tests

Before discussing the results, it's important to note that games are often considered soft real-time systems and generally operate on a deadline of either 60 frames per second (FPS) or 30 FPS, i.e. $16.6ms$ or $33.3ms$ respectively (Gregory 2015, p. 152). For the purposes of this report, as the only operation being undertaken per system is moving and rendering a sprite, the acceptable deadline for each test can be considered $16.6ms$. Also taken into account is the amount cache misses recorded per frame as a ratio of $\frac{misses\ per\ frame}{total\ instructions}$, this is not considered a metric of performance but as a statistic to use in explaining results. The results of immediate interest for these tests are displayed below, see **Appendix B** for further results.

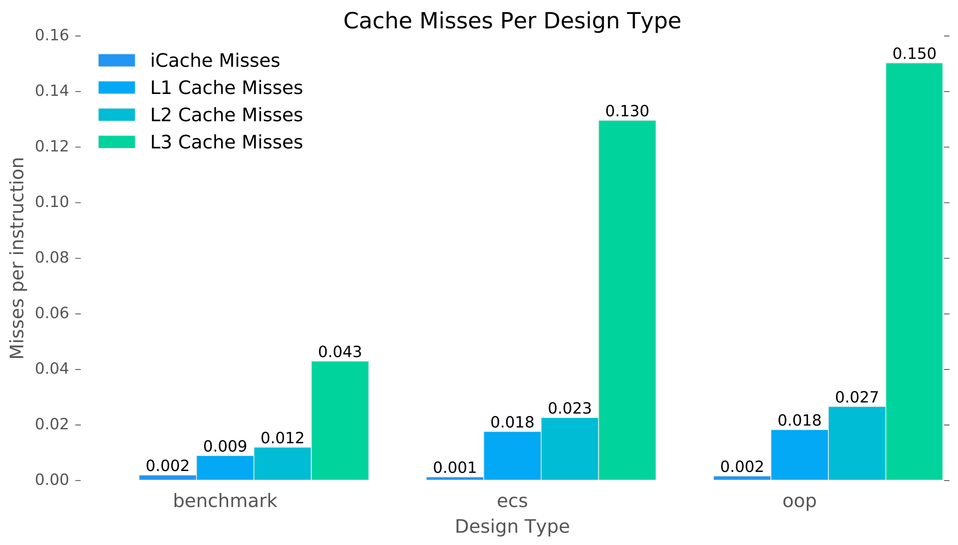


Figure 6: Cache misses per cache per instruction for each category of design type

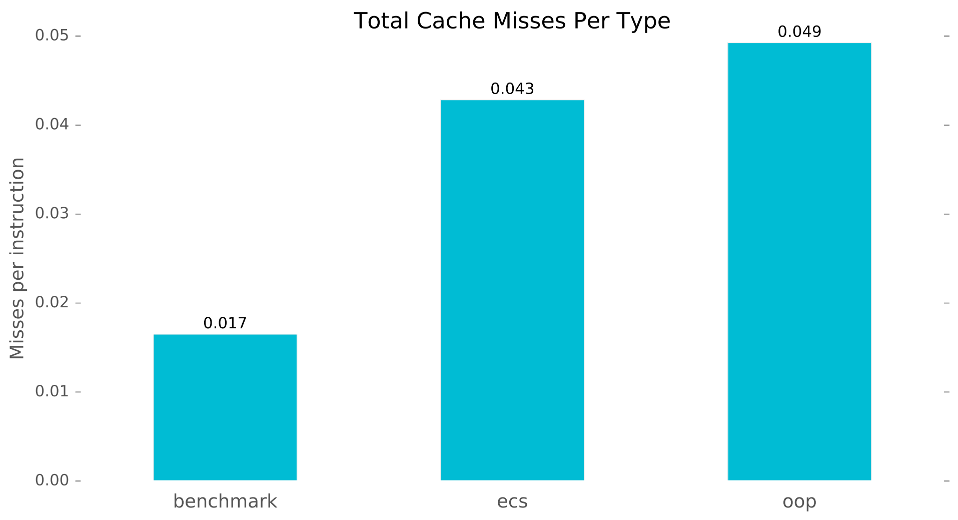


Figure 7: Combined cache misses per instruction for category design type

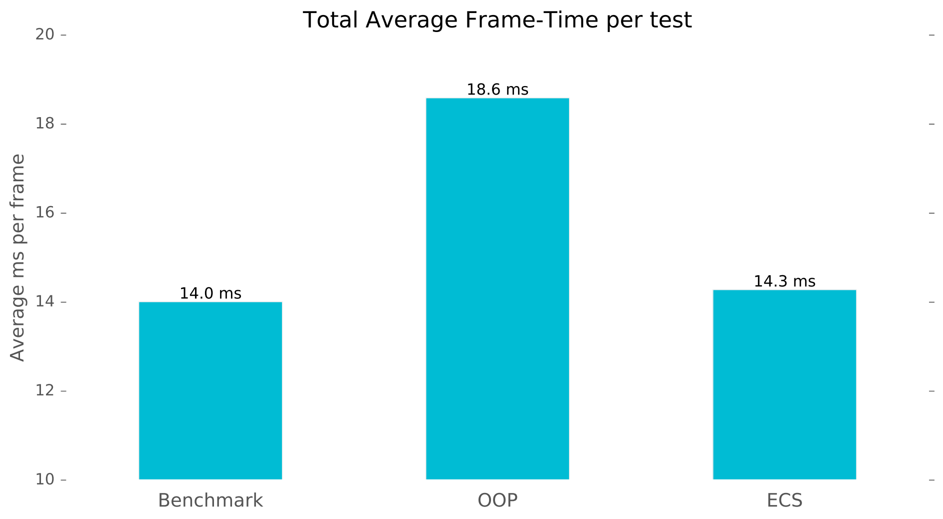


Figure 8: Average frame time per design type

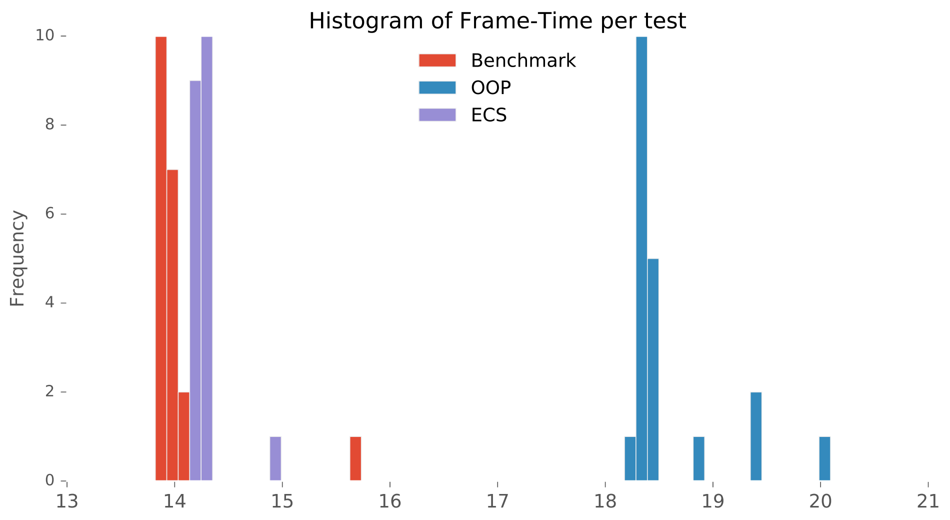


Figure 9: Histogram showing the spread of average frame times per 60 frame sample

Experiment Type	Mean	Median	Std. Deviation
Benchmark	14.000	13.926	0.391
OOP	18.590	18.389	0.488
ECS	14.279	14.253.609	0.167

Table 1: Run-Time Results

The results relating to cache usage showed that, as expected, the Benchmark system had quite a low miss rate on average per instruction while both the ECS and the OOP had much higher cache miss rates ($\bar{x} = 0.043$ & $\bar{x} = 0.049$ respectively). However, the run time results of each test were quite different, much more so than anticipated. While both the Benchmark and the ECS were on average able to draw all 65535 entities to the screen under 60*fps* ($\bar{x} = 14.000ms$ & $\bar{x} = 14.279ms$ respectively), the traditional OOP test was unable to draw a single group of 60 frames at the acceptable speed (under 16.6*ms*). Moreover, the variance of the OOP test ($s = 0.488$) was higher than the variance of the ECS test ($s = 0.167$), while the higher variance of the Benchmark can most likely be attributed to the fact that it was the first test to run on launch and as such, the first sample taken may have been affected by window start-up overhead, starting SFML systems etc. and therefore can be considered an outlier.

6. Comparisons of Designs

6.1. Run-Time

The traditional OOP design was much slower than initially expected, unable to draw a single group of 60 frames in under 16.6*ms*, whereas both the benchmark and ECS design came in under 16.6*ms* average. This may have to do with the amount of L2 misses that *also* missed the L3 cache, as an L3 cache miss means fetching data from main memory which can have a latency of ~60*ns* – ~100*ns* in many systems, versus the latency of an L3 read (an L2 miss), ~14*ns* (Levinthal

2009, p. 22). However, there isn't enough evidence to suggest that this is the case as other factors such as virtual memory usage etc. could have also contributed and as such, the cache results are of limited use in reasoning about why the run-time difference between the ECS and traditional OOP implementations is so large.
Furthermore, the **move & update** system for each design type suffered from enormous cache miss rates in comparison with all other systems. These functions made use of the SFML `Sprite::move()` library function, the only function called by these systems internally, and upon deeper inspection into the profiling results shows that this function was indeed responsible for each *move* systems cache misses. This is more of a reason to not look into the cache results too deeply as a single function call to an external library completely thrashed the L3. For further information see **Appendix** B. Ultimately, while there isn't enough data to explain fully why the differences were so large, it can be said that the ECS performed far better than the traditional OOP design in every test.

6.2. Flexibility

While difficult to measure, the ECS design was far more flexible than the traditional OOP design. This is due to a flat hierarchy and complete decoupling between systems and components - each part of the ECS can be altered, added, and removed in complete isolation. As components live inside templated containers, the only extra implementation for new systems is to create a new class derived from `ecs::System` and implement the base functions and only minimal consideration must be made for other systems, essentially thinking only about the *order* in which the systems are called to maximize cache locality and reuse of data. In contrast, the OOP design must be reasoned about much more carefully, thinking about the overall inheritance tree, where new data or logic should be placed within it. Moreover, while relatively simple, the inheritance structure used for this report already created multiple dependency and coupling issues where any alteration made to the **Drawable** or **GameObject** classes affected any class down the inheritance tree - a very undesirable result for flexibility concerns.
However, due to the fact that within an ECS a game object properties are no longer unified and encapsulated inside a single class, but spread across the program, the problem of *communication between components*, an area where the traditional OOP implementation shines, becomes an issue. While a somewhat intricate problem and not explicitly solved here, the overall simplest solution can be to create a centralized messaging system - each system holds a pointer to the message bus, allowing inter-system and inter-component communication.

6.3. Maintainability

One of the major areas that the ECS implementation performs best at is testability and isolation. Each part of the ECS - the systems, component data, entity map - are able to be tested easily without requiring other parts to be implemented before testing or extra *getter* and *setter* functions to be defined. Ultimately, one of the major purposes of an ECS is to **expose** rather than *hide* data that is operated on frequently as it is essential for maximizing efficiency and flexibility. In contrast the OOP implementation required each data member that needed to be operated on to implement a matching *getter*, a very idiomatic and typical use of encapsulation within OOP programs, however this requires far more boilerplate and helper functions before being able to test it as thoroughly as the ECS implementation.
Moreover, OOP design generally treats each object as an individual, separated from the rest of a program, and while can be useful for identifying issues in isolation, the dependencies that hierarchy and the use of inherited data can cause tends to remove the ability to isolate these

problems. This isn't to say that this is the case in all, or even most, OOP programs - it's to say that ensuring that this doesn't occur can be difficult within the context of OOP, as the paradigm facilitates this as a core principle in the use of polymorphism and inheritance. In contrast, the ECS design in this report was implemented to ensure that *dependencies are very difficult to create*:

- All data is separated into `ComponentData` containers
- All systems are self-contained and operate on a very specific subset of self-contained data
- Entities (game objects) have no dependencies as they are just an ID

This ensures that any errors created due to dependencies are extremely rare. As a side-effect to this design choice, errors proved extremely simple to isolate - they can only occur within a single function and across a single piece of data.

7. Conclusions - Combining ECS and OOP

Data-Oriented, Entity-Component-Systems and good Object-Oriented design do not have to be mutually exclusive. Programmer productivity and usability can be as important to the development of a game engine as efficiency and modularity. As shown here, these factors can be combined into an efficient and highly maintainable, flexible, and usable interface. The modularity of an ECS facilitates the easy use of Data-Driven practices as entities are really just groups of properties that can, with a little effort, be implemented using data files such as JSON and XML as well as editor tools.
We've seen how using OOP principles with restraint can lead to a more desirable framework than without - the use of inheritance allows *Systems* to avoid the need to redefine boiler-plate code, subtype polymorphism can be used to store and operate on each system within a centralized control point, while parametric polymorphism (templates) can be used to define extremely useful containers (`ComponentData`) that can store any type of data required. Moreover, both the principles of abstraction and encapsulation can be used to create a centralized entity map, an approachable interface.
Ultimately the combination between a well thought out, restrained OOP design, and a highly decoupled ECS that places an emphasis on reasoning about data at the hardware level, can be a possible solution for data-oriented game development; enabling flexibility, generality, and efficiency within an engine.

References

Amazon (2016). *Best Sellers in Computer CPU Processors*. Webpage. URL: `https://www.amazon.com/Best-Sellers-Electronics-Computer-CPU-Processors/zgbs/electronics/229189`.

Apple (2016). *Instruments*. Software Application. (Version 8.0).

Bilas, Scott (2002). *A Data-Driven Game Object System*. Presentation Slides. URL: `http://gamedevs.org/uploads/data-driven-game-object-system.pdf`.

Comstock, Craig, Zhizhong Jiang, and Peter Naudé (2007). "Strategic Software Development: Productivity Comparisons of General Development Programs". In: *International Journal of Computer and Information Science and Engineering* 1.4, pp. 224–229.

Dice (2010). *Introduction To Data-Oriented Design.* Electronic. URL: http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf.

– (2013). *Culling The Battlefield.* Electronic. URL: http://www.frostbite.com/wp-content/uploads/2013/05/CullingTheBattlefield.pdf.

Google (2016). *Google Test.* Software Application. (Version 1.80). URL: https://github.com/google/googletest.

Gregory, Jason (2015). *Game Engine Architecture.* 2nd Edition. Boca Raton, Florida, US: CRC Press.

Griliopoulos, Dan (2011). *A History of Id Tech.* Web article. URL: http://au.ign.com/articles/2011/04/28/a-history-of-id-tech.

Leonard, Tom (1999). *Postmortem: Thief: The Dark Project.* Electronic. URL: http://www.gamasutra.com/view/feature/3355/postmortem_thief_the_dark_project.php.

Levinthal, David (2009). *Performance Analysis Guide for Intel® Core™ i7 Processor and Intel® Xeon™ 5500 processors.* 1.0. Intel. Santa Clara, California, US. URL: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.

Llopis, Noel (2009). *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP).* URL: http://gamesfromwithin.com/data-oriented-design (visited on 12/04/2009).

López, Cuauhtémoc, Martín Arturo Chavoya, and Maria Elena Meda Campaña (2012). "Comparison of software development productivity based on object-oriented programming languages". In: *The 2012 International Conference on Software Engineering Research & Practice (SERP 2012).* Ed. by Hamid R. Arabnia and Jay Xiong Hassan Reza. Vol. 1. Las Vegas, Nevada, US, pp. 47–52.

Newegg (2016). *10 best selling cpus 2016 first half.* Website. URL: https://blog.neweggbusiness.com/components/10-best-selling-cpus-2016-first-half/.

SFML (2106). *Simple and Fast Multimedia Library.* Software Library. (Version 2.4.1). URL: http://www.sfml-dev.org/index.php.

stackoverflow.com (2016). *Stack Overflow 2016 Developer Survey.* URL: http://stackoverflow.com/research/developer-survey-2016#technology (visited on 2016).

UE4 Programming Guide (2016). Epic Games. North Carolina, US.

Al-Zoubi, Hussein, Aleksandar Milenkovic, and Milena Milenkovic (2004). "Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite". In: *Proceedings of the 42Nd Annual Southeast Regional Conference.* ACM-SE 42. Huntsville, Alabama: ACM, pp. 267–272. ISBN: 1-58113-870-9. DOI: 10.1145/986537.986601. URL: http://doi.acm.org.ezproxy.lib.swin.edu.au/10.1145/986537.986601.

A. Appendix A - Code

A.1. Implementations

```
/// Entity is an id and index pair that map this instance
/// to a series of systems and components, tying them all together
struct Entity {
  uint16_t id;
  uint16_t index;
  uint16_t generation;
};
```

Listing 1: ComponentData implementation

```
/// \brief ComponentData is a container of components with garunteed
 ↪ contiguous
/// storage that can be queried with an Entity id
/// \tparam Type of component to store
template <typename T>
struct ComponentData {

  /// \brief All instances of this component mapped to an entity
  std::vector<T> instances;

  /// \brief Initializes the container and lookup table
  ComponentData()
  {
    for ( int e = 0; e < UINT16_MAX; ++e ) {
      lookup_[e].id = e;
      lookup_[e].index = UINT16_MAX;
      lookup_[e].generation = UINT16_MAX;
    }
  }

  /// \brief Gets an entities associated component as a pointer.
  /// \details Uses a pointer rather than a reference to allow for
   ↪ better
  /// auto compatibility and checking against nullptr
  /// \param entity The entity whose component is being retrieved
  /// \return Pointer to the component instance if found, nullptr
   ↪ otherwise
  inline T* get(const Entity& entity)
  {
    auto lookup = lookup_[entity.id];
    if ( lookup.generation != entity.generation )
      return nullptr;
    return (lookup.index < UINT16_MAX) ? &instances[lookup.index] :
     ↪  nullptr;
  }

  /// \brief Checks to see if the entity has an instance of the
   ↪  component
  /// \param entity To check for
  /// \return True if found, false otherwise
  inline bool has_component(const Entity& entity)
  {
    auto lookup = lookup_[entity.id];
    if ( lookup.generation != entity.generation )
      return false;

    return lookup.index < instances.size() && lookup.index <
     ↪  UINT16_MAX;
  }

  /// \brief Attaches an instance of the component to an entity
  /// \param entity Entity to attach
  /// \return The entity with id and index into the data
  inline Entity attach(const Entity& entity)
  {
    instances.emplace_back();
    Entity e;
    e.index = instances.size() - 1;
    e.id = entity.id;
    e.generation = entity.generation;
    lookup_[e.id] = e;
    return e;
  }

  /// \brief Removes an entity from this container
  /// \param entity Entity to remove
  inline void detach(const Entity& entity)
  {
    auto index = lookup_[entity.id].index;
    std::swap(*get(entity), instances.back());
    instances.pop_back();
    lookup_[entity.id].index = UINT16_MAX;
  }

private:
  /// \brief Lookup array used to index into the actual data
  Entity lookup_[UINT16_MAX];
};
```

Listing 2: ComponentData implementation

```
/// \brief EntityMap maps ComponentData, System, and Entity
↪ instances to one
/// another and coordinates creation, destruction, and component
↪ adding for all
/// entities. All operations on data should be done via this
↪ interface.
class EntityMap {
public:

  /// \brief Initializes a new EntityMap
  EntityMap() : nextId_(0), currentGeneration_(0), size_(0) {}

  /// \brief Registers a system to this EntityMap for convenient
  ↪ lookup
  /// \tparam Type of System to add
  /// \return Pointer to the System instance
  template <typename T>
  inline T* add_system()
  {
    systems_.emplace(typeid(T), std::make_unique<T>());
    return get_system<T>();
  }

  /// \brief Creates a new Entity
  /// \return The new entity
  inline Entity create()
  {
    size_++;
    return get_entity();
  }

  /// \brief Creates a new Entity with the specified tag
  /// \return The new entity
  inline Entity create(const std::string& tag)
  {
    Entity e = create();
    tags_[tag] = e;
    return e;
  }

  /// \brief Destroys an entity by removing it from all registered
  ↪ System instances
  /// \param entity Entity to remove
  void destroy(const Entity &entity)
  {
    size_--;
    for ( auto &s : systems_ ) {
      s.second->remove(entity);
    }
  }

  /// \brief Gets a tagged entity
  /// \param tag Tag to search for
  /// \return The entity
  Entity get_tagged_entity(const std::string& tag)
  {
    return tags_[tag];
  }

  /// \brief Attaches an Entity to the specified System instance
  /// \tparam Type of System to attach to
  /// \param entity Entity to attach
  template <typename T>
  inline void attach(const Entity &entity)
  {
    systems_[typeid(T)]->add(entity);
  }

  /// \brief Removes an Entity from the specified System instance
  /// \tparam Type of System to remove from
  /// \param entity Entity to remove
  template <typename T>
  inline void remove(const Entity& entity)
  {
    systems_[typeid(T)]->remove(entity);
  }

  /// \brief Gets a pointer to the specified registered System
   ↪  instance
  /// \tparam Type of System to get
  /// \return Pointer to the System
  template <typename T>
  inline T* get_system()
  {
    return static_cast<T*>(systems_[typeid(T)].get());
  }

  /// \brief Checks if an entity belongs to a specific system
  /// \param entity Entity to check
  /// \return True if belongs to system, false otherwise
  template<typename T>
  inline bool belongs_to(const Entity& entity)
  {
    return systems_[typeid(T)]->has_entity(entity);
  }
```

```
  /// \brief Gets the number of Entity instances currently alive
  /// \return Number of alive Entity instances
  inline uint64_t size() { return size_; }

private:
  /// \brief Map for lookup and retrieval of stored System instances
  std::unordered_map<std::type_index, std::unique_ptr<System>>
   ↪  systems_;
  /// \brief All tagged entities
  std::unordered_map<std::string, Entity> tags_;
  /// \brief The next available unique ID for assigning to Entity
   ↪  instances
  uint16_t nextId_;
  /// \brief The current generation of entity id's
  uint16_t currentGeneration_;
  /// \brief The number of currently living entities
  uint64_t size_;

  /// \brief Gets the next available Entity with id and generation
  /// \return
  inline Entity get_entity() {
    nextId_++;

    // Check for next generation
    if ( nextId_ == 0 )
      currentGeneration_++;

    Entity e;
    e.id = nextId_;
    e.index = UINT16_MAX;
    e.generation = currentGeneration_;

    return e;
  }
};
```

Listing 3: EntityMap implementation

A.2. Tests

```
/// \brief SpriteSystem operates on a set of sf::Sprite data,
 ↪  rendering them
/// to an sf::RenderWindow
struct SpriteSystem : public System {
```

```cpp
  /// Sprite data to operate on
  ComponentData<sf::Sprite> spriteData;

  /// \brief Adds a new entity to this systems component data.
  /// \param entity Entity to add
  inline void add(const Entity& entity) override
  {
    spriteData.attach(entity);
  }

  /// \brief Removes an entity from this systems component data.
  /// \param entity Entity to remove
  inline void remove(const Entity& entity) override
  {
    spriteData.detach(entity);
  }

  /// \brief Checks if this system contains a specified entity
  /// \param entity Entity to check for
  /// \return True if has entity, false otherwise
  inline bool has_entity(const Entity& entity) override
  {
    return spriteData.has_component(entity);
  }

  /// \brief Moves all sprites 1px down and to the right
  void move()
  {
    auto size = spriteData.instances.size();
    for ( int e = 0; e < size; ++e ) {
      spriteData.instances[e].move(1, 1);
    }
  }

  /// Renders all sprites to a window
  /// \param window Widow to render to
  void render(sf::RenderWindow &window)
  {
    auto size = spriteData.instances.size();
    for ( int e = 0; e < size; ++e ) {
      window.draw(spriteData.instances[e]);
    }
  }

};

/// \brief CollisionSystem is a placeholder for a collision checking
↪  system.
```

```cpp
// Currently it only reassigns a rectangle component
struct CollisionSystem : public System {

  /// The collision boxes to operate on
  ComponentData<sf::FloatRect> boxes;

  /// \brief Adds a new entity to this systems component data.
  /// \param entity Entity to add
  inline void add(const Entity& entity) override
  {
    boxes.attach(entity);
  }

  /// \brief Removes an entity from this systems component data.
  /// \param entity Entity to remove
  inline void remove(const Entity& entity) override
  {
    boxes.detach(entity);
  }

  /// \brief Checks if this system contains a specified entity
  /// \param entity Entity to check for
  /// \return True if has entity, false otherwise
  inline bool has_entity(const Entity& entity) override
  {
    return boxes.has_component(entity);
  }

  /// \brief Reassigns each collision box to a new sf::FloatRect
  void update_collision(const sf::FloatRect &rect)
  {
    auto size = boxes.instances.size();
    bool intersect = false;

    for ( int e = 0; e < size; ++e ) {
      intersect = boxes.instances[e].intersects(rect);
    }
  }

};
```

Listing 4: Systems used in the test

```cpp
//
// Main.cpp
// ECS_Framework
//
```

```
// ----------------------------------------------------------------------
↪ -----------
//
// Created by Jacob Milligan on 29/10/2016.
// Copyright (c) 2016 Jacob Milligan All rights reserved.
//

#include <iostream>
#include <fstream>
#include <vector>

#include <SFML/Graphics.hpp>
#include <FPSCounter.hpp>
#include <GameObject.hpp>
#include <Entity.hpp>

/// \brief Fills a vector of sprites with the needed data
/// \param sprites Vector to fill
/// \param numEntities Number of entities to create
/// \param texture Texture to assign to each sprite
void setup_raw(std::vector<sf::Sprite> &sprites, const int
↪ numEntities, sf::Texture &texture)
{
  auto textureSize = texture.getSize();

  for ( int i = 0; i < numEntities; ++i ) {
    sprites.emplace_back(texture);
    sprites.back().setPosition(i * textureSize.x, i);
  }
}

/// \brief Sets up all game objects with the necessary data
/// \param gameObjects Container to setup
/// \param numEntities Number of entities to create
/// \param texture Texture to assign to each sprite
void setup_oo(std::vector<ecs::Character> &gameObjects, const int
↪ numEntities, sf::Texture &texture, sf::FloatRect &rect)
{
  auto textureSize = texture.getSize();

  for ( int i = 0; i < numEntities; ++i ) {
    gameObjects.emplace_back(
      texture, sf::Vector2f(i * textureSize.x, i), sf::Vector2f(1,
      ↪ 1), rect
    );
  }
}
```

```cpp
/// \brief Updates the game objects calling their update, draw etc.
 ↪  functions in the
/// correct order
/// \param gameObjects Container to update
/// \param window Window to draw the game objects sprites to
void update_game_objects(std::vector<ecs::Character> &gameObjects,
 ↪  sf::RenderWindow &window)
{
  for ( int i = 0; i < gameObjects.size(); ++i ) {
    gameObjects[i].update();
  }
  for ( int i = 0; i < gameObjects.size(); ++i ) {
    gameObjects[i].update_aabb();
  }
  for ( int i = 0; i < gameObjects.size(); ++i ) {
    gameObjects[i].draw(window);
  }
}

/// Updates a sprite vector and draws them to the window
/// \param sprites Vector of sprites to update
/// \param window Window to draw sprites to
void update_sprites(std::vector<sf::Sprite> &sprites,
 ↪  sf::RenderWindow &window)
{
  auto size = sprites.size();
  for ( int i = 0; i < size; ++i ) {
    sprites[i].move(1, 1);
  }
  for ( int i = 0; i < size; ++i ) {
    window.draw(sprites[i]);
  }
}

/// \brief Sets up the ECS with the needed data
/// \param ecs EntityMap to setup
/// \param numEntities Number of entities to create
/// \param texture Texture to allocate to each entities Sprite
 ↪  component
void setup_dod(ecs::EntityMap &ecs, const int numEntities,
 ↪  sf::Texture &texture)
{
  auto render = ecs.add_system<ecs::SpriteSystem>();
  auto movement = ecs.add_system<ecs::CollisionSystem>();
  auto textureSize = texture.getSize();

  for ( int i = 0; i < numEntities; ++i ) {
    auto e = ecs.create();
    ecs.attach<ecs::SpriteSystem>(e);
    ecs.attach<ecs::CollisionSystem>(e);

    render->spriteData.get(e)->setTexture(texture);
    render->spriteData.get(e)->setPosition(i * textureSize.x, i);

    *(movement->boxes.get(e)) =
     ↪  render->spriteData.get(e)->getLocalBounds();
  }
}

/// \brief Entry point
int main(int argc, char** argv) {

  auto numEntities = UINT16_MAX;

  auto testType = 0;
  auto iteration = 0;
  std::vector<std::tuple<int, float>> averages;

  // Get the desktops current resolution and divide it by 1.3
  // to make it smaller
  auto mode = sf::VideoMode::getDesktopMode();

  // Open a new window with the video mode and the title 'Entity
   ↪  Framework'
  sf::RenderWindow window(mode, "Entity Framework");

  sf::Texture texture;
  texture.loadFromFile(
    "file.png"
  );

  std::vector<sf::Sprite> rawSprites;
  ecs::EntityMap ecs;
  std::vector<ecs::Character> testObjects;

  sf::FloatRect check(10, 10, 10, 10);

  setup_dod(ecs, numEntities, texture);
  setup_oo(testObjects, numEntities, texture, check);
  setup_raw(rawSprites, numEntities, texture);

  ecs::FPSCounter fps;
  ecs::FPSCounter sampleFps;

  std::ofstream csv(
    "data.csv"
```

```
);
csv << "Experiment,Sample,Average Frame Time\n";

while ( window.isOpen() && testType < 3 ) {

  while ( sampleFps.total_frames() <= 60.0f ) {
    sf::Event event;

    while ( window.pollEvent(event) ) {

      if ( event.type == sf::Event::Closed )
        window.close();

    }

    window.clear(sf::Color::Black);

    switch (testType) {
      case 0:
        update_sprites(rawSprites, window);
        break;
      case 1:
        update_game_objects(testObjects, window);
        break;
      case 2:
        ecs.get_system<ecs::SpriteSystem>()->move();
        ecs.get_system<ecs::CollisionSystem>()-
         ↪ >update_collision(check);
        ecs.get_system<ecs::SpriteSystem>()->render(window);
        break;
    }

    fps.update();
    sampleFps.update();

    window.display();
  }

  csv << testType << "," << iteration
      << "," << sampleFps.average_delta_time() << "\n";

  iteration++;
  sampleFps.reset();

  if ( iteration > 19 ) {
    averages.emplace_back(testType, fps.average_delta_time());
    iteration = 0;
    testType++;
```

```
      fps.reset();
    }
  }

  window.close();

  csv.close();

  return 0;
}
```

Listing 5: The main tests and functions used for non-ECS tests

```
//
// EntityTests.cpp
// ECS_Framework
//
// -----------------------------------------------------------------
   ↪ -----------
//
// Created by Jacob Milligan on 30/10/2016.
// Copyright (c) 2016 Jacob Milligan All rights reserved.
//

#include <gtest/gtest.h>
#include <vector>
#include "Entity.hpp"

namespace ecs {

TEST(test_component_data, entity_is_added)
{
  ecs::ComponentData<sf::Vector2f> data;
  auto max = 50;
  Entity e;

  for ( int i = 0; i < max; ++i ) {
    e.id = i;
    e = data.attach(e);
  }

  ASSERT_EQ(data.instances.size(), max);
  ASSERT_TRUE(data.has_component(e));
}

TEST(test_component_data, entity_is_removed)
{
  ecs::ComponentData<sf::Vector2f> data;
  auto max = 50;
  std::vector<Entity> entities;

  for ( int i = 0; i < max; ++i ) {
    Entity e;
    e.id = i;
    e = data.attach(e);
    entities.push_back(e);
  }

  ASSERT_EQ(data.instances.size(), max);
  ASSERT_TRUE(data.has_component(entities[34]));

  for ( int i = 0; i < max; ++i ) {
    data.detach(entities[i]);
  }

  ASSERT_EQ(data.instances.size(), 0);
  ASSERT_FALSE(data.has_component(entities[10]));
}

TEST(test_component_data, component_data_changes)
{
  ecs::ComponentData<sf::Vector2f> data;
  Entity e;
  e.id = 1;
  e = data.attach(e);

  auto vec = data.get(e);
  vec->x = 100;
  vec->y = 234;

  ASSERT_EQ(data.get(e)->x, 100);
  ASSERT_EQ(data.get(e)->y, 234);
}

TEST(test_component_data, component_data_doesnt_exist)
{
  ecs::ComponentData<sf::Vector2f> data;
  Entity e;
  e.id = 1;

  auto vec = data.get(e);

  ASSERT_EQ(vec, nullptr);
}
```

```
TEST(test_component_data, systems_are_added)
{
  ecs::EntityMap ecs;
  ecs.add_system<ecs::CollisionSystem>();
  auto e = ecs.create();
  ecs.attach<ecs::CollisionSystem>(e);
  auto movement = ecs.get_system<ecs::CollisionSystem>();

  ASSERT_NE(movement, nullptr);
}

TEST(test_component_data, component_is_detached)
{
  ecs::EntityMap ecs;
  ecs.add_system<ecs::CollisionSystem>();
  auto e = ecs.create();
  ecs.attach<ecs::CollisionSystem>(e);
  auto movement = ecs.get_system<ecs::CollisionSystem>();

  ecs.remove<ecs::CollisionSystem>(e);

  ASSERT_FALSE(movement->boxes.has_component(e));
}

TEST(test_component_data, entity_is_destroyed)
{
  ecs::EntityMap ecs;
  ecs.add_system<ecs::CollisionSystem>();
  auto e = ecs.create();
  ecs.attach<ecs::CollisionSystem>(e);
  auto movement = ecs.get_system<ecs::CollisionSystem>();

  ecs.destroy(e);

  ASSERT_FALSE(movement->boxes.has_component(e));
  ASSERT_EQ(ecs.size(), 0);
}

TEST(test_component_data, generations_work)
{
  ecs::EntityMap ecs;
  ecs.add_system<ecs::CollisionSystem>();

  auto e1 = ecs.create();
  for ( int i = 0; i < UINT16_MAX; ++i ) {
    ecs.create();
  }
  auto e2 = ecs.create();
  ecs.attach<CollisionSystem>(e1);

  ASSERT_TRUE(ecs.belongs_to<CollisionSystem>(e1));

  ecs.attach<CollisionSystem>(e2);

  ASSERT_TRUE(ecs.belongs_to<CollisionSystem>(e2));
}

TEST(test_component_data, tags_Work)
{
  ecs::EntityMap ecs;
  ecs.add_system<ecs::CollisionSystem>();

  auto e1 = ecs.create("player");
  auto e2 = ecs.get_tagged_entity("player");

  ASSERT_EQ(e1.id, e2.id);
}


}
```

Listing 6: ECS Unit tests

B. Appendix B - Results

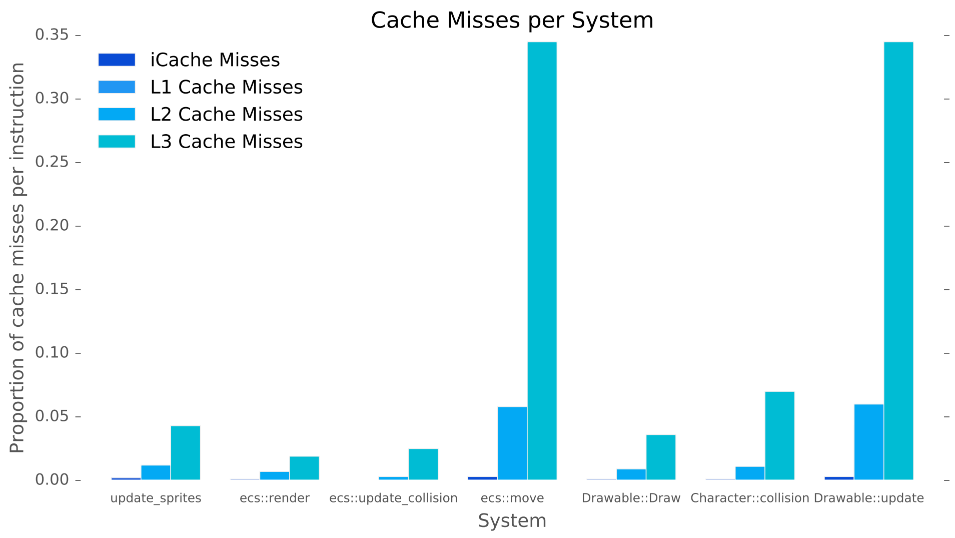


Figure 1: Cache misses per cache per instruction for each system