Data-oriented Entity Component System

Jungwan Byun
jungwan82@naver.com

July 2019

Version 1.0

Abstract

Entity Component System(ECS) is one of the methods for architecturing programs and well known for its flexibility. In this method, the architecturing problem is solved by separating data and logic. Related data is grouped together and forms a Component. Logic resides in Systems and transforms Data. As a result, we can understand that Systems transform Components. Related Components are grouped together and form an Entity.

Data-oriented Entity Component System is the ECS with a high CPU cache hit rate. We can create high performing ECS by allocating components in the consecutive area of memory.

Disclaimer

DOECS project in Github is experimental and may contain bugs or errors in terms of program integrity and performance. As so, the content of this document may do. Any discussions or grammar corrections are welcome.

Contents

1 What is ECS

ECS is an abbreviation for Entity Component System. In actual implementation, Entities are just a unique number. Components are data set and Systems are logic. We will look into Entity, Component and System separately, and how they work together organically to understand what the ECS is.

1.1 Entity

An entity represents every actor and non-actor existing in the program, changes their properties(components) and interacts with each other through Systems. An entity has a set of components. In other words, an entity consists of several components that describe what the entity is.

In the actual code, an entity is just an integer number but in some cases it could be a class with additional properties and functions for convenience.

```
using EntityId = uint64_t;
```

Listing 1: The EntityId is just defined as an interger.

1.2 Component

Component is a placeholder for the entity properties in the same category. An entity has several components. In other words, several components represent an entity. We can put a component as a common denominator among certain types of entities. For example, two entities, E1 and E2 can have C1 component each, so the C1 is the common denominator.

As the number of properties residing in a component increases, the reusability(chances to be a common denominator) of the component decreases. Conversely, if the number of properties in a component is too low, the number of components required to form an entity increases and this causes difficulty in managing the software.

```
struct LocationComponent
{
    int X;
    int Y;
}

struct MovementComponent
{
    float Speed;
    Float MovementRemained;
    int TargetX;
    int TargetY;
    Coordinates Path[10];
    int NextPathIndex;
    int ValidPathCount;
}
```

Listing 2: Examples of Components.

1.3 System

We can consider entities and components as data, and systems as data processors. A system defines how data should be transformed in certain condition and how entities interact with each other.

The System doesn't know anything about Entity itself. i.e. the system is independent from entities. Its only interests are components.

```
void MovementSystem::Execute(uint32_t elementCount, LocationComponent*
    locComps, MovementComponent* moveComps)
{
    for(int i =0; i< elementCount; ++i)
    {
        ProcessMovement(locComps[i], moveComps[i]);
    }
}
```

Listing 3: An example of a system.

2 What is Data-oriented

Data-oriented programs can read data from the memory very quickly. The key point is allocating consecutive memory space for data and processing the data in order from the beginning to the end. By doing this, the CPU cache hit rate would go high and this gives huge performance benefits.

Why is the CPU cache so important? For the recent years, the speed of CPU development has been far faster than the memory development. It happens very often that CPU is ready to compute the data but should wait on idle for retrieving the data from the memory.

To overcome this situation, there is a small size of cache in the CPU. Data retrieved from the memory are kept in the cache chunked in size of 64 bytes. If CPU requests the same data that is already in the cache, it reuses the cache item instead of the data in the memory. Reusing the data in cache is about 50 times faster than newly retrieving the data from the memory.

Data-oriented programs are designed considering these traits of the CPU, so perform better than other programs.

3 Data-oriented and ECS

ECS can have thousands of entities and more. ECS also has a number of systems to execute operations on these entities' data(components). We can implement ECS without considering the concept of 'Data-oriented'. However, this implementation will suffer from cache misses caused by reading scattered entity data and executing operations on them, which would result in low performance.

On the other hand, Data-oriented ECS minimizes the cache misses and shows higher performance. Entities' data(which are components) is located in memory consecutively and the systems execute operations on the data in batch.

How can we implement Data-oriented ECS? In a later chapter, we are going to see every detail but here, let's just briefly look at the performance penalty of NOT Data-oriented ECS.

```
 1  class MyEntity : public IEntity
 2  {
 3      struct MyComponent1* Comp1;
 4      struct MyComponent2* Comp2;
 5      struct MyComponent3* Comp3;
 6      struct MyComponent4* Comp4;
 7  }
 8
 9  void MySystem(IEntity* entity) {
10      MyComponent1* pComp1 = entity->GetComp<MyComponent1>();
11      MyComponent4* pComp4 = entity->GetComp<MyComponent4>();
12      if (!pComp1 || !pComp4)
13          return;
14
15      // Logic goes here
16  }
```

Listing 4: Not Data-oriented ECS.

In line 3 to 6, we are defining the components that are grouped and form an Entity. These components should be dynamically allocated at run-time. The allocations don't guarantee that every component will be located in the consecutive area of the memory.

Line 9 defines a System function. This function gets an entity as a parameter. Somewhere else in the code, we will iterate all the entities and call all the system functions with the entity. In the system function, we are going to check the entity to find out whether the entity has appropriate components to process the system. If not, it will ignore the entity by returning from the system function as shown in line 12 and 13. This doesn't work well with the CPU's branch prediction algorithms. Once the execution pointer reaches to line 15, the logic is going to access components data that is highly likely not in the CPU cache. The program has to repeat those processes for thousands of entities.

The biggest difference between Data-oriented ECS and the normal ECS is in the method of creating components. In the ‘normal ECS’ example, ‘new operator’ is used for dynamic allocation of Components, but in the ‘Data-oriented ECS’, the components are pre-allocated in the consecutive area of memory pool and one of them will be used for a new entity, so it can avoid dynamic allocation.

3.1 Data-oriented and Multi-core processing

It helps better performance to run the systems in parallel since we can fully utilize the CPU. To do this, we need to understand how CPU cores and caches are organized.

Cache configuration of Intel i7 8700 is:

- L1 Data cache 32 KB per core,
- L1 Instruction cache 32 KB per core,
- L2 cache 256 KB per core,
- L3 cache 12 MB shared

Each core has its own L1 and L2 caches. Therefore, it is possible to maintain the high cache hit rate while each core is executing different Systems with corresponding datasets[1] that are not adjacent.

4 Implementation

In this section, we are going to look into implementation details of Data-oriented ECS. The language we are going to use is C++.

The key is sequential data and batch processing. To achieve this, we are going to introduce the concept of archetype. An archetype represents an entity type and every archetype has its own pool managing its memory space for the components. For example, if there is an entity type called Player, we can say there is an archetype Player and an archetype pool for Player. Archetype pools’ key role is managing memory and guaranteeing the entities from the pool located consecutively in the memory.

[1]In this context, the dataset denotes a set of data located in memory sequentially, waiting for some operations to be executed on it.

4.1 Defining Archetype

In ECS, an Entity is just an integer. In other words, an entity itself doesn't represent the type. Archetype represents the entity type.

```
EntityId playerId = ecs.CreateEntity<PlayerArchetype>();
EntityId enemyId = ecs.CreateEntity<EnemyArchetype>();
```

Listing 5: Entities are just integers.

In the code above, we have created two entities. One is created by the type PlayerArchetype, and the other by EnemyArchetype. The two EntityId values returned from the CreateEntity() function call are just integers, so, we cannot figure out what the entity type is with this value only.

Entity type is determined by archetype and archetype is a set of components. That is, the set of components determines the entity type. For example, player archetype can be defined as below.

```
#define PlayerArchetype LocationComponent, LifeComponent, InputComponent
```

Listing 6: PlayerArchetype

This means the player entity has LocationComponent, LifeComponent and InputComponent. In other words, if an entity has these components, it is a player entity.

And enemy archetype can be defined as below.

```
#define EnemyArchetype LocationComponent, LifeComponent
```

Listing 7: EnemyArchetype

This means the enemy entity has LocationComponent and LifeComponent. The difference between the two entities is that only player entity has InputComponent.

4.2 Archetype Pool

Now, we are going to implement the ArchetypePool. ArchetypePool helps allocating consecutive memory space for the components of the archetype.

Let's look at the case of PlayerArchetype. PlayerArchetypePool allocates memory for the components in the form like below.

| ———————————————— 16kb ——————————————— |
[LocationComponent 0...N] [LifeComponent 0...N] [InputComponent 0...N]

The pool allocates the memory for accommodating N number of Location-Components, LifeComponents and InputComponents.

What is the appropriate value for N? N should be determined considering the CPU cache size. L1 data cache has size of 32 kb in i7 CPU, so this implementation uses the half of it. If you set ‘N’ too high and there are many archetypes that instantiate only a few entities during the runtime, memory will be wasted by the ArchetypePools. If you set ‘N’ too low, you are not going to fully utilize the cache.

4.3 Chunk

If too many instances have been created for the archetype, the total size can be exceeded 16 kb. We solve this problem by introducing Chunks. Archetype-Pools create chunks that have size of 16 kb. When the archetype pool is initialized, only one chunk is created. However, If the chunk is full of instances, a new chunk is going to be created. Created chunks are managed in a form of linked-list like data structure.

Chunks are concealed inside of the ArchetypePool. No classes or codes outside of the ArchetypePool know about the existence of chunks. All operations related to the chunks are dealt by the ArchetypePool, so users don’t need to know about the complexity of memory processing steps.

Let’s start to implement the ArchetypePool and the Chunk.

First of all, we will define the interface of ArchetypePool like below,

```
class IArchetypePool
{
public:
    virtual EntityId CreateEntity() = 0;
    virtual bool RemoveEntity(EntityId entity) = 0;
};
```

Listing 8: ArchetypePool interface

and define the ArchetypePool class template.

```
template <typename ... ComponentTypes>
class ArchetypePool : public IArchetypePool
{
public:
    EntityId CreateEntity() override {}
    bool RemoveEntity(EntityId entity) override { return false; }
};
```

Listing 9: Definition of ArchetypePool

Chunk is defined in ArchetypePool as below.

```
 1 template <typename ... ComponentTypes>
 2 class ArchetypePool : public IArchetypePool
 3 {
 4     using Tuple = std::tuple<ComponentTypes...>;
 5     static constexpr uint32_t EntitySize = SizeOf<ComponentTypes...>::
           Value;
 6     static constexpr uint32_t EntityCountPerChunk =
 7         (ChunkSize - sizeof(void *) - sizeof(uint32_t)) / EntitySize;
 8     static constexpr uint32_t ComponentCount = sizeof...(ComponentTypes);
 9
10     struct Chunk
11     {
12         std::tuple<std::array<ComponentTypes, EntityCountPerChunk>...>
               Components;
13         constexpr static uint32_t InvalidIndex = -1;
14         uint32_t Count = 0;
15         Chunk *Next = nullptr;
16
17         void *operator new(std::size_t size)
18         {
19             return std::aligned_alloc(size, CacheLineSize);
20         }
21
22         void operator delete(void *p)
23         {
24             std::free(p);
25         }
26     };
27
28     Chunk *RootChunk;
29     /* omitted */
30 }
```

Listing 10: Definition of Chunk

In line 6, prior to defining the Chunk, the maximum number of entities a chunk can contain is calculated in a constant member field named EntityCountPerChunk.

In the above formula, the size of void* and uint32_t are subtracted from the originally planned ChunkSize 16 kb. void* is used to form a linked list when an additional chunk is created. uint32_t is used to indicate the valid number of entities in a chunk. The remaining size after the subtraction is used for accommodating actual entities.

In line 5, the size of entity is calculated by using struct template SizeOf<ComponentTypes...> since the entity size is just the sum of all components' sizes consisting the entity. SizeOf<> receives variadic template arguments and sums them all into Value constant member field.

Chunk defines the most important member field called Components in line 12. The type of the Components is std::tuple<std::array<ComponentTypes, EntityCountPerChunk>...>. For the EnemyArchetype which has LocationComponent and LifeComponent, the type of Components will be expanded as shown below assuming the EntityCountPerChunk is 100.

```
std::tuple<std::array<LocationComponent, 100>, std::array<LifeComponent,
    100>> Components
```

Listing 11: Definition of Chunk

The memory layout looks like below:
|——————————————16KB——————————————|
[LocationComponent 0...99][LifeComponent 0...99]
If you want to process the first enemy entity in the chunk, you need to access the first component in the first and the second array. Just to help your understanding, we could write the pseudo code accessing the first entity in Components as below.

```
ProcessEntity(std::get<0>(Components)[0], std::get<1>(Components)[0]);
```

Listing 12: Accessing the first entity.

4.4 Creating Entities

In this section, we are going to look into the process of entity creation. Let's define the interface class between ECS framework and the user, named DOECS. We are going to create entities through this interface class.

```
1  class DOECS
2  {
3   using PoolContainer = std::unordered_map<uint64_t, impl::IArchetypePool
        *>;
4   PoolContainer Pools;
5   std::unordered_map<EntityId, uint64_t> EntityPoolMap;
6
7   template<typename ... ComponentTypes>
8   EntityId CreateEntity(bool autoCreatePool = true);
9  }
```

Listing 13: DOECS interface class.

PoolContainer type alias in the line 3 is an associative container for the ArchetypePools' instances. The hash value of an ArchetypePool is represented as uint64_t and derived by combining all hash values of each component type. For example, the hash value of EnemyArchetypePool is:

Hash(ArchetypePool<LocationComponent, LifeComponent>) = Hash(LocationComponent) combine Hash(LifeComponent)

In line 5, EntityPoolMap is a kind of cache used to get the hash value of the ArchetypePool containing the entity.

To create an entity, call CreateEntity() function which is defined in line 8. This function template requires types of components consisting the entity as template arguments. If the function argument autoCreatePool is true and the ArchetypePool for the entity doesn't exist, the new ArchetypePool for the entity will be created automatically.

The following listing shows the implementation of CreateEntity() function.

```
 1  template <typename... ComponentTypes>
 2  EntityId DOECS::CreateEntity(bool autoCreatePool = true)
 3  {
 4      uint64_t poolHash = 0;
 5      impl::IArchetypePool *pool = nullptr;
 6      impl::ComponentsHash<0, ComponentTypes...>(poolHash);
 7
 8      auto it = Pools.find(poolHash);
 9      if (it == Pools.end())
10      {
11          if (autoCreatePool)
12          {
13              it = AddPool<ComponentTypes...>();
14              pool = it->second;
15          }
16      }
17      else
18      {
19          pool = it->second;
20      }
21      if (!pool)
22          return INVALID_ENTITY_ID;
23
24      auto entity = pool->CreateEntity();
25      if (entity != INVALID_ENTITY_ID)
26      {
27          EntityPoolMap[entity] = poolHash;
28      }
29      return entity;
30  }
```

Listing 14: CreateEntity function.

In line 6, we calculate the hash value of the entity archetype by calling ComponentHash() function. This function template gets types of components as template arguments, calculates hash values of each type and combines them to make the final result, the entity archetype's hash value. Codes in line 8 through 22 find ArchetypePool with the computed hash value. If the ArchetypePool doesn't exist and the argument autoCreatePool is true, a new ArchetypePool for the requested entity archetype is created in line 13. In line 24 , we call IArchetypePool::CreateEntity() function of the Archtype-Pool instance we have found or created to actually create an entity. This function deals with all aspects of entity creation in the pool and the detailed explanation will follow. In line 27, we map the newly created EntityId to the hash value of the ArchetypePool. With this mapping, we can quickly find which ArchetypePool contains the entity we are interested in.

Now, let's take a look at the process of creating an entity in the Archetype-Pool.

```
 1  template<typename ... ComponentTypes>
 2  EntityId ArchetypePool::CreateEntity() override
 3  {
 4      static_assert(EntityCountPerChunk > 50, "Entity␣is␣too␣big");
 5      auto chunk = RootChunk;
 6      while (chunk)
 7      {
 8          if (chunk->Count < EntityCountPerChunk)
 9          {
10              auto entity = GenerateEntityId();
11              auto componentIndex = chunk->Count++;
12              EntityToComponent[entity] = {chunk, componentIndex};
13              EntityIdsPerChunk[chunk][componentIndex] = entity;
14              return entity;
15          }
16          if (!chunk->Next)
17          {
18              // create new chunk
19              chunk->Next = new Chunk;
20          }
21          chunk = chunk->Next;
22      }
23      return INVALID_ENTITY_ID;
24  }
```

Listing 15: CreateEntity function.

The while statement starting from line 6 iterates forever until the chunk that can hold the new entity is found. If all chunks are full(line 16: if(!chunk->Next)), a new chunk will be created in line 19. The program eventually runs the code starting from line 10 if there's no lack of memory.

In line 10, a new entity id is generated by calling GenerateEntityId() function. In line 11, Chunk::Count indicating how many valid entities are in the chunk is increased by one. The maps in line 12 and 13 are kind of caches for the fast lookup. With the EntityId, EntityToComponent is used to figure out which chunk is containing the entity and what index is assigned to the entity. EntityIdsPerChunk caches all the entityIds in a chunk.

4.5 Removing Entities

The important thing in the entity removal process is that we need to fill the empty spaces made by removing entities in a chunk. The more empty spaces between valid entities are made, the lower cache hit rate we expect.

To remove an entity, call DOECS::RemoveEntity() function.

```cpp
bool DOECS::RemoveEntity(EntityId entity)
{
    auto pool = GetPoolForEntity(entity);
    if (!pool)
        return false;
    return pool->RemoveEntity(entity);
}
```

Listing 16: DOECS::RemoveEntity() function.

This function finds the pool in which the entity resides and call ArchetypePool::RemoveEntity() function again.

```cpp
 1  bool ArchetypePool::RemoveEntity(EntityId entity) override
 2  {
 3      void *chunk;
 4      uint32_t index;
 5      if (HasEntity(entity, chunk, index))
 6      {
 7          std::lock_guard l(Mutex);
 8          PendingRemove.insert(RemovingEntity{entity, (Chunk *)chunk, index
                });
 9          return true;
10      }
11      return false;
12  }
```

Listing 17: ArchetypePool::RemoveEntity() function.

ArchetypePool::RemoveEntity() function finds the chunk in which the entity resides and the index of the entity in line 5. In line 8, we keep the found data - chunk and index - into a queue called PendingRemove. We don't remove the entity right away, since the systems can be executed in a multi-threaded manner. After all systems are processed, user should call DOECS::Flush() function to actually remove the entities in the queue.

Flush function in DOECS interface looks like below. It is a simple forwarding function.

```cpp
void DOECS::Flush()
{
    for (auto &pool : Pools)
    {
        pool.second->Flush();
    }
}
```

Listing 18: DOECS::Flush() function.

Actual Implementation of Flush() function in ArchetypePool is shown below.

```
1  void ArchetypePool::Flush() override
2  {
3      std::vector<uint32_t> indexToRemove;
4      Chunk *lastChunk = nullptr;
5      for (size_t i = 0; i < PendingRemove.size(); ++i)
6      {
7          if (lastChunk == nullptr || lastChunk == PendingRemove[i].Chunk)
8          {
9              lastChunk = PendingRemove[i].Chunk;
10             indexToRemove.push_back(PendingRemove[i].Index);
11         }
12         else
13         {
14             if (!indexToRemove.empty())
15             {
16                 RecacheMovedEntities(lastChunk, lastChunk->RemoveEntities
                       (indexToRemove));
17                 indexToRemove.clear();
18             }
19             lastChunk = PendingRemove[i].Chunk;
20             indexToRemove.push_back(PendingRemove[i].Index);
21         }
22     }
23     if (!indexToRemove.empty())
24     {
25         RecacheMovedEntities(lastChunk, lastChunk->RemoveEntities(
               indexToRemove));
26     }
27     PendingRemove.clear();
28 }
```

Listing 19: ArchetypePool::Flush() function.

This function deletes entities in chunks. The ‘for’ statement in line 5 iterates for all the elements in PendingRemove vector. PendingRemove is a SortedVector in that all elements are grouped by chunk and sorted by index in ascending order. In line 10, we are gathering entity indices that will be removed into indexToRemove vector. This gathering process will continue until the next chunk group is found. Once the next chunk group is found, line 16 is executed.

In line 16, the two functions are called. One is Chunk::RemoveEntities() function and the other is ArchetypePool::RecacheMovedEntities() function. Let’s take a look one by one.

The following is the Chunk::RemoveEntities() function.

```
std::vector<MovedFromTo> Chunk::RemoveEntities(const std::vector<uint32_t
    > &entities)
{
    assert(!entities.empty());
    assert(entities.size() <= Count);
    std::vector<MovedFromTo> invalidatedIndex;
    for (uint32_t index : entities)
    {
```

```
 8          uint32_t lastValidIndex = FindLastValidIndexWhileRemoving(
                entities, invalidatedIndex);
 9          if (lastValidIndex != InvalidIndex)
10          {
11              Memcpy<0>(index, lastValidIndex);
12              invalidatedIndex.push_back({lastValidIndex, index});
13          }
14          --Count;
15      }
16      return invalidatedIndex;
17  }
```

Listing 20: Chunk::RemoveEntities() function.

This function deletes an entity(in line 14: –Count), and fills the vacant space by copying a valid entity residing in the most end of the chunk.

FindLastValidIndexWhileRemoving() function in line 8 has the role of finding the valid entity residing in the most end of the chunk. We are not going to look into this function in detail here, but the important consideration while selecting the valid entity is that:

- The entity must not be removed in the current flush i.e. the entity index must not be in the removing list.
- The entity has not been copied to the other location already.

ArchetypePool::RecacheMovedEntities() function has a role of updating EntityToComponent cache according to the entity movement that happened during the removal process. The implementation is shown below.

```
 1  void RecacheMovedEntities(Chunk *chunk, std::vector<MovedFromTo> &&
        movedEntities)
 2  {
 3      auto &entityIds = EntityIdsPerChunk[chunk];
 4      for (auto &it : movedEntities)
 5      {
 6          entityIds[it.second] = entityIds[it.first];
 7          EntityToComponent[entityIds[it.second]] = {chunk, it.second};
 8          EntityToComponent.erase(entityIds[it.first]);
 9          entityIds[it.first] = 0;
10      }
11  }
```

Listing 21: ArchetypePool::RecacheMovedEntities() function.

The vector ‘movedEntities’ has elements that are ‘MovedFromTo’ type. This type is defined as std::pair<uint32_t, uint32_t>. The first uint32_t indicates the index of the entity before the movement, and the second uint32_t indicates the index of the new position where the entity is moved to.

In line 3, the EntityId list for the chunk is aliased as entityIds. In line 4, the ‘for’ statement iterates all the moved entities, and line 6 actually moves the EntityId from the old index to the new index. In line 7 and 8, EntityToComponent cache is updated.

4.6 Getting Components of Entities

In some cases, we want to get a certain component of an entity with knowing the EntityId only. To support this, DOECS interface class has the following function.

```
1  template <typename ComponentType>
2  ComponentType *DOECS::GetComponent(EntityId entity)
3  {
4      auto pool = GetPoolForEntity(entity);
5      if (!pool)
6          return nullptr;
7      return (ComponentType *)pool->GetComponent(entity, typeid(
           ComponentType).hash_code());
8  }
```

Listing 22: DOECS::GetComponent() function.

You call this function with an EntityId. Then, this function finds the ArchetypePool containing the EntityId in line 4, and forwards the request to it in line 7. If you want to get LocationComponent of an entity id 5, you can call as below.

```
EntityId entity = 5; // created somewhere else.
LocationComponent* locComp = doecs.GetComponent<LocationComponent>(entity
    );
```

Listing 23: Getting a component from an entity.

ArchetypePool::GetComponent() function is defined as below.

```
void *ArchetypePool::GetComponent(EntityId entity, uint64_t componentHash
    ) override
{
    void *chunk;
    uint32_t index;
    if (HasEntity(entity, chunk, index))
    {
        return GetComponent(chunk, index, componentHash);
    }
    return nullptr;
}

void *ArchetypePool::GetComponent(void *chunk, uint32_t index, uint64_t
    componentHash)
{
    auto it = std::find(ComponentHashes.begin(), ComponentHashes.end(),
        componentHash);
    if (it == ComponentHashes.end())
16            return nullptr;
17
18        return ((Chunk *)chunk)->GetComponent((uint32_t)std::distance(
              ComponentHashes.begin(), it), index);
19    }
```

Listing 24: Getting a component from an entity.

There are two ArchetypePool::GetComponent() functions. The distinctions between them are the parameters. The parameters of ArchetypePool::GetComponent() function in line 1 are EntityId and ComponentHash. In contrast, the parameters of the function with the same name in line 12 are a chunk pointer, index and componentHash.

The first function has a role of forwarding the request to the second function after finding the ‘chunk’ containing the entity and the ‘index’ of the entity. The second function finds the component ordering index - note that this differs from the entity index - based on the parameter ‘componentHash’ and forwards the request again to the ‘chunk’ by calling Chunk::GetComponent() function with parameters of the component ordering index and the entity index.

The actual process of getting a component is done by Chunk::GetComponent() function. Let’s take a look in detail.

```
template <std::size_t I>
std::enable_if_t<I == sizeof...(ComponentTypes), void *> GetComponent(
    uint32_t componentTupleIndex, uint32_t entityIndex)
{
    return nullptr;
}

template <std::size_t I = 0>
    std::enable_if_t < I<sizeof...(ComponentTypes), void *> GetComponent(
        uint32_t componentTupleIndex, uint32_t entityIndex)
{
    if (I == componentTupleIndex)
    {
        return &std::get<I>(Components)[entityIndex];
    }
    else
    {
        return GetComponent<I + 1>(componentTupleIndex, entityIndex);
    }
}
```

Listing 25: Chunk::GetComponent() function.

Let’s start with the function parameters. The first parameter ‘componentTupleIndex’ is the component ordering index I have mentioned in the previous paragraph. This value represents where in the tuple the component we are looking for is located. For example, let’s assume we have the PlayerArchetype defined as below,

```
#define PlayerArchetype LocationComponent, LifeComponent, InputComponent
```

and we are looking for LifeComponent. The tuple Chunk::Components of ArchetypePool for the Player entities is going to be defined as below.

```
std::tuple<std::array<LocationComponent, EntityCountPerChunk>,
                  std::array<LifeComponent, EntityCountPerChunk>,
                  std::array<InputComponent, EntityCountPerChunk>>
                      Components;
```

LifeComponent is the second element in the tuple so the component ordering index (componentTupleIndex which is the first parameter) value is one(index starts from zero).

The second parameter entityIndex locates the component from the selected array, by the index of entity.

Chunk::GetComponent() is a function template. The template parameter is the integer I as you can see in line 1 and line 7. In line 10, if the integer I is the same value with componentTupleIndex, we get Ith element from the 'Components' tuple. This element is an array of the component we are looking for. The function is done by returning the component's address corresponding to the entityIndex from the component's array.

If the interger I and componentTupleIndex are different in line 10, we call the same function again with the template parameter I that has increased by one in line 16. This is a kind of recursion process. If I+1 exceeds the range of the tuple 'Components', Chunk::GetComponent() function in line 2 is executed rather than the function in line 8. This behavior is achieved by the 'std::enable_if' statement.

4.7 Systems

Systems are a logic that transforms component's data. A system can read from and/or write to multiple components. The following are examples of systems.

There could be

- a MovementSystem which reads from LocationComponent, MovementComponent and SightComponent, and writes to LocationComponent and MovementComponent.
- a SightSystem which reads from LocationComponent and SightComponent, and writes to SightComponent.

- a HumanAISystem which reads from HumanAIComponent, Location-Component, MovementComponent and SightComponent, and writes to several other components.

4.7.1 System Interface

Every system has the common interface defined as below.

```
1 class ISystem
2 {
3 public:
4     volatile bool Done = false;
5 
6     virtual std::size_t GetComponentHashes(const uint64_t *&pHashes) = 0;
7     virtual void Execute(uint32_t entityCount, const de2::ComponentsArg &
          components) = 0;
8 };
```

Listing 26: System interface

GetComponentHashes() function in line 6 is used to get all components' hash values which are necessary to run the System. The return value of the function is the number of the components and the argument 'pHashes' which is another output of the function will be filled with the address of the array containing the components' hash values.

Execute() function in line 7 needs to be overridden by the function executing the actual system logic. The first parameter 'entityCount' indicates how many entities can be processed in the call. In other words, it means how many valid components exist in one of the elements in 'components' parameter. The second parameter 'components' is de2::ComponentsArg type. This type is defined as std::vector<void*>. Each void* element is an array of components, which has the size of 'entityCount'. You need to explicitly cast void* to the matching component type. The order of component types is the same to the order of the component hash values returned by GetComponentHashes() function.

4.7.2 Creating a new System

In this section, we are going to assume that we need to create a MovementSystem which handles all movement of entities in the world. First of all, you need to define a new class derived from ISystem interface class as below.

```
class MovementSystem : public de2::ISystem
{
    virtual std::size_t GetComponentHashes(const uint64_t *&pHashes)
        override;
4       virtual void Execute(uint32_t entityCount, const de2::ComponentsArg &
            components) override;
5   };
```

Listing 27: MovementSystem

We are overriding GetComponentHashes() function in line 3 and Execute() function in line 4. Now, let's take a look into the implementation of these functions.

```
1   std::size_t MovementSystem::GetComponentHashes(const uint64_t *&pHashes)
2   {
3       static uint64_t ComponentHashes[] = {typeid(FLocationComponent).
            hash_code(), typeid(FMovementComponent).hash_code(),
4                                            typeid(FSightComponent).
                                                hash_code()};
5       pHashes = ComponentHashes;
6       return ARRAYCOUNT(ComponentHashes);
7   }
```

Listing 28: MovementSystem::GetComponentHashes() function.

We know that the MovementSystem requires LocationComponent, MovementComponent and SightComponent when it runs and transforms data. This information is static, so in line 3 ComponentHashes array is defined as a static data which contains those three components' hash values. The function returns the number of components related to this system in line 6 and also fills the output parameter pHashes with the components' hash values in line 5.

The following listing demonstrates the Execute() function for the MovementSystem.

```
1   void FMovementSystem::Execute(uint32_t entityCount, const de2::
        ComponentsArg &components)
2   {
3       FLocationComponent *locComps = (FLocationComponent *)components[0];
            // read & write
4       FMovementComponent *moveComps = (FMovementComponent *)components[1];
            // read & write
5       FSightComponent *sightComps = (FSightComponent *)components[2];
            // used after systems finished.
6
7       PathFindingRequestMap PathFindingRequestPerWorld;
8       for (uint32_t i = 0; i < entityCount; ++i)
9       {
10          auto &locComp = locComps[i];
11          auto &moveComp = moveComps[i];
12
13          if (moveComp.TargetFragment == nullptr)
14          {
15              continue;
16          }
17
18          // when arrived.
19          if (IsSameLocation(locComp, moveComp))
20          {
21              moveComp.TargetFragment = nullptr;
22              continue;
23          }
24
25          // When path is invalid.
26          if (!moveComp.ValidPathCount || moveComp.NextPathIndex >=
                FMovementComponent::MOVE_MAX_PATH)
27          {
28              AddPathFindingRequest(PathFindingRequestPerWorld, &locComp, &
                    moveComp, &sightComps[i]);
29              continue;
30          }
31
32          // Move
33          locComp.CellCoord = moveComp.Path[moveComp.NextPathIndex++];
34      }
35
36      for (auto it : PathFindingRequestPerWorld)
37      {
38          auto world = it.first;
39          world->QueueFindPaths(it.second);
40      }
41  }
```

Listing 29: Execution() function for the Movement System

From line 3 to 5, we are explicitly casting the components array to the type that is already known and required to process the system. The elements in the component array have the same order with the hash values returned by GetComponentHashes() function. The code in line 8 to 34 iterates all entities and processes the movement logic.

4.7.3 Running the Systems

Each System is processed as a task. Tasks are registered to TaskScheduler and executed in an arbitrary worker thread. To register a task to the TaskScheduler, call the following function

```
void FTaskSchedular::AddTask(FTask *task, ETaskPriority priority)
```

There could be dependencies between two Systems. For example, Sight System needs to be processed completely before the Movement System. To achieve this, we need a concept of dependency between tasks. Task Scheduler library we are going to use meets this requirement. The library name is FBTaskSchedular and you can download it from Github.

To wrap the execution of a system as a task, you need to create a new Task class as below. This example is for SightSystem.

```
class FSightSystemTask : public fb::FTask
{
    de2::DOECS *ECS;

public:
    FSightSystemTask(de2::DOECS *ecs)
        : ECS(ecs)
    {
        assert(ecs);
    }
    virtual void Run() override
    {
        assert(gSightSystem);
        ECS->RunSystem(gSightSystem);
        NotifyFinish();
    }

    virtual bool CanRun() const override
    {
        return true;
    }
};
```

Listing 30: Definition of FSightSystemTask.

Run() function will be called when the task is started. Before the task is executed, CanRun() function is called. If this function returns false, the task will be postponed until it returns true. This mechanism is for dependency. FSightSystemTask doesn't have any dependency to other systems so the function always returns true. Let's take a look at FMovementSystemTask. This task is dependent on FSightSystemTask.

```
class FMovementSystemTask : public fb::FTask
{
    de2::DOECS *ECS;
    // Dependencies : AISystem
    static constexpr int DependenciesCount = 1;
    volatile int CurDependenciesCount = DependenciesCount;

public:
    FMovementSystemTask(de2::DOECS *ecs)
        : ECS(ecs)
    {
        assert(ecs);
    }

    virtual void Run() override
    {
        assert(gMovementSystem);
        ECS->RunSystem(gMovementSystem);
        CurDependenciesCount = DependenciesCount;
        NotifyFinish();
    }

    virtual bool CanRun() const override
    {
        return CurDependenciesCount == 0;
26      }
27
28      virtual void OnTaskFinish(FTask *task)
29      {
30          --CurDependenciesCount;
31          assert(CurDependenciesCount >= 0);
32      }
33  };
```

Listing 31: Definition of FMovementSystemTask.

For further discussion, we also need to check the class Task and the interface ITaskListener as well.

```
 1  class FTask;
 2  class ITaskListener
 3  {
 4  public:
 5      virtual void OnTaskFinish(FTask *task) {}
 6  };
 7
 8  class FTask : public ITaskListener
 9  {
10  public:
11      std::vector<ITaskListener *> TaskListeners;
12      bool AutoDelete = false;
13
14      virtual void Run() = 0;
15      virtual bool CanRun() const = 0;
16      void NotifyFinish()
17      {
18          for (auto listener : TaskListeners)
19          {
20              listener->OnTaskFinish(this);
21          }
22      }
23  };
```

Listing 32: Task and ITaskListener.

In line 11, ‘TaskListeners’ member field will hold ITaskListener pointers that are used to notify whenever the status of the task is changed(currently only for listening to ‘finish’ event). FTask::NotifyFinish() function in line 16 will notify the listeners of the ‘finish’ event by calling ITaskLIstener::OnTaskFinish() function.

To set a dependency on FMovementSystemTask to FSightSystemTask, you need to set FMovementSystemTask::CurDependenciesCount to value of one and add FMovementSystemTask into FSightSystemTask::TaskListeners member field. When FSightSystemTask is completed, FMovementSystemTask will be notified via FMovementSystemTask::OnTaskFinish() function. In this function, FMovementSystemTask::CurDependenciesCount will be decreased by the value of one as shown in line 30, listing 31. After the CurDependenciesCount variable becomes zero, FMovementSystemTask::CanRun()

function will return true and the TaskScheduler will start the FMovementSystemTask once any worker thread becomes available.

5 Further Researches

If Systems are executed in several threads at the same time, we cannot avoid data race condition. To prevent this, we can introduce dependencies between Systems that simultaneously access the same data. Dependencies guarantee that related systems are not run at the same time. However, this method will drop CPU utilization. Another method is introducing Mutex per component or per variable. Mutex state changes also use CPU power, but there is a higher chance of mitigation by fully utilizing the CPU running as many Systems as possible in parallel. The third method is that we don't change any data while Systems are processed but just record the data change requests. After all Systems are processed, we can actually perform the data change in a single thread according to the requests recorded. In this method, the number of data change requests made per tick will affect the performance.

Another aspect we should consider is that the interaction between entity system and other parts of the program can harm the CPU cache hit rate. For example, the MovementSystem can ask the World to find a valid path leading to the target location. World data is not considered to fit into the CPU cache along with the entity data. There is a high chance of CPU cache miss if we access the world data while the entities' component data are fully loaded in the CPU cache. I cannot say this is deterministic and predictable. If we access a small portion of World data and the data fetch prediction algorithm existing on the CPU works well somehow, CPU cache miss won't happen or at least doesn't harm the performance. In a case where accessing World data harms the CPU cache hit rate, we can avoid it by introducing postponed bulk processing step for the interaction requests - in this case, the path finding requests -.

6 References

- Mike Acton, Data-oriented Design and C++, CppCon 2014
  https://youtu.be/rX0ItVEVjHc

- Stoyan Nikolov, OOP is Dead, Long Live Data-oriented Design, CppCon 2018
  https://youtu.be/yy8jQgmhbAU

- Wikipedia, CPU cache
  https://en.wikipedia.org/wiki/CPU_cache