Swinburne University, COS30031 – Games Programming

Custom Project Documents

Andre Villalon (103056831)

Contents

Section-ILO Alignment Summary 2
Online Git Repository 3
Video Demonstration 3
Game Design 4
  Annotated Explanation of UI 4
  Gameplay and Game rules 4
  Ship & Bullet Details 5
  Level Creation using JSON Files 6
Use of Third-Party Libraries 7
  SDL2 (https://www.libsdl.org/) 7
  Nlohmann’s Json.hpp (https://github.com/nlohmann/json) 7
  Sgorsten’s linalg.h (https://github.com/sgorsten/linalg) 7
Class Descriptions 8
  Architectural/Abstract Classes 8
  Concrete Component Classes 14
  Concrete System Classes 15
UML Diagrams 16
  Architectural Class Diagram 16
  Sequence Diagram 18
Commit History Details 19
Screenshots/Appendices 22

Section-ILO Alignment Summary

The table below indicates what ILOs the sections/sub-sections throughout this document best represents. The ILOs of the Games Programming unit are as follows:

- **ILO 1 - Design:** *"Discuss game engine components including architectures of components, selection of components for a particular game specification the role and purpose of specific game engine components, and the relationship of components with underlying technologies."*
- **ILO 2 - Implementation: "***Create games that utilise and demonstrate game engine component functionality, including the implementation of components that encapsulate specific low-level APIs."*
- **ILO 3 - Maintenance:** *"Explain and illustrate the role of data structures and patterns in game programming, and rationalise the selection of these for the development of a specified game scenario."*
- **ILO 4 - Performance: "***Identify performance bottlenecks by using profiling techniques and tools, and applying optimisation strategies to improve performance."*

Section/Sub-section	**ILO 1**	**ILO 2**	**ILO 3**	**ILO 4**
Game Design	✓	✓	✓	
- *Other sub-sections*	✓			
- Level Creation using JSON Files		✓	✓	
Use of Third-Party Libraries		✓	✓	
Class Descriptions	✓	✓	✓	✓
- Architectural/Abstract Classes	✓	✓	✓	✓
- Concrete Component Classes	✓	✓		
- Concrete System Classes	✓	✓		
UML Diagrams	✓	✓	✓	
- Architectural Class Diagram	✓		✓	
- Sequence Diagram	✓	✓		
Commit History Details		✓	✓	✓

Online Git Repository

GitHub Repository Link: https://github.com/Andre-V/Andre-Shmup

Test Cases: https://github.com/Andre-V/Andre-Shmup/tree/master/ShmupTests

Textures: https://github.com/Andre-V/Andre-Shmup/tree/master/textures

Level File: https://github.com/Andre-V/Andre-Shmup/tree/master/levels

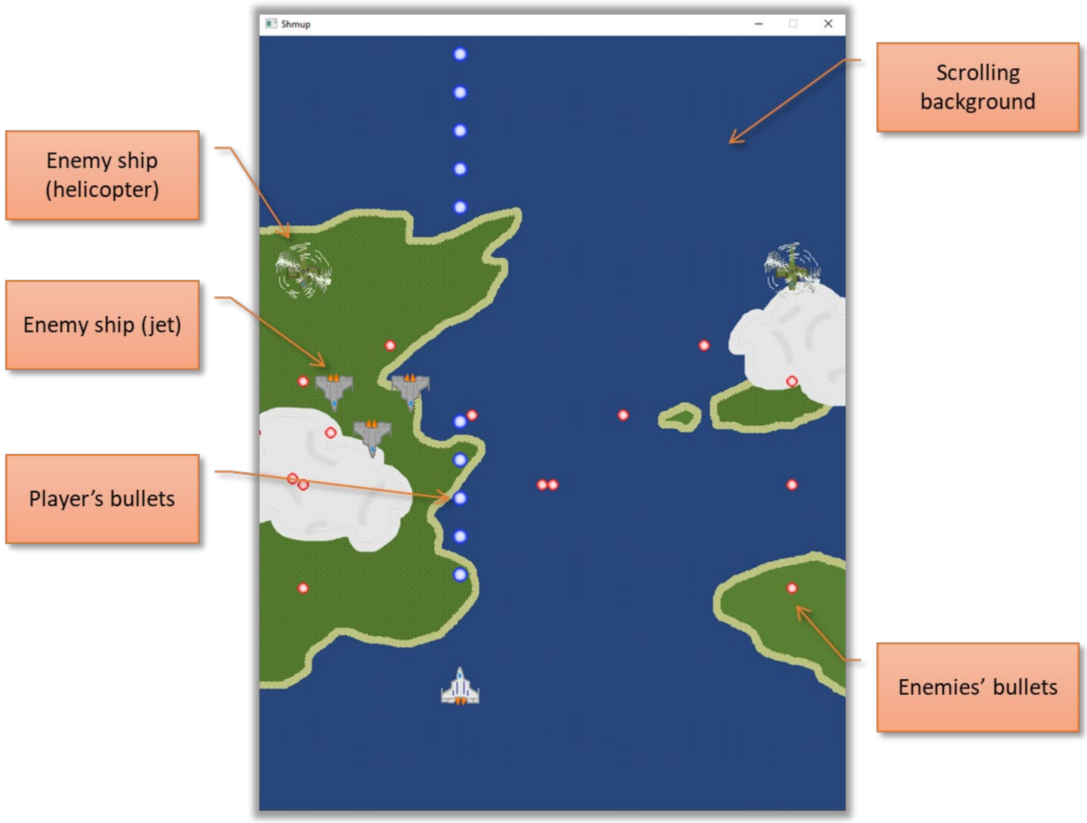
The repository includes all code, documents as well as other relevant files such as textures (stored in their respective folder).

Video Demonstration

Video Link: https://liveswinburneeduau-my.sharepoint.com/:v:/g/personal/103056831_student_swin_edu_au/EaJxKt3oc9lMuopHQBG3fJcBjp4hX0_1jtJ-cXahGim22Q?e=VgIREU

Game Design

Annotated Explanation of UI



Gameplay and Game rules

Andre's Shmup is a **Sh**oot 'e**m up** game where the player is tasked with controlling a single ship across a single level. The player's goal is to reach the end of the level, indicated in this game by enemies no longer spawning.

Enemies spawn throughout the level to prevent the player from reaching the end and can shoot bullets to destroy the player's ship. These enemies come in different varieties and behaviour. Some ships may be stronger, faster, or smarter than others.

The player's ship can be moved within the game area using the WASD or arrow keys. The player can press the spacebar key to shoot bullets to destroy enemy ships.

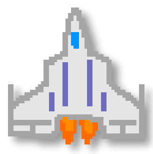
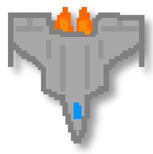
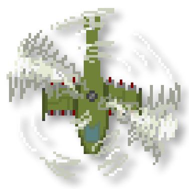
The player's ship is instantly destroyed if it collides with either an enemies' bullet or another ship. Enemy ships, upon collision with a player's bullet, take damage that reduces its health value. They are destroyed if their health reaches 0 or lower. They are not destroyed if they collide with the player's ship.

The level is defined using a text file, which controls the background and foreground texture, and the enemies spawned. A user may want to edit it for their own purposes.

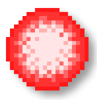
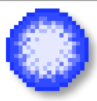
Ship & Bullet Details

The following tables list the ships as well as the bullets that exist within the game. Within the code, they share similar components – all being built off of the "Ship" factory method. Images can be found within the game's "textures" folder. The names used may be shortened for classes and factory methods.

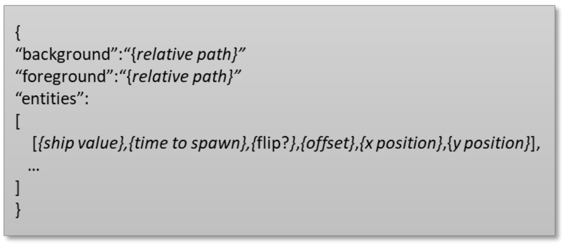
Ships:

Name	**Level Editor Value**	**Health**	**Relative Speed**	**Description**	**Image**
Player	N/A (Always spawned)	1 life	Moderate	Controlled by the player using WASD and arrow keys. Shoot when the spacebar is pressed.	
Jet	0	5 HP	Fast	Moves downwards from the top for a set duration, followed by shooting a single bullet downwards alongside movement diagonally left or right.	
Helicopter	1	10 HP	Slow	Moves downwards from the top for a set duration. Then, alternates between moving left and right, while stopping and shooting in between.	
Stealth Bomber	2	15 Hp	Moderate	Comes quickly upwards from the bottom of the screen, followed by slowly moving in front of the player so it can shoot.	

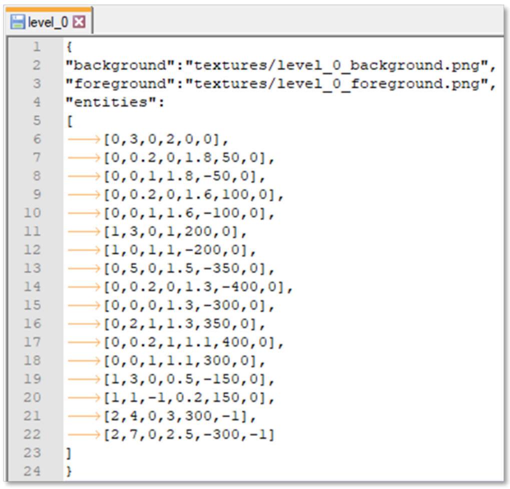
Bullets:

Description	**Image**
Bullet shot by an enemy ship.	
Bullet shot by the player. Deals 1 HP of damage to enemy ships.	

Level Creation using JSON Files



```
{
"background":"{relative path}"
"foreground":"{relative path}"
"entities":
[
   [{ship value},{time to spawn},{flip?},{offset},{x position},{y position}],
   …
]
}
```



```
{
"background":"textures/level_0_background.png",
"foreground":"textures/level_0_foreground.png",
"entities":
[
	[0,3,0,2,0,0],
	[0,0.2,0,1.8,50,0],
	[0,0,1,1.8,-50,0],
	[0,0.2,0,1.6,100,0],
	[0,0,1,1.6,-100,0],
	[1,3,0,1,200,0],
	[1,0,1,1,-200,0],
	[0,5,0,1.5,-350,0],
	[0,0.2,0,1.3,-400,0],
	[0,0,0,1.3,-300,0],
	[0,2,1,1.3,350,0],
	[0,0.2,1,1.1,400,0],
	[0,0,1,1.1,300,0],
	[1,3,0,0.5,-150,0],
	[1,1,-1,0.2,150,0],
	[2,4,0,3,300,-1],
	[2,7,0,2.5,-300,-1]
]
}
```

(Left) Overview of how a level file is structured. (Right) The actual implementation of the tile used for the game.

Level creation is defined in the JSON file titled "level_0" in the "levels" folder. Incorrect values or formats will most likely result in the game crashing. Reading such files is implemented within "FileLoadFactory.h" and "FileLoadFactory.cpp" files. Its implementation primarily uses C++ standard library's file I/O API and nlohmann's JSON library.

The 'background' and 'foreground' keys are given a string value specifying the relative path to the texture file. The 'entities' key is given a 2D array specifying the list of enemy ships to be spawned. Each array within the array is defined as follows (left-to-right):

- 0 – {ship value}: a value from 0 to 2 (see *Ship Details*) corresponding to a type of ship.
- 1 – {time to spawn}: a positive decimal value specifying the time in seconds for a ship to spawn since the last ship spawned. (A value of 0 means a ship is spawned right after another).
- 2 – {flip?}: a value of either 0 or 1 (converted to bool in code), specifies whether a ship is initially flipped. Depending on the system that implements a ship's logic, having this value set to true results in a ship moving in the opposite direction.
- 3 – {offset}: a positive decimal value representing the time in seconds. Value is used as a timer to determine when a ship changes its initial state (default value is 100 ticks, a bit more than one second).
- 4 – {x position}: a value in pixels for the x position of the ship when it spawns. A value of 0 is the central point.
- 5 – {y position}: a value in pixels for the y position of the ship when it spawns. A value of 0 is the central point. For convenience, setting a value of -1 sets the ship's position to the bottom of the screen, so the screen height does not have to be known. Setting a value of -1 or 0 moves the initial position of a ship such that the ship is not seen spawning in, which is done by accounting for its dimensions.

While there is only a single level, with no architecture and systems in place for loading more, the creation of one using a file is considerably flexible and maintainable. Other programmers (and regular users) do not have to consider parts of the code and changes can be made after the code has been compiled. If additional levels were to be implemented, this file loading system would make development faster and more convenient.

Use of Third-Party Libraries

SDL2 (https://www.libsdl.org/)

The SDL2 library was utilized to quickly implement graphical rendering of textures in a window, the handling of keyboard events and presses, and rectangle shapes (SDL_Rect) for AABB collision detection (using SDL_HasIntersection). Without this library, there would be no game visible!

Nlohmann's Json.hpp (https://github.com/nlohmann/json)

Json.hpp was used to read JSON files in such a way that files loaded into a JSON object could be treated like a Standard Library collection object. Such an API means interacting with this object can be done reliably and intuitively while avoiding having to write my own implementation of parsing a file, which may change over time.

Sgorsten's linalg.h (https://github.com/sgorsten/linalg)

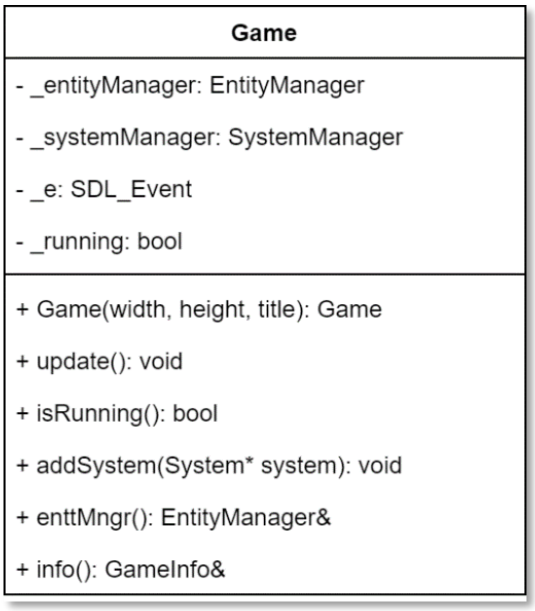
Linalg.h was used for its 2D vectors so I didn't have to deal with implementing the mathematics of vectors and spending additional effort testing for correctness. I didn't make use of complicated vector operations and functions though. The most complicated would have to be vector rotation and the rest is just vector addition. Later on, more complex mathematical operations may have to be used and this library should help speed up development time.

Throughout this document, especially within the *Class Descriptions* section, the 'float2' data type may be encountered. This data type is an alias for a vector of type float with 2 elements/dimensions, accessible via indexing or the 'x' and 'y' properties.

Class Descriptions

Architectural/Abstract Classes

Game


Game
- _entityManager: EntityManager
- _systemManager: SystemManager
- _e: SDL_Event
- _running: bool
+ Game(width, height, title): Game
+ update(): void
+ isRunning(): bool
+ addSystem(System* system): void
+ enttMngr(): EntityManager&
+ info(): GameInfo&

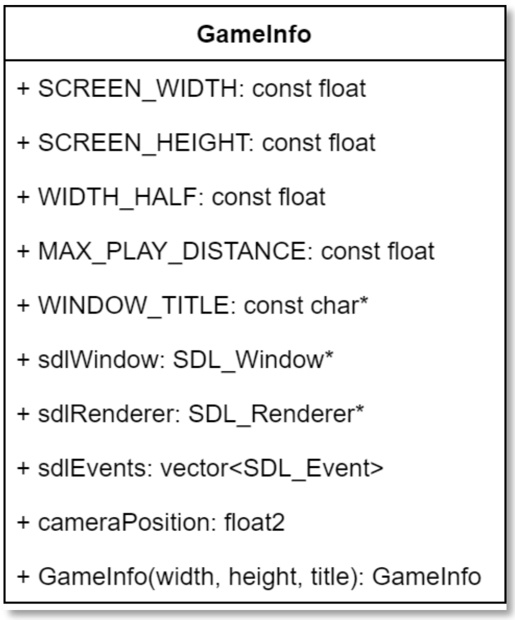
Responsible for game-related events, such as updating the game loop and retrieving SDL key events, and information. Its responsibilities of updating the game loop are delegated to the respective manager classes.

Initialization of a game instance configures the SDL Window and Renderer. Only the screen's width, height and title are needed as parameters.

Systems can be added using a game instance. The instance assigns some of its information to the system (itself as GameInfo and its EntityManager) before it is added to its SystemManager.

The game class could've very well been made to utilise the singleton pattern (as well as many other management classes) since only one instance is ever really needed. In my experience, it has made managing relationships easier, as it's very clear what an instance is responsible for and who's responsible for it.

GameInfo


GameInfo
+ SCREEN_WIDTH: const float
+ SCREEN_HEIGHT: const float
+ WIDTH_HALF: const float
+ MAX_PLAY_DISTANCE: const float
+ WINDOW_TITLE: const char*
+ sdlWindow: SDL_Window*
+ sdlRenderer: SDL_Renderer*
+ sdlEvents: vector<SDL_Event>
+ cameraPosition: float2
+ GameInfo(width, height, title): GameInfo

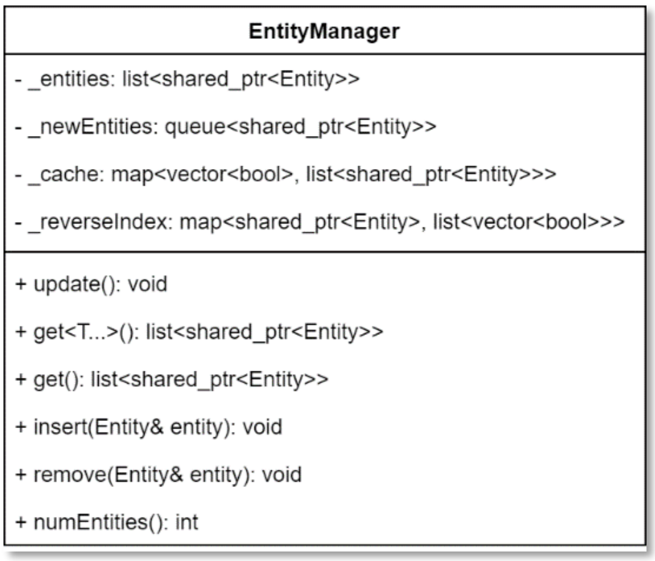
Contains game-related information and is publicly inherited by the Game class.

Its purpose of being separate from the Game class is for encapsulation and ease of maintenance. Other classes rely on game information (most importantly the System class) but should never access a Game's other public methods; the programmer shouldn't need to update the game loop or add another system within a system – this is done outside!

Most of these properties are also constant since these properties were set upon the initial configuration of the screen and having them change would result in potentially illogical behaviour.

MAX_PLAY_DISTANCE indicates the maximum distance the player's ship may travel from the centre of the playable area. This is different from SCREEN_WIDTH as the player's ship can travel a bit more than the width allows. Its also used by the camera system, along with the cameraPosition property, to move the camera in a specific way.

EntityManager


EntityManager
- _entities: list<shared_ptr<Entity>>
- _newEntities: queue<shared_ptr<Entity>>
- _cache: map<vector<bool>, list<shared_ptr<Entity>>>
- _reverseIndex: map<shared_ptr<Entity>, list<vector<bool>>>
+ update(): void
+ get<T...>(): list<shared_ptr<Entity>>
+ get(): list<shared_ptr<Entity>>
+ insert(Entity& entity): void
+ remove(Entity& entity): void
+ numEntities(): int

Responsible for storing all entities in a game, facilitating the insertion, removal and retrieval of entities.

Upon being updated, inactive entities are removed from every collection and new entities are added to the collections. This maintains the logical order of the systems using the newly inserted entity.

Retrieval of entities involves querying what components are desired. The entities retrieved must have these components. They may have other components which could be accessed but run the risk of crashing the program should an instance not have one. So, it's best to query everything that is needed.

Such queries are done using variadic templates to potentially speed up performance. Alternatives include passing in a Boolean array as a key or a list of strings. Using strings would come at a performance cost and have the additional overhead of mapping strings to the respecting components. Using a Boolean array should be efficient as well but is a lot less readable, making maintenance of existing queries harder, and the entire array has to be defined in the correct order, which would be very annoying to deal with. It should be noted that this "key" of components varies between entities varies in size as newly encountered components are added, so there's additional stuff to deal with there too. So not only is the use of templates faster, but also human-readable where only the components needed have to be specified, and in any order.

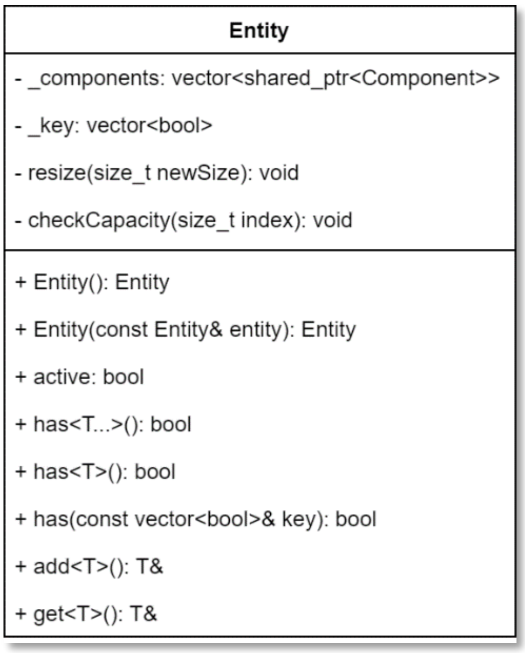
Such queries would normally involve looping through every entity and checking for a match – not performance efficient. This method was primarily used throughout most of the development. No lag was noticeable yet, but it was decided to sacrifice memory for better performance. While it did not result in any noticeable difference, it may speed things up if the same architecture was ever reused for games containing a lot more entities.

The cache and reverseIndex properties achieve this. The cache stores the results of previous queries and is updated in case entities matching these queries are added or removed. The reverseIndex stores each entity's queries that may retrieve it. It is purpose comes into fruition when entities are removed by no longer having to go through every cache query to find the entity to be removed. The performance could've been improved further by storing its iterator position within the list as the list still has to be searched entirely.

Shared pointers are used instead of pointers by themselves to simplify the management and deletion of entities, which occurs when there are 0 instances of pointers to an entity.

The class currently does not handle entities having their components change through addition and removal due to time constraints and since components are currently no longer being added or removed once they are inserted into the EntityManager. Should this be implemented, an efficient method would be to be let the EntityManager know exactly what entity changed from a System instance instead of having to keep track of what components entities used to have.

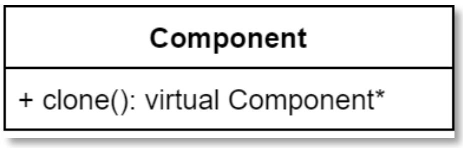
Entity


Entity
- _components: vector<shared_ptr<Component>>
- _key: vector<bool>
- resize(size_t newSize): void
- checkCapacity(size_t index): void
+ Entity(): Entity
+ Entity(const Entity& entity): Entity
+ active: bool
+ has<T...>(): bool
+ has<T>(): bool
+ has(const vector<bool>& key): bool
+ add<T>(): T&
+ get<T>(): T&

The entity is responsible for storing related components, whatever they may represent, and the retrieval and addition of new components. Components are stored in a vector array. A Boolean array, the key, tracks what components exist in an entity.

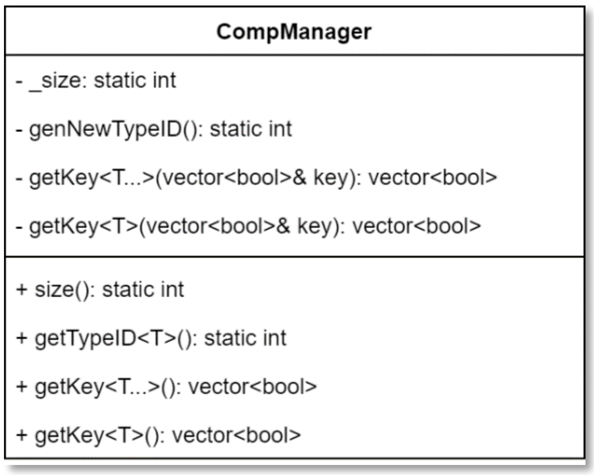
Retrieval and addition of components are implemented using templates and is done in such as way that the programmer never has to keep track of what index position a component should be stored in (*see CompManager*). Neither do they have to be concerned about how many components there are as the vector's size dynamically increases to fit more components. This makes maintenance a lot easier as nothing has to be done if a new component class is written.

Component



Concrete implementations of the component class simply store data used by System classes. A virtual clone method does exist so information can be properly copied. Otherwise, bugs were encountered where the copied component had null or illogical values.

CompManager

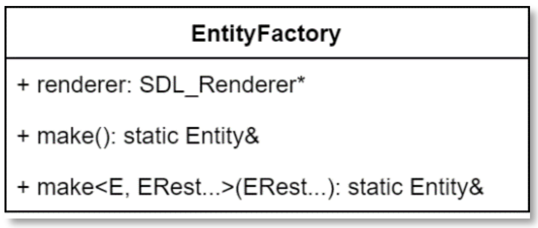

CompManager
- _size: static int
- genNewTypeID(): static int
- getKey<T...>(vector<bool>& key): vector<bool>
- getKey<T>(vector<bool>& key): vector<bool>
+ size(): static int
+ getTypeID<T>(): static int
+ getKey<T...>(): vector<bool>
+ getKey<T>(): vector<bool>

Responsible for keeping track of the index values that correspond to each component stored in an entity. Its implementation relies on templates and static integers within the functions themselves. Upon calling getTypeID with a newly encountered class name, genNewTypeID is called to generate a new index value for that class, which is done by incrementing the _size property. Subsequent calls retrieve this generated value. This avoids the issue of having to manually track the index position of a component in entities, greatly improving maintenance.

The key-related methods generate a Boolean array, essentially converting the variadic template parameters to an array. This is used by an EntityManager for its _cache's keys since templates can't be stored like variables.

Due to the nature of templates, they cannot be treated like variables and the algorithms involving them make use of recursion, calling the same or different function but with fewer template arguments passed until only one template argument is encountered.

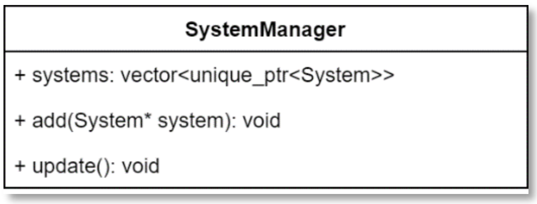
EntityFactory


EntityFactory
+ renderer: SDL_Renderer*
+ make(): static Entity&
+ make<E, ERest...>(ERest...): static Entity&

Responsible for creation and configuration of entities and its components, improving the maintainability of the code by preventing code reuse and allowing changes to a specific kind of entity to be done at a single location. The creation of an entity type involves creating a template specialisation for the make<E, ERest…>(ERest…) function, specifying how the entity is created and the arguments needed. Otherwise, an entity with no components will be created. Implementing template specialisation isn't necessary, as separate, differently named functions could've just been made so it was just done for aesthetic/readability purposes, making it consistent with the use of template arguments for other classes.

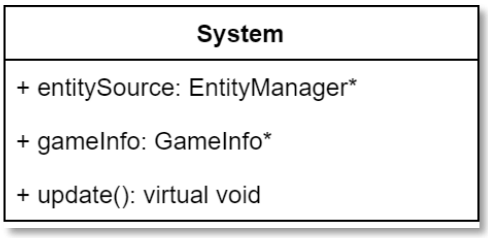
Since the factory adds texture components to entities, it needs a renderer to assign to textures.

SystemManager


SystemManager
+ systems: vector<unique_ptr<System>>
+ add(System* system): void
+ update(): void

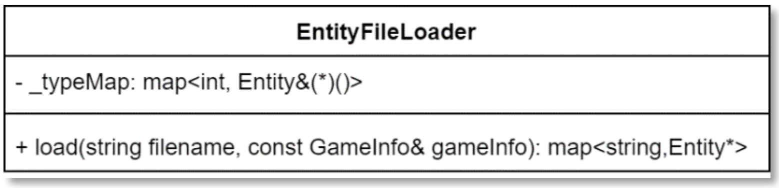
Responsible for updating the list of systems it contains. A sorted set was initially used to store systems, with systems having a priority value, so they're processed in the correct order. However, tracking the specified order meant additional effort and it was ultimately unnecessary since the proper order can be given by just adding systems in the correct order. The order should never change anyways.

System



Implementations contain the logic to be applied to entities and their sets of components within the update function called once per game loop. The entitySource property allows the use of the EntityManager for its responsibilities and the gameInfo property allows access to game information.

EntityFileLoader


EntityFileLoader
- _typeMap: map<int, Entity&(*)()>
+ load(string filename, const GameInfo& gameInfo): map<string,Entity*>

Parses a level file (*see Level Creation using JSON Files*) and contains the mappings of an integer value to a function pointer to create an entity. The mappings are set in its .cpp file and the function pointers come from the EntityFactory specialisations. It returns a map of entities to be added to the game, such as a spawner entity containing the enemy ships to be spawned and the entities for the background and foreground.

Concrete Component Classes

When creating these components, a general principle that was followed was to break down related data into the smallest sets of components so that entities don't contain redundant information and so that there aren't components with overlapping data. That may entail the creation of components with at most a single property, but some properties don't make sense by themselves (e.g., a component with a width property should have a height property).

In addition to the components depicted in the table below, other components exist simply for identification to facilitate the separation of behaviour or to conveniently reduce queries into a single type. Hence, these don't contain any data. The following components are Player, Jet, Heli, Stealth, EnemysBullet, PlayersBullet and Background.

Component Name, Properties	**Description**
Health • health : float	Stores an entity's health.
Position • position : float2	Stores the position of an entity.
Velocity • velocity : float2	Stores the velocity of an entity.
Dimensions • w : float • h : float	Stores the width and height of an entity.
TextureBox • texture : Texture	Stores the texture (class containing SDL_Texture) of an entity.
Spawner • sequence : vector<pair<Entity*, int>> • offsets : vector<float2> • index : int • ticks : int • active : bool • loop : bool	Stores a spawner with properties configuring what, when and how entities are spawned. The sequence property specifies what entities are spawned and how soon after the last entity. Offsets are meant to adjust the location of a spawned entity. The index property indicates the current entity to spawn in a vector. The ticks property serves as a timer to track what to spawn. The active property indicates if a spawner should start spawning. The loop property indicates if a spawner should restart the index once it's gone through each item in the sequence. These properties are managed using a dedicated system.
Ship • shoot : false	Currently just indicates if a ship can shoot.
AIShip • state : int • stateOffset : int • flipped : bool	Stores information used by a ship to behave as an agent.
Bullet • damage : int	Stores the damage to be dealt to an entity with health.

Concrete System Classes

The table below indicates the systems used for the game in order of how they are processed. The logic is split into systems to be as reasonably small as possible, such that they make maintenance easy by making each system simple and manageable by a single person and so that they can be left alone when more logic is being added.

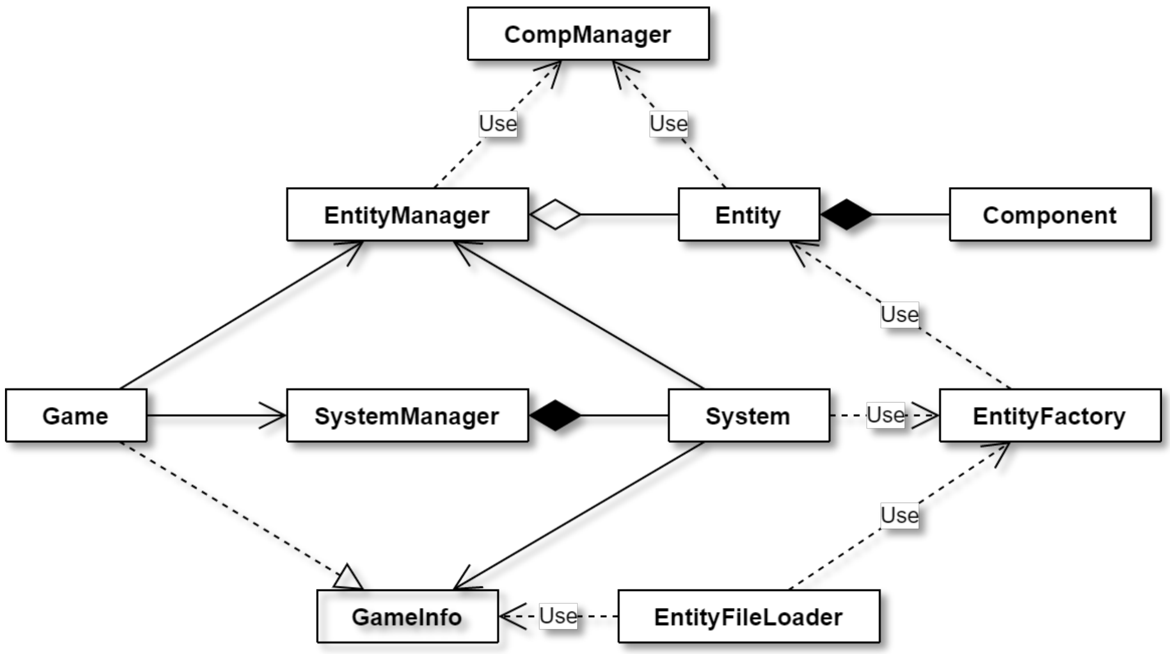
The AI-related systems use a basic implementation of the state pattern using the AIShip component's state property and a switch. An alternative, more cohesive, but more difficult to track method, would be to add and remove components reflecting its state. Hence, such systems would only act on entities with that state and only contain the logic for that state. That approach would make a lot more sense if the AI was much more complex/intelligent.

System Name	Queries	Description
SysPlayerInput	• Player	Takes keyboards inputs to translate into movement.
SysJetAI	• Jet	The behaviour of the Jet ship.
SysHeliAI	• Heli	The behaviour of the Heli ship.
SysStealthAI	• Stealth	The behaviour of the Stealth Bomber ship.
SysMove	• Position, Velocity	Changes the position of an entity according to its current velocity.
SysMoveCamera	• Player	Adjusts the camera's position according to the location of the player.
SysRenderBackground	• Background, TextureBox, Position, Dimensions	Renders the background and adjusts its position such that it scrolls forever. As there already exists a rendering system, this system renders a second texture if the first image does not fill the screen.
SysRender	• TextureBox, Position, Dimensions	Renders the textures of an entity according to its location and size.
SysUpdateSpawners	• Spawner	Handles the logic for the spawner entities.
SysShipShoot	• Ship, Spawner	Activates spawners if ship.shoot is true.
SysHitEnemyCollisions	• Bullet, PlayersBullet, Position, Dimensions • Ship, Health, Position, Dimensions	Handles AABB collision between ships and bullets shot by the player. Reduces the health of ships that do collide.
SysPlayerCollisions	• Player • Bullet, EnemysBullet, Position • AIShip, Position	Handles AABB collision between the player's ship, other ships and bullets shot by the enemy. Deletes the player's ship upon collision.
SysDestroyNoHealth	• Health	Sets entity.active to false if health is equal to or less than 0.
SysDestroyOutOfBounds	• Bullet, Position • Ship, Position	Sets entity.active to false if the position of bullets or ships is outside the specified range (a couple of hundred pixels from the play area).

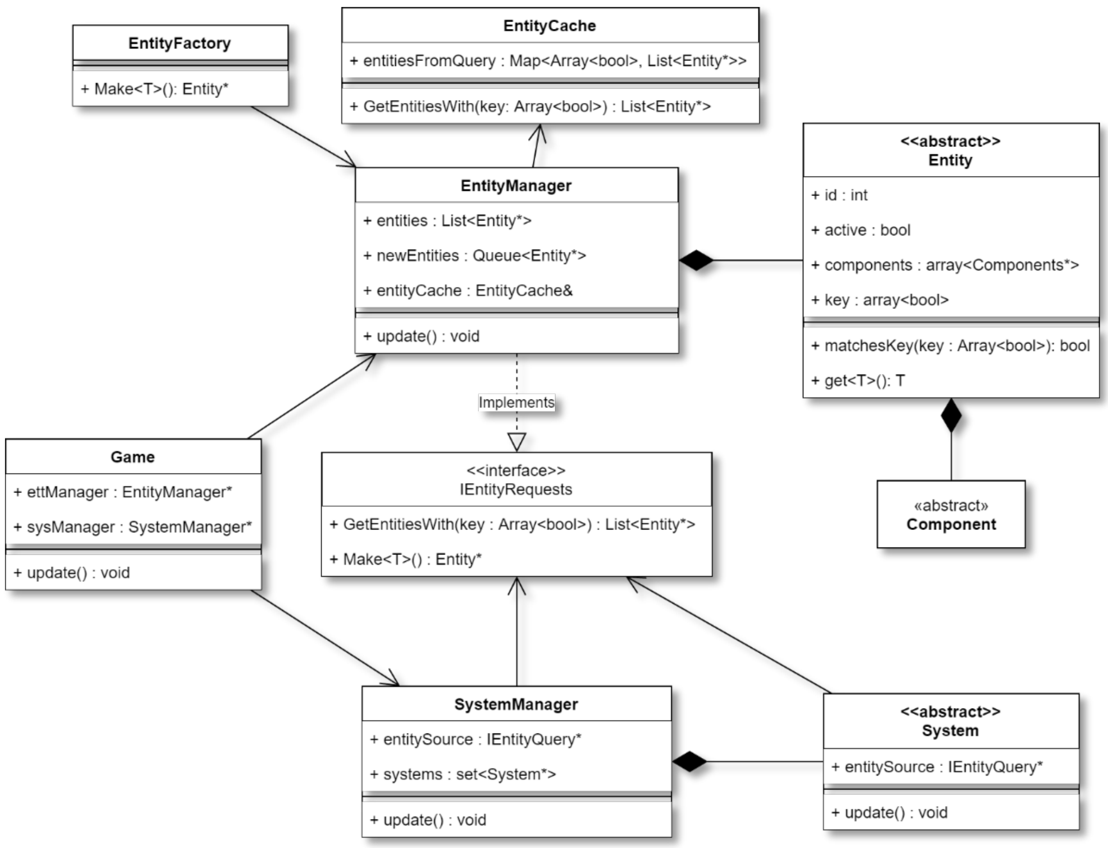
UML Diagrams

Architectural Class Diagram

Final Class Relationship Outcome (see *Architectural/Abstract Classes* for more class details)



Initially Proposed Design (from Task 26 - Custom Project Plan)


EntityFactory
+ Make<T>(): Entity*
EntityCache
+ entitiesFromQuery : Map<Array<bool>, List<Entity*>>
+ GetEntitiesWith(key: Array<bool>) : List<Entity*>
EntityManager
+ entities : List<Entity*>
+ newEntities : Queue<Entity*>
+ entityCache : EntityCache&
+ update() : void
<<abstract>>
Entity
+ id : int
+ active : bool
+ components : array<Components*>
+ key : array<bool>
+ matchesKey(key : Array<bool>): bool
+ get<T>(): T
Implements
Game
+ ettManager : EntityManager*
+ sysManager : SystemManager*
+ update() : void
<<interface>>
IEntityRequests
+ GetEntitiesWith(key : Array<bool>) : List<Entity*>
+ Make<T>() : Entity*
«abstract»
Component
SystemManager
+ entitySource : IEntityQuery*
+ systems : set<System*>
+ update() : void
<<abstract>>
System
+ entitySource : IEntityQuery*
+ update() : void

Explanation of Architectural Design

The goal of the architectural design throughout the creation of this program was to maximise cohesion and minimise coupling while also properly encapsulating the functions allowed to be used. To achieve this, the Entity-Component System (ECS) architectural pattern was used. My implementation adapts the ECS pattern to the Object-Oriented Programming paradigm for a more familiar and flexible methodology.

The Entity-Component System maximises cohesion by separating the logic from data. Rather than having to go through each class to implement logic, each system provides a centralized area to manipulate any entity desired. Each system is also very cohesive as they're only responsible for the logic they carry out. To minimise coupling, an effort was made to eliminate as many unnecessary relationships as possible. For instance, the SystemManager class is only responsible for its collection of Systems and never has to know about the game or entities involved. Additionally, the EntityManager only ever cares about its entities. The coupling between systems is also such that the removal of a system never affects the other – if all rendering-related systems were removed, the user wouldn't be able to see anything, but the game is otherwise technically playable. The biggest increases towards coupling would have to come from the singleton CompManager, EntityFactory and EntityFileLoader classes (as all their fields and functions are static) as their relationships are universally dependencies. These have been designed and implemented in this way to simplify the programming process as the data stored for these classes are applicable and never changes between all instances.

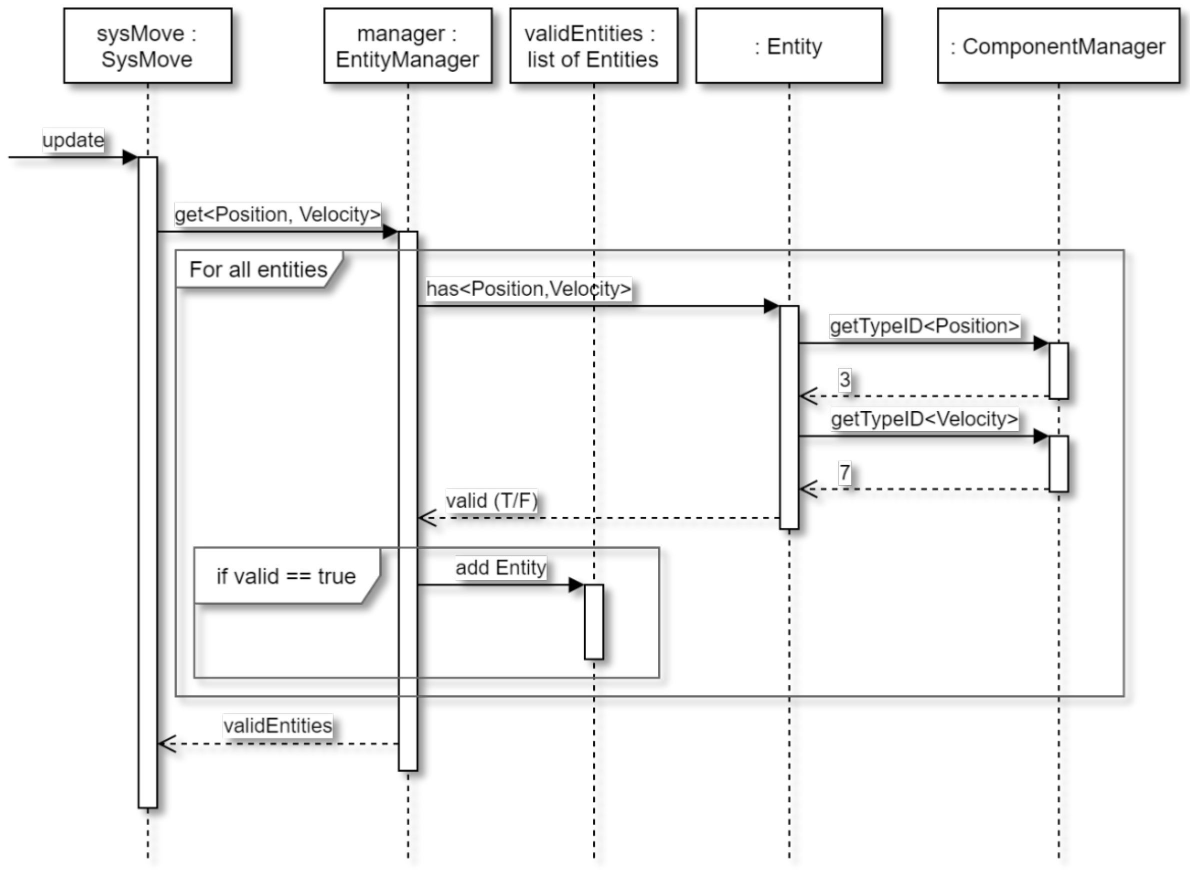
This approach also emphasises composition over inheritance to provide significantly better flexibility on what kinds of entities exist. Rather than relying on inheritance for the required data and methods, which can lead to difficult-to-manage inheritance trees, entities simply only need to have the correct sets of components to be attached to them. The factory pattern can then be used to have a centralised location for creating variations of these entities.

In my experience, the custom program I previously created for the Object-Oriented Programming unit, which uses the inheritance approach with game-related entities containing both their logic and data, was a lot more difficult to maintain and keep track of the relationships between classes was a nightmare. Compared to the final UML diagram included here the UML diagram for the previous program very much looks like spaghettified messed!

The architecture was also designed in a way such that the code is easily portable to an entirely different program, as the data and logic specific to this game primarily come from inheriting and implementing the abstract component and system classes and adding the factory methods. With minimal, most likely no changes at all, all classes depicted in the final UML diagram, asides from the EntityFactory and EntityFileLoader, can be reused to create another game or program.

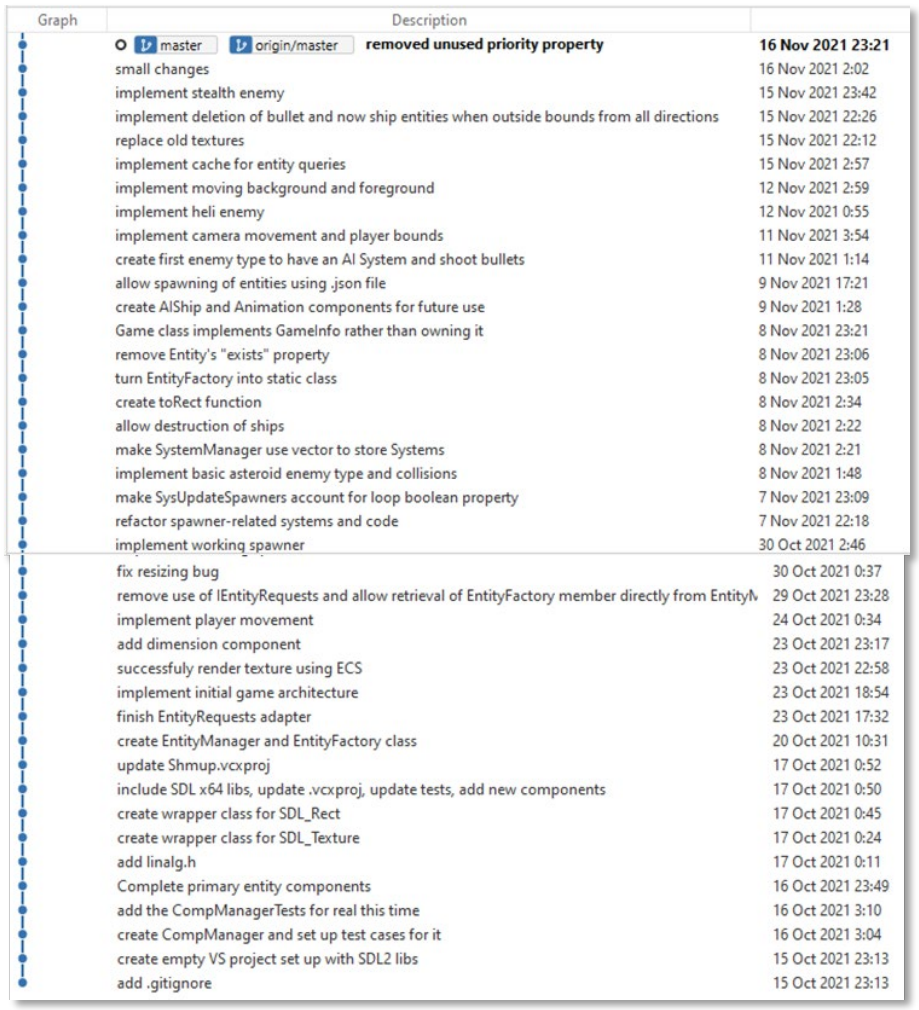
The initially proposed design is rather similar to the final design with some impactful differences. The GameInfo classes weren't designed yet since it was known what information would need to be accessed and encapsulated. There exists an IEntityRequests class to encapsulate the interactions between Systems and the EntityManager, however, the template functions couldn't be inherited. So, for simplicity, all Systems know the EntityManager. The EntityFactory used to store an EntityManager (and during development, the other way around). This made sense initially since an EntityManager could also be responsible for Entity creation, delegating the responsibility to the factory. However, the first issue arose when the IDE's IntelliSense would not reveal the factory's template specialisations from the EntityManager. The EntityManager and Factory also ended up mutually owning each other so the factory could insert created entities into the manager, which I felt negatively impacted cohesion and coupling more so than just having a standalone singleton factory.

Sequence Diagram



This sequence diagram, depicted above, shows interactions between the SysMove System, the EntityManager, Entities and the ComponentManager. SysMove queries the EntityManager for entities containing the Position and Velocity component via template arguments. No cache exists yet so the EntityManager proceeds to go through all entities, calling the ‘has’ function to check if they contain those components. This check involves retrieving the corresponding index value of the component type via the ComponentManager to check the vector elements contained by the entity. 3 and 7 are retrieved as an example.

Commit History Details



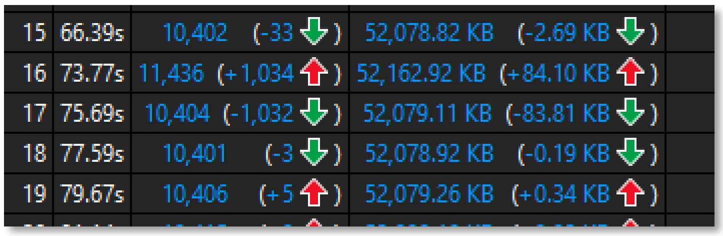
Description	
master origin/master **removed unused priority property**	**16 Nov 2021 23:21**
small changes	16 Nov 2021 2:02
implement stealth enemy	15 Nov 2021 23:42
implement deletion of bullet and now ship entities when outside bounds from all directions	15 Nov 2021 22:26
replace old textures	15 Nov 2021 22:12
implement cache for entity queries	15 Nov 2021 2:57
implement moving background and foreground	12 Nov 2021 2:59
implement heli enemy	12 Nov 2021 0:55
implement camera movement and player bounds	11 Nov 2021 3:54
create first enemy type to have an AI System and shoot bullets	11 Nov 2021 1:14
allow spawning of entities using .json file	9 Nov 2021 17:21
create AIShip and Animation components for future use	9 Nov 2021 1:28
Game class implements GameInfo rather than owning it	8 Nov 2021 23:21
remove Entity's "exists" property	8 Nov 2021 23:06
turn EntityFactory into static class	8 Nov 2021 23:05
create toRect function	8 Nov 2021 2:34
allow destruction of ships	8 Nov 2021 2:22
make SystemManager use vector to store Systems	8 Nov 2021 2:21
implement basic asteroid enemy type and collisions	8 Nov 2021 1:48
make SysUpdateSpawners account for loop boolean property	7 Nov 2021 23:09
refactor spawner-related systems and code	7 Nov 2021 22:18
implement working spawner	30 Oct 2021 2:46
fix resizing bug	30 Oct 2021 0:37
remove use of IEntityRequests and allow retrieval of EntityFactory member directly from EntityN	29 Oct 2021 23:28
implement player movement	24 Oct 2021 0:34
add dimension component	23 Oct 2021 23:17
successfuly render texture using ECS	23 Oct 2021 22:58
implement initial game architecture	23 Oct 2021 18:54
finish EntityRequests adapter	23 Oct 2021 17:32
create EntityManager and EntityFactory class	20 Oct 2021 10:31
update Shmup.vcxproj	17 Oct 2021 0:52
include SDL x64 libs, update .vcxproj, update tests, add new components	17 Oct 2021 0:50
create wrapper class for SDL_Rect	17 Oct 2021 0:45
create wrapper class for SDL_Texture	17 Oct 2021 0:24
add linalg.h	17 Oct 2021 0:11
Complete primary entity components	16 Oct 2021 23:49
add the CompManagerTests for real this time	16 Oct 2021 3:10
create CompManager and set up test cases for it	16 Oct 2021 3:04
create empty VS project set up with SDL2 libs	15 Oct 2021 23:13
add .gitignore	15 Oct 2021 23:13

Comments on Key Commits (From oldest to most recent)**:**

- The comments primarily focus on the motivations behind certain decisions and changes made to previous iterations.
- "Create CompManagerTests and set up test cases for it" (16 Oct)
    - Upon writing the first iteration of the ComponentManager, I wanted to make sure it works exactly as intended before moving on to writing the other architectural classes. This was motivated by the complexity of using variadic templates, which I've never used before.
- "Add CompManagerTests for real this time" (16 Oct)
    - I decided to use a dedicated testing library in a separate project so I didn't have to run and read everything in main.cpp. These tests can be found in: https://github.com/Andre-V/Andre-Shmup/tree/master/ShmupTests.
    - Later on, test cases were also written for the Entity, EntityManager and EntityFactory classes.
    - Note that these tests are no longer valid due to the numerous changes made since the last time they were used. Once the first few systems and game loop was implemented, testing was as simple as running the game and checking if everything was working as intended.

- "Create wrapper class for SDL_Texture" & "… for SDL_Rect" (17 Oct)
    - Tasks typically carried out for these classes, such as initialisation and rendering were placed within these wrapper classes (Texture.h and Rect.h) containing these classes as properties to delegate responsibilities away from systems and to hide anything related to SDL.
    - Rect.h never ends up being used in the end since this just added unnecessary complexity for something simple as converting position and dimension data into an SDL_Rect (which is a global function located in ConcreteSystems.h).
- "Remove use of IEntityRequests and allow retrieval of EntityFactory member directly from EntityManager" (29 Oct)
    - As stated by one of the comments added in this commit, it was very annoying having to continuously write methods that are just immediately delegated to the EntityManager anyways.
- "Refactor spawner-related systems and code" (7 Nov)
    - The spawner originally used the queue data structure to spawn entities. Looping the entities to be spawned was achieved by popping the entity and then pushing it back into the queue. However, this meant that the original ordering was not preserved, and I wanted the spawner to reset to the first entity when the spawner is no longer active. So, the queue was replaced by a vector with an index property to keep track of the current position.
- "Make SystemManager use vector to store Systems" (8 Nov)
    - As mentioned in the *Class Descriptions* section, using an ordered data structure was unnecessary as the ordering is never changed and can just be done by pushing the systems into the vector in the correct order in main.cpp.
- "Turn EntityFactory into static class" (8 Nov)
    - The motivations behind this are laid out elsewhere. Simplifies the relationships between it the EntityManager.
- "Remove Entity's "exists" property" (8 Nov)
    - A consequence of letting the EntityFactory be responsible for inserting entities into the EntityManager was that I didn't want certain entities to be handled by systems (those stored in a Spawner's sequence). This property told the EntityManager to ignore it during retrieval regardless of a match. Having this property alongside the 'active' property made things confusing and convoluted and was no longer necessary now that the factory was turned into a singleton. Entities to be ignored by the EntityManager are no longer given to it in the first place.
- "Game class implements GameInfo rather than owning it" (8 Nov)
    - Looking back, it seems that I went too far with composition over inheritance. Having the game class have its information in a separate class as its member was harder to track and ended up with data serving the same purpose.
- "create AIShip and Animation components for future use" (9 Nov)
    - Due to time constraints, animations were not implemented and hence this component is never used. This component was meant to work similarly to spawner, storing textures rather than entities.
- "Implement cache for entity queries" (15 Nov)
    - *(See sub-section Architectural/Abstract Classes, EntityManager)*
- "Implement deletion of bullet and now ship entities when outside bounds for all directions" (15 Nov)

- Now that most of the game has been implemented, I went back to an older system to prevent memory leaks as bullets were still being spawned by ships outside the playable area and would never hit the player anyways.



15	66.39s	10,402	(-33 ⬇)	52,078.82 KB	(-2.69 KB ⬇)
16	73.77s	11,436	(+1,034 ⬆)	52,162.92 KB	(+84.10 KB ⬆)
17	75.69s	10,404	(-1,032 ⬇)	52,079.11 KB	(-83.81 KB ⬇)
18	77.59s	10,401	(-3 ⬇)	52,078.92 KB	(-0.19 KB ⬇)
19	79.67s	10,406	(+5 ⬆)	52,079.26 KB	(+0.34 KB ⬆)

- Out of interest, the profiler was used to measure the impact of the memory leaks after this commit was pushed. The screenshot above shows measurements of the heap. One measurement was taken after firing several shots with all bullets still on the screen (#16, +84.10KB). Two more measurements were taken after all those bullets went off-screen (#17, -83.81 KB & #18, -0.19KB), indicating that all of the bullet entities were properly deleted. So, without SysDestroyOutOfBounds, there would've been memory leaks.

Screenshots/Appendices

Appendix 1 – Background & Foreground Textures