Cecil Beeland
SE456 - Professor Keenan
19 March 2018

SE456 - Space Invaders Design Paper

Cecil Beeland
College of Computing & Digital Media, DePaul University, Chicago, IL, 60604

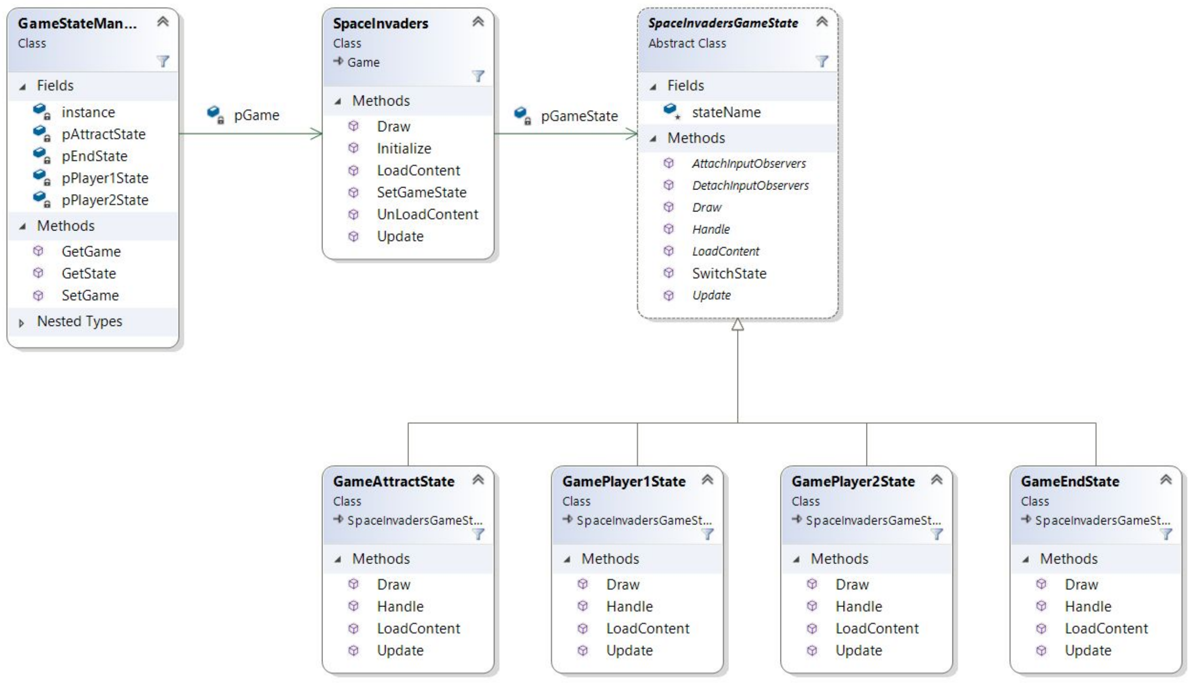
Overall Design

As part of our course on the Architecture of Real-Time Systems, we were tasked with developing a version of Space Invaders over the course of the quarter, emulating the basic functionality of the game, while leveraging modern object-oriented design patterns and best practices. For this exercise, we leveraged the Azul Game Engine, an existing game engine that was developed here at DePaul University, as the base of our system. The game engine provides the basic game update loop, as well as underlying classes that define the basic texture, image, and sprite implementations that we built upon.

We followed an iterative development approach, initially building some performant underlying data structures, followed by basic texture and sprite rendering, and eventually iterating out to the resulting game implementation. To achieve a working game, a number of subsystems needed to be architected, including the following:

- Overall Game State System
- Collision System
- Input System
- Sound System
- Font System
- Sprite System
- Game Object System
- Game Event Timer System

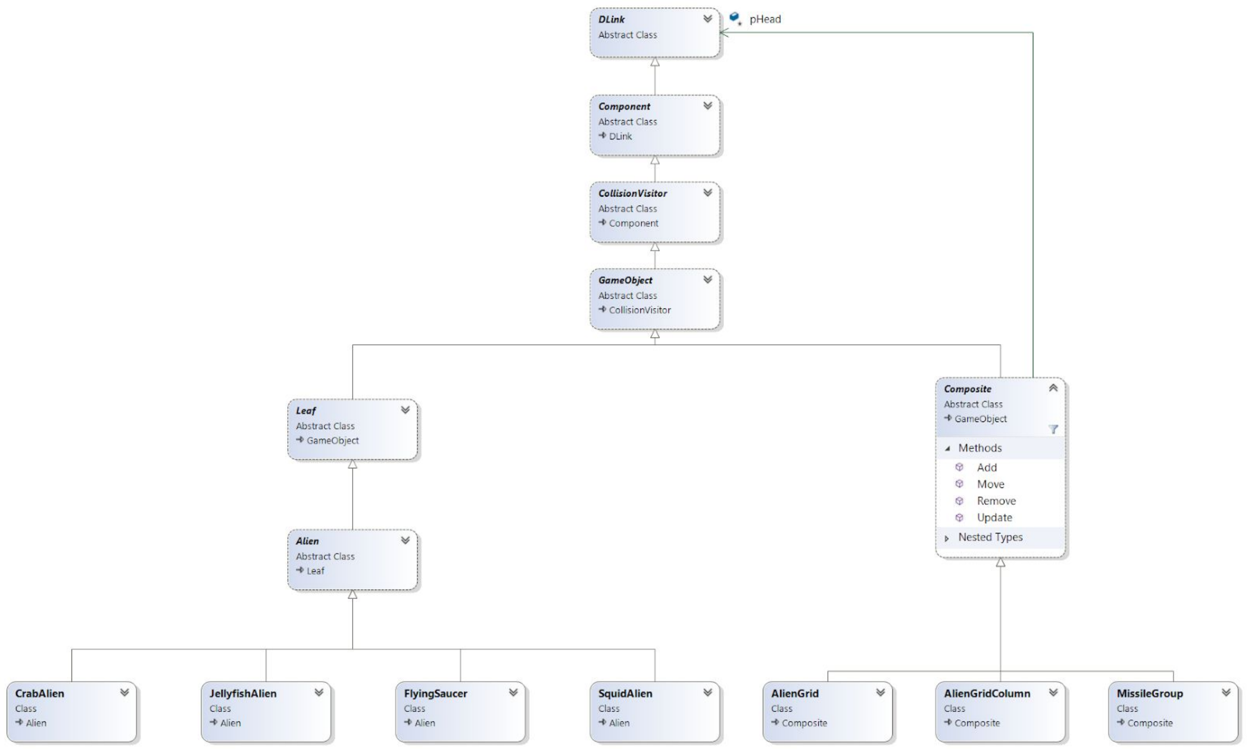
At the highest-level, the game is driven by an update loop that calls the Azul.Game Update() and Draw() methods every cycle. To enable different game states such as Attract mode, Player 1 Mode, Player 2 Mode, and End / Game Over Mode, I added a layer of abstraction to this update loop using the State pattern.



Game Modes- State Implementation

As shown in the figure above, the SpaceInvaders game contains a reference to a SpaceInvadersGameState object that defines LoadContent(), Update(), and Draw() methods. In the overall game update loop, I call the state-specific Update() and Draw() methods to implement game state scopes. Furthermore, each state defines a number of state-scoped objects and managers, which allows for the game to maintain distinct sets of objects for each active state.

Moving Further down the layers of the architecture, I want to highlight one of the most centrally important class hierarchies in the system: the GameObject. This class ties together the lower-layer subsystems like Texture, Image and Sprite, with the higher level systems like the Collision system and Input systems.



Alien Grid - Composite Implementation

The diagram above (which doesn't even include the lower-level systems like sprite, image, and texture that the GameObject class is built upon) illustrates how central the GameObject class is to all of the other systems in the game. From this we can see that child classes of GameObject are fundamental elements in the higher-level system interactions like Composites, Collisions, and Input responses.

Problem / Component Discussions

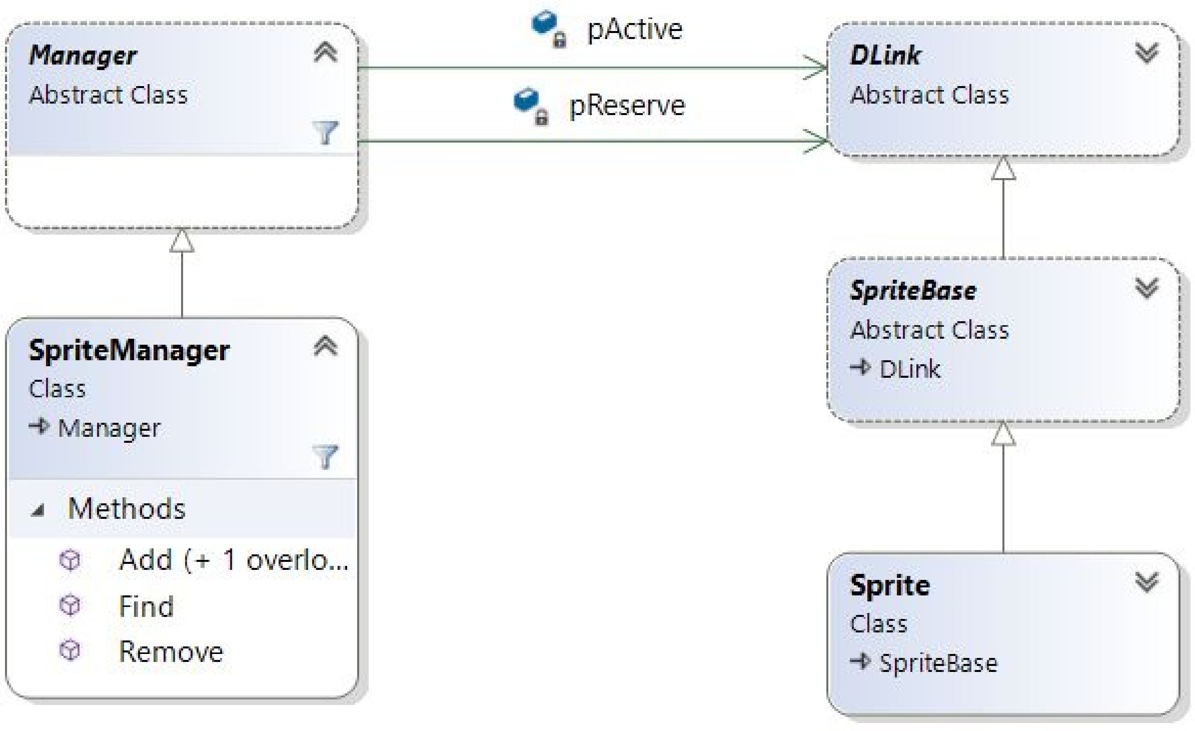
Object Pool

Problem:

We had the problem of managing the allocation and deallocation of collections of different types of objects in an efficient manner. Leveraging the ***new*** keyword and releasing references to objects throughout the game execution could cause undesired performance degradation due to changes in the heap and garbage collection activities.

Solution:

Collections of objects needed to be allocated and deallocated efficiently throughout gameplay. I used the Object Pool design pattern to manage these collections of objects. This pattern is a creational design pattern that leverages a set of objects (a “pool”) that are reserved for future use rather than allocating objects on demand. When an object is needed, a client of the pool can request an object from the pool instead of creating a new instance of the object. When tuned appropriately for a given use case, this allows a system to front-load the cost of object creation, so that the performance impact of object creation is not felt during performance-critical sections of code. Furthermore, this pattern allows for objects that are no longer actively being used to be recycled back into the object “pool” instead of being completely deallocated. This has the benefit of avoiding the performance impact of triggering a managed-language’s garbage collection routine during performance-critical sections of code.



Sprite Manager - Object Pool Implementation

This pattern provides the object pool via a managing class that contains two lists, an active/used list of objects and a reserved/available list of objects. When a client requests a new object to be allocated, the manager moves an instance from the reserved list to the active list and initializes it with any necessary values. If a client requests an object allocation and the reserved list is empty, the managing class can instantiate one or more new objects to repopulate the reserved list. However, this list replenishment can be minimized or even completely avoided with appropriate tuning of the list sizes.

In my implementation of Space Invaders, the Object Pool pattern is found in every manager contained in the game implementation. The general pattern is that any manager inherits from a Manager abstract class that holds pointers to the head of an active list and a reserved list of DLink doubly-linked list node objects, with supporting functions. Any object that needs to be managed in an object pool inherits from DLink to inherit the necessary list attributes and functions.

I have managers for almost every type of object in the game code - including, but not limited to, Textures, Images, Sprites, SpriteBatches, GameObjects, Composites, and the game Timer object. This allows me to create all necessary objects during the initial ***LoadObjects*** section of the game code and avoid object creation and destruction during the performance-critical game update/render loop section of the code.

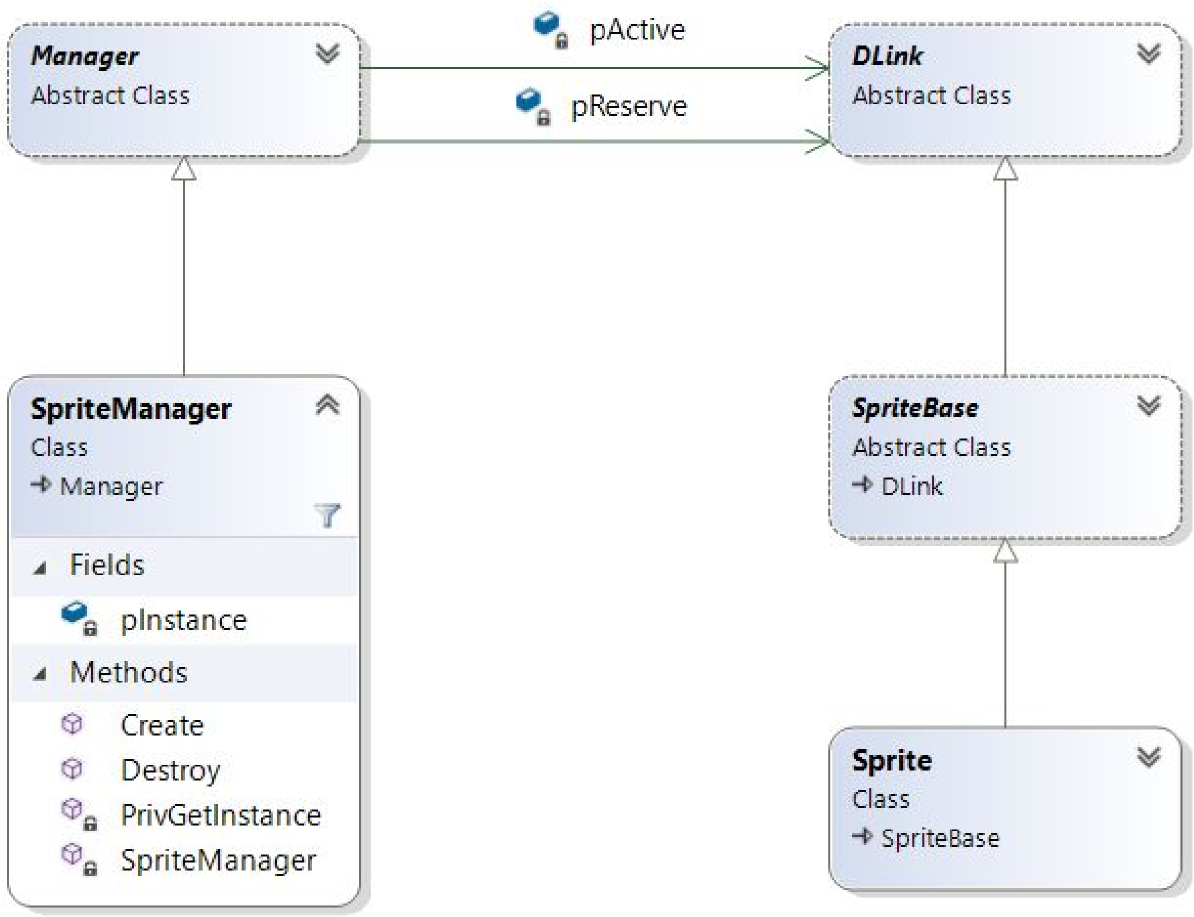
Singleton

Problem:

We had the problem of managing multiple sets of objects in a consistent, efficient, and global manner.

Solution:

Collections of objects needed to be managed in a consistent, efficient, and global manner throughout gameplay. I used the Singleton design pattern to provide a single, consistent manager for each collection of objects. This pattern is a creational design pattern that restricts the instantiation of a class to a single object. This is useful when exactly one object is needed to coordinate actions across the system.



Sprite Manager - Singleton Implementation

This pattern restricts the instantiation of a class to a single object by making the only constructor private so that only the singleton can instantiate itself. In addition, the Singleton class contains a Static attribute of itself (pInstance in the diagram above) and number of static methods for manipulating the instance. Furthermore, there is a Static method (generally called **Create()** or **GetInstance()**) that will create an instance of the Singleton class and store it in the Static instance attribute - if and only if there is not already an instance stored in the Static attribute. This ensures that there is only ever one instance of the class in existence at a given point in the code. Additional static methods can be added to retrieve, manipulate, and/or destroy the instance.

In my implementation of Space Invaders, the Singleton pattern is found in many of the managers contained in the game implementation - including Textures, Images, Sprites, SpriteBatches, CollisionPairs, and the game Timer object. This allows me to create and manage these objects from a single, consistent point of ownership, loading necessary objects into Singleton managers during the initial ***LoadObjects*** section of the game code and ensuring object destruction at the appropriate point in the game.

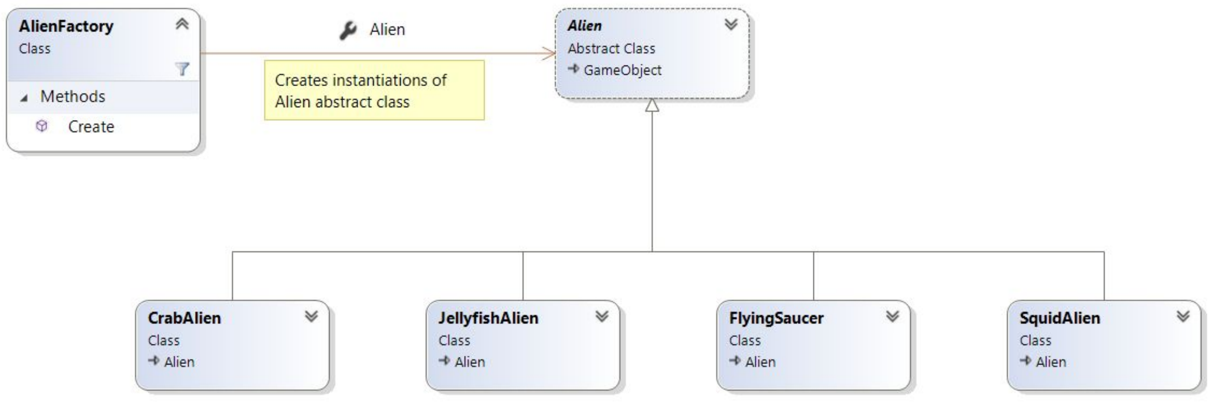
Factory

Problem:

We had the problem of creating a large number of similar objects that may only vary in attribute values or a few specific characteristics. Rather than create them in the primary gameflow code-block, we needed an organized way to consistently create many similar objects.

Solution:

A large number of objects needed to be created in a consistent, organized manner throughout gameplay. I used the Factory design pattern to provide a single, consistent managing class for the creation of these objects. This pattern is a creational design pattern that creates objects without exposing the instantiation logic to the client and refers to the newly created object through a common parent class. This is useful when many similar objects need to be created repeatedly in a consistent, reusable manner.



Alien Factory - Factory Implementation

This pattern leverages a Create() method that returns an instantiation of the Alien abstract class, configured according to any provided parameters. This implementation of the factory method is often described as the *Simple Factory* pattern, where there is a simple switch statement in the Create() method that determines which type of instantiation of the abstract class to create. There are other, more complex, variations of this factory pattern, including one called the Factory Method pattern, where there is an abstract Factory class (or interface) that would define the general functionality of all factories, and then a specific Factory instantiation for each type of instantiation of the abstract class to create.

In my implementation of Space Invaders, the Factory pattern is being leveraged for the creation of the Alien game objects. During the LoadContent() code section at the initialization of the game, the AlienFactory is created and is used to generate the multiple copies of the different types of aliens that need to be on the screen at the beginning of gameplay.

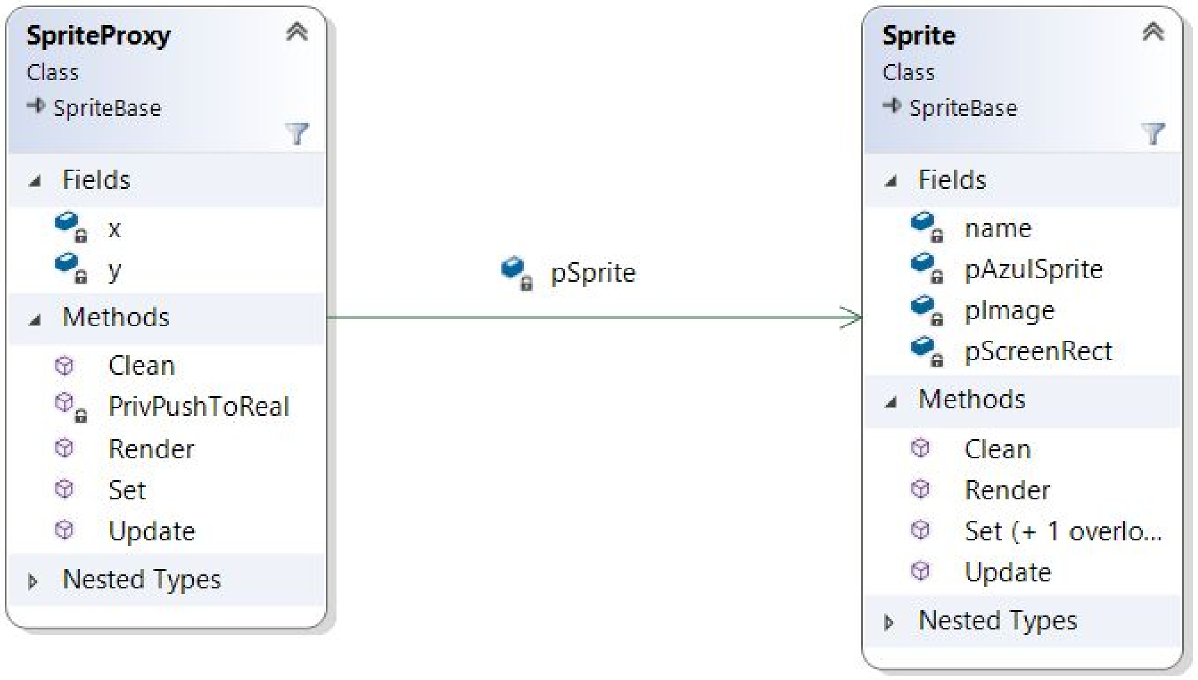
Proxy

Problem:

We had the problem of generating and managing many copies of objects that shared most of their attributes and vary only in a few unique characteristics. Rather than create a large number of the full objects, we wanted to be able to create smaller objects that can set their unique attributes, but otherwise rely on the complete objects.

Solution:

I used the Proxy design pattern to create smaller objects that can set their unique attributes, but otherwise rely on the complete objects. This pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it or to provide additional logic or attribute values unique to the surrogate object.



Sprite Proxy - Proxy Implementation

In the Proxy pattern, the proxy object points to a full service object. The proxy objects often have a small number of attributes that they push updates to the full service object, thereby leveraging the full service for most attributes, but providing unique values for those shared attributes. Proxy objects have the same functionality of the service object by first pushing any updates to shared attributes, then delegating function calls to the full service object.

In my implementation of Space Invaders, the pattern is used to create multiple copies of a single sprite object that differ only by their X and Y coordinates, providing improved efficiency in both complexity and performance.

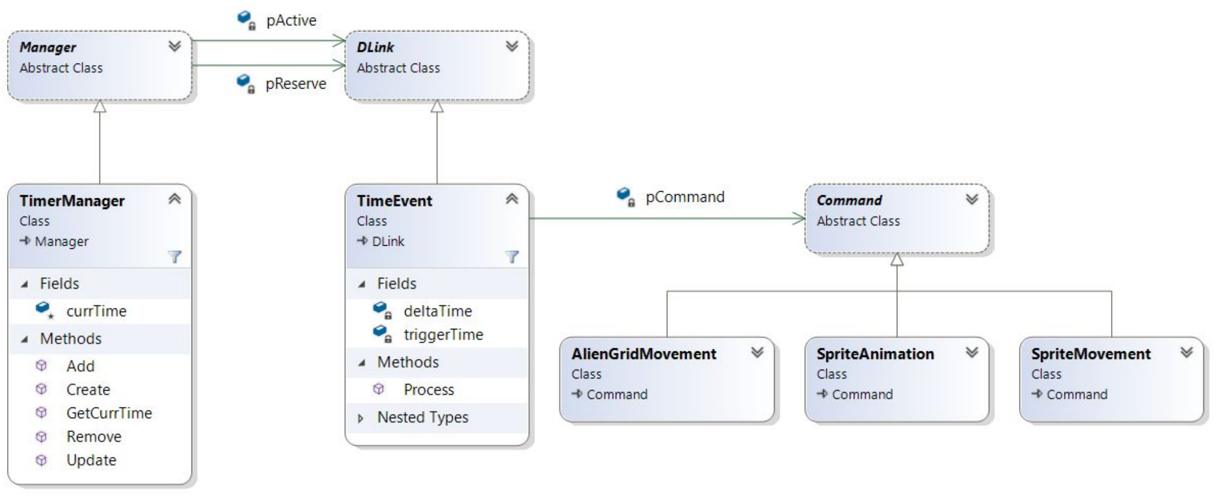
Command

Problem:

We had the problem of managing a large number of events with a consistent ordering of execution of those events. Furthermore, we needed a mechanism to dynamically add new events and potentially modify the ordering of the events at runtime.

Solution:

I used the Command design pattern to create event objects that could be executed in an ordered manner via iteration over a containing list data structure. This pattern is a behavioral design pattern that allows you to use an object to encapsulate all information needed to perform an action or trigger an event at a later time.



Time Event - Command Implementation

In the Command pattern, a Client asks for a command to be executed and an Invoker takes the command, encapsulates it and places it in a queue. When invoked from the queue, a concrete implementation of the command executes and sends its results to the Receiver.

In my implementation of Space Invaders, the pattern is used to manage events on the game timer, including Sprite animation actions, as well as Sprite movement commands, both individually and collective movements through the Composite pattern. The TimerManager maintains a list of pending commands with predefined trigger times. When the game time reaches or passes the trigger time of a particular command, the concrete command class executes its behavior on the receiver class. In the diagram above, the concrete commands are SpriteAnimation, SpriteMovement, and AlienGridMovement. The first two execute actions against individual sprites, while the last executes an action against a Composite of sprites.

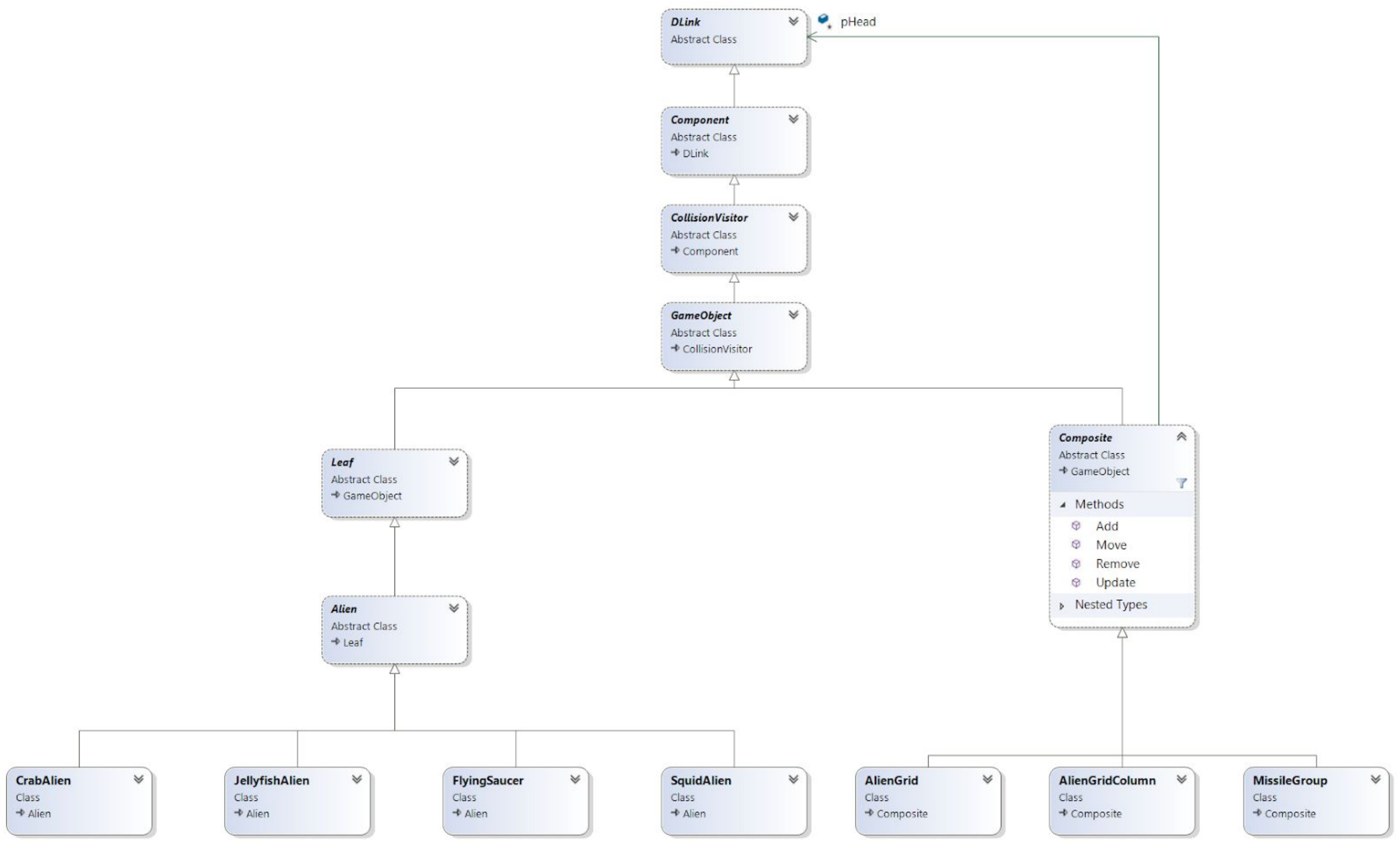
Composite

Problem:

We had the problem of managing and moving game objects together in a consistent and organized manner. We needed a way to manage the movements of aliens without having to individually manage each alien's movement.

Solution:

I used the Composite design pattern to create object hierarchies that could be managed and manipulated in a structured and ordered manner. This pattern is a structural design pattern that allows you to compose multiple objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.



Alien Grid - Composite Implementation

In the Composite pattern, a tree data structure is used to structure objects and to perform operations on all nodes, regardless of if a node is a branch of a leaf. One important aspect to note is that only Leaf objects represent actual elements in the data structure. Composite objects only exist to define and maintain the tree structure.

In my implementation of Space Invaders, the pattern is used to define and manipulate the grid of 55 Aliens that moves synchronously around the game window, as well as the grids of Shield Bricks that comprise each shield in the game. The composite pattern allows for us to define a structure of rows and columns for the aliens and shields, giving us the necessary control structure to manipulate each type of object as a single unit. In the diagram above, the AlienGrid game objects are included to demonstrate the pattern. You can see that the composite pattern is interleaved with elements of other patterns in the class hierarchy. By adding the GameObject class above both the Leaf and Composite classes, I was able to treat Composite objects and GameObjects equally, which greatly simplified iteration of the composites.

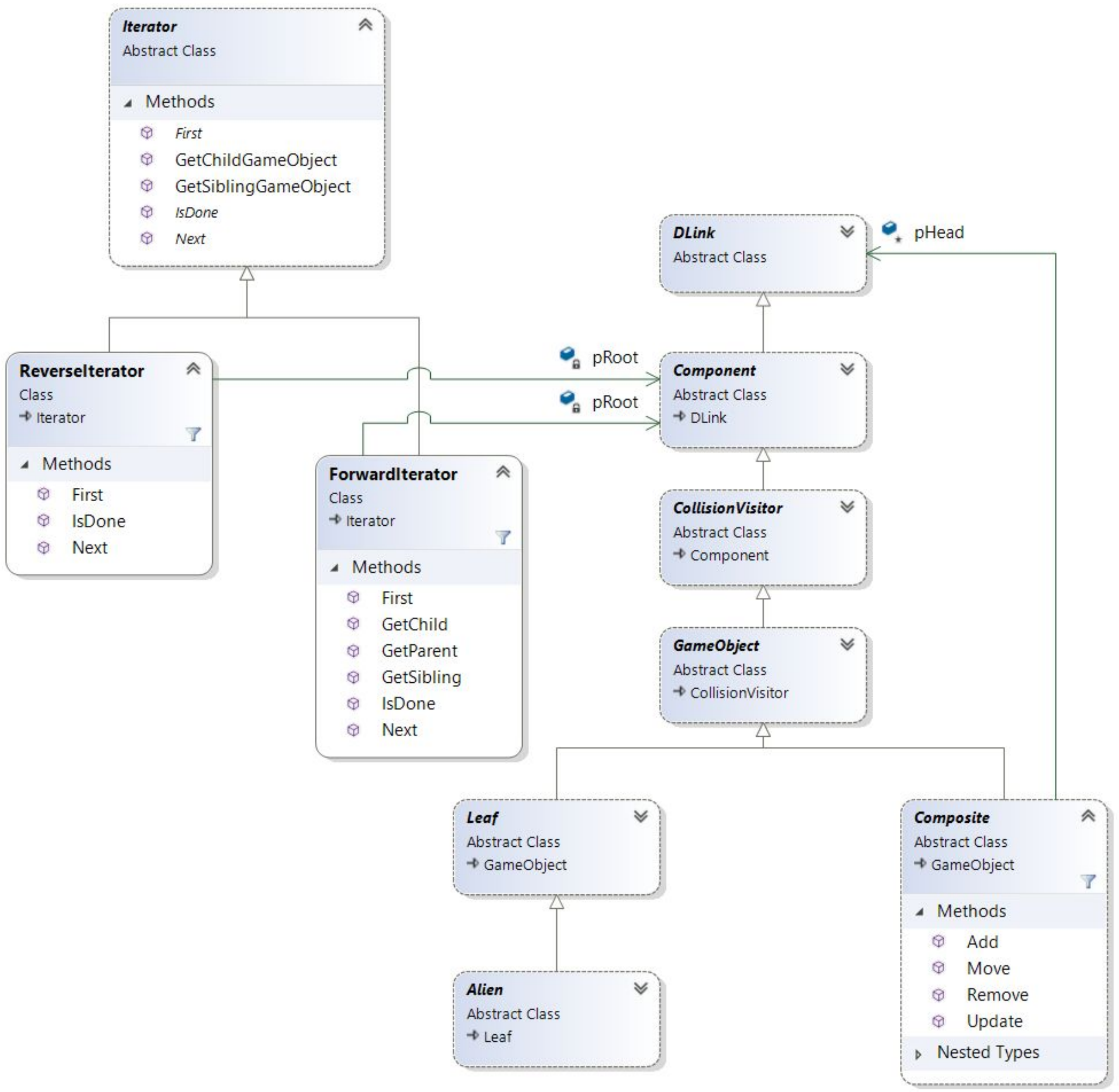
Iterator

Problem:

We had the problem of iterating through our Composite tree structure in a consistent and efficient manner. Furthermore, we wanted to encapsulate the details of the structure of the Composites so that clients would not need to know the structural details of the collections to iterate over their elements.

Solution:

I used the Iterator design pattern to provide a way to access the elements of the Composite objects sequentially without exposing its underlying representation. This pattern is a behavioral design pattern that allows you to use an object to iterate through a collection in a consistent and encapsulated manner. It also provides an abstraction that makes it possible to decouple collection classes and algorithms.



Component - Iterator Implementation

In the Iterator pattern, a Client creates an iterator object that is associated with a collection and leverages methods in that iterator object to iterate over the collection, rather than iterating over the collection directly. The iterator object generally has methods such as First(), IsDone(), and Next() to enable the iteration by the client.

In my implementation of Space Invaders, the pattern is used to iterate over the collections in the game, specifically any instances of the Composite pattern, such as the AlienGrid and Shield composites. Furthermore, my implementation of the iterator pattern specifically avoids recursion as a mechanism for traversing the tree structure of the Composites. While this approach complicates the iterator implementation somewhat, we have a particular focus on performance for the Space Invaders game (both algorithmic complexity and memory usage). Using a non-recursive approach avoids adding additional layers onto the stack as the recursion progresses, which can become a significant issue if the collections are large enough.

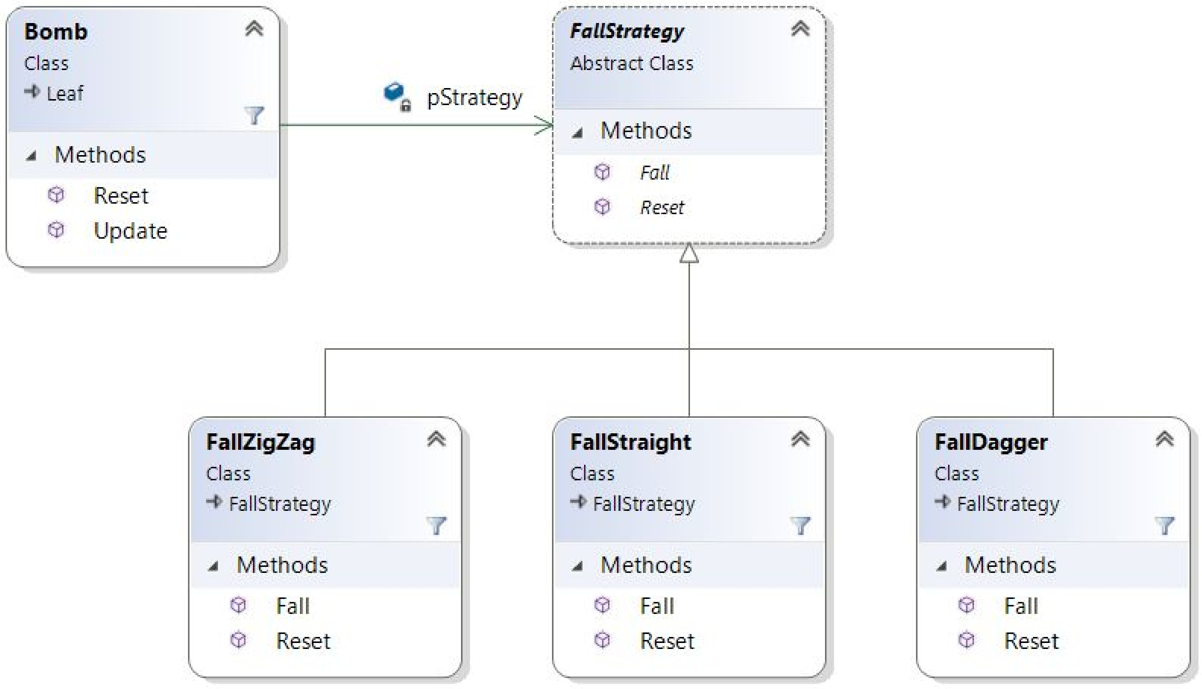
Strategy

Problem:

We had the problem of creating and leveraging a family of algorithms that are similar but each unique in their own way, specifically for managing different bomb fall strategies. We wanted a way to provide these strategies to their encapsulating objects without the objects being hard-coded to a specific implementation of the strategy.

Solution:

I used the Strategy design pattern to define the different strategies and provide them to their encapsulating classes. This pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. It allows you to capture the abstraction in an interface and encapsulate the implementation details in derived classes.



Bomb Fall Type - Strategy Implementation

In the Strategy pattern, a Client contains a reference to an abstract class that represents the family of algorithms and provides the general interface that all of the strategies implement. While the client sees the abstract class as the instance that it interacts with, in actuality an implementation of the abstract class is provided to the client. This allows for the client to be unaware of the specific implementation of the strategy that it is using, as well as the ability to switch out the strategy implementation at runtime.

In my implementation of Space Invaders, the pattern is used to define and manage the Bomb Fall strategies. In the bomb fall context, each bomb is provided an instance of FallStrategy upon instantiation. The FallStrategy.Fall() method is called during each update cycle and each implementation of FallStrategy has its own Fall() behavior. From the bomb's perspective it is always calling the same method, even though the underlying behavior varies depending on the implementation.

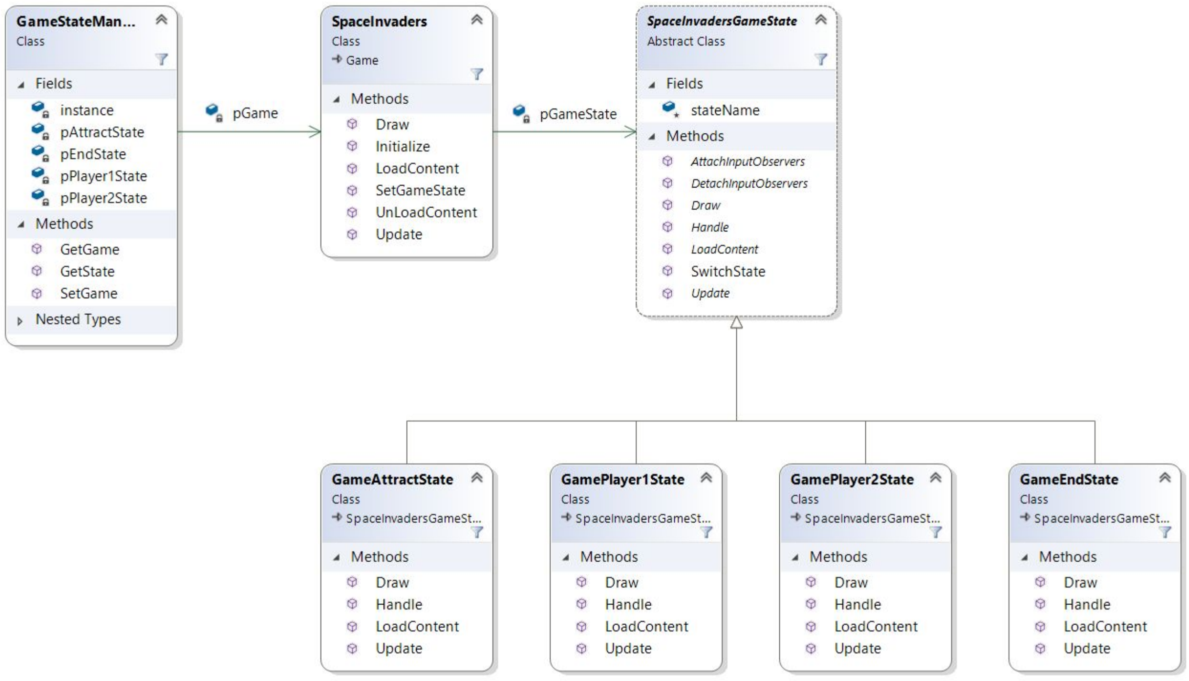
State

Problem:

We had the problem of creating and managing complex state-based workflows. The basic underlying game update loop needed to be enhanced to support multiple game states, such as Attract mode, Player 1 mode, Player 2 mode, and End mode. Additionally, the player's ship required state-based behavior such as only allowing the player to fire a missile when there is not already a missile on the screen.

Solution:

I used the State design pattern to implement a state machine in an object-oriented way. This pattern is a behavioral software design pattern that enables implementing a state machine by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass.



Game Modes- State Implementation

In the State pattern, states are implemented as full-blown classes that derive from the abstract class that defines the state pattern interface. The client contains an instance of the abstract class and has the ability to switch out states, and in turn the data and behavior defined in the state implementations. Generally the state interface has a Handle() method that triggers a state change from one state to the next state defined in the workflow.

In my implementation of Space Invaders, the pattern is used to define and manage the overall game states, as well as the player's ship states. In the game state context, each state implements Draw(), LoadContent(), and Update(). In the overall game loop methods of the same names, I call the state methods to provide game state scopes. Furthermore, while there is some data that is global across all game states, there is other data that is scoped to a game state, where each state has its own copy of the data that can vary between states.

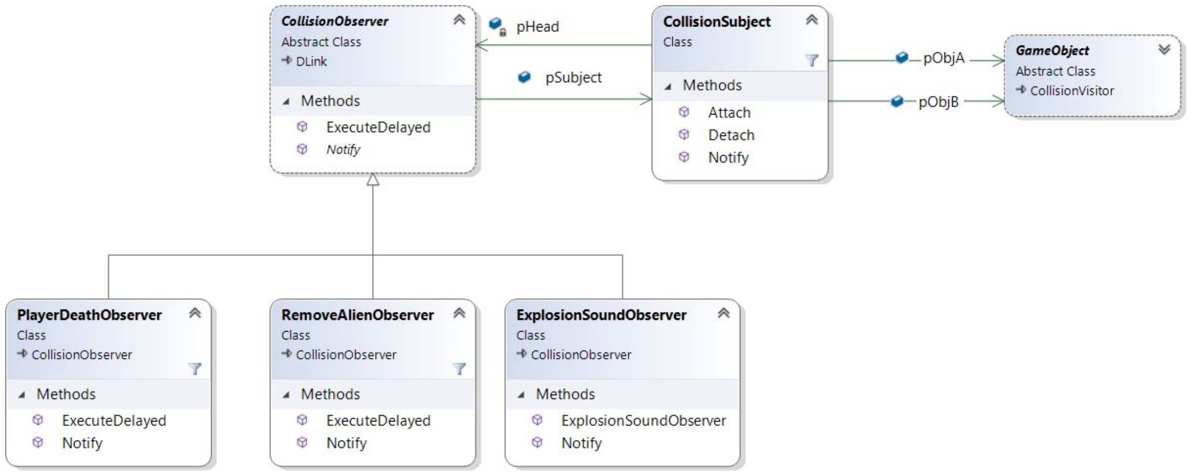
Observer

Problem:

We had the problem of managing complex interactions between different game objects. The collision system of the game required a set of different actions to be triggered when two types of objects that we care about collided on the game screen.

Solution:

I used the Observer design pattern to implement the collision system, decoupling the management of collisions from the management of the resulting actions. This pattern is a behavioral software design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Collision System- Observer Implementation

In the Observer pattern, an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods, such as Notify(). The observers can execute any desired behavior based on the notification by the subject.

In my implementation of Space Invaders, the pattern is used to define and manage the reactions to collisions in the game. In the collisions context, each CollisionPair (Ex: Alien vs Missile) maintains a CollisionSubject that maintains a list of observers, as well as two GameObject attributes that can be set to the objects involved in the collision at notification time. The desired collision reaction behavior is added as observers during LoadContent(), defining the collision rules and reactions for each game state.

Post-Mortem

The experience of developing Space Invaders was as illuminating as it was challenging. Even a game as simple as Space Invaders results in an incredibly complex real-time system and I truly cannot imagine how I would have been able to complete this project without leveraging the design patterns that were presented throughout the course.

While I am proud of the work that I have accomplished over the course of the project, there are a number of items that I left outstanding due to time constraints, including:

- Death Animations
    - Explosion Splat when Alien is struck by missile
    - Explosion Splat when Alien Bomb is hit by missile
    - Explosion sprite animation when UFO is killed
    - Splat animation when Missile and Bombs hit each other
    - Ship death animation when collision with bomb or alien
    - Graphical noise base Dissolve effects (optional)
- Improving the Bomb Drop Command
- Refining the Bomb Drop Rate
- Adding a “Credits” system to further emulate the arcade format
- Additional testing of system behavior corner-cases

I feel that I have all of the underlying infrastructure in place for the items that I was not able to achieve, but simply ran out of development time to refine the gameplay. Additionally, while I am confident that the basic gameplay of the system is functional. I expect that there are still a number of bugs that will exhibit themselves then corner-cases in gameplay are reached, as I did not have time to test the system as thoroughly as I would have liked.