Game Design Document

Joey Domino

Space Invaders is a very complicated project. Sure, a lazier developer could slap something together with minimal effort, but I'm no lazy developer! My version of Space Invaders is built using object oriented programming principles and development strategies. However, building the game using these complicated systems will and did lead to many difficult problems. I will be covering various aspects of the game, its development, and the problems unearthed during the construction of many of the game systems required to complete the project as well as the solutions put in place to rectify them.

Let's start off by taking a look at one of the most integral parts of the game: the alien army. The army must consist of 22 crab aliens, 22 octopus aliens, and 11 squid aliens to fit the original game's design. Here is where I reach our first problem. How do I create these aliens? I could just grab the image from a sprite sheet and display that, but that would only allow us to see the alien on the screen. There would be no interaction.

To solve this, I have created a game object class to store all required data for each item displayed on screen that is able to be interacted with. These game objects hold a sprite proxy class (this holds a sprite, which is a box somewhere on the screen that holds the given image), a collision object class, and the x and y coordinates where the game object is on screen. The coordinates will line up to the sprite proxy's location.

The sprite proxy's purpose is to save on resources on runtime. It allows me to update the sprite's information without rendering to the screen. Then, when I decide it is time, I update the sprite's info and call Azul (the system used to render to screen) with said information to update what I see on the screen. Reducing the amount of times I call Azul is paramount to keeping the game running quickly. With the proxy in place, I need to create a way for this game object to be collided with by the player's missile. Enter the collision object.

A collision object is a sprite box around the sprite that was just created. This will be used to compare one collision object, like the missile, against another, like an alien. This leads to another very complicated problem. How do I handle collisions between two game objects? The solution may sound simple, but programmatically it's very intense.

I have built a group of collision classes to handle this. To associate these collisions easily with the game objects, I have created a collision visitor abstract class that all game objects are derived from. This class on its own does nothing but hold virtual methods for every type of collision possible in the game. These virtual methods do nothing but error out if they are called. It is each type of game object's responsibility to handle their collisions via overriding the collision visitor's virtual methods as needed. For example: Say the player fires a missile and it collides with an octopus alien. What will happen in the code?

At first, the missile will constantly check an iterator of collision types using the collision pair manager, one of which being a missile vs. alien check. The manager will call collision pair's

Collide method to check if these collisions have occurred. Then, once it connects to another game object's collision box, the manager will call the missile's Accept() method, which will then call the VisitMissile override method of the octopus alien it collided with and process what do to next by calling the collision pair's listeners.

A listener is a series of commands that I have called observers that execute given a certain collision. In this case, our listener's would require us to remove the alien, remove the missile, and allow the ship to fire again. So I created 3 observers, RemoveAlienObserver, ShipRemoveMissileObserver, and ShipReadyObserver. These three observers are added to the collision pair's list of collision types to check before runtime which are then repeatedly referenced during runtime.

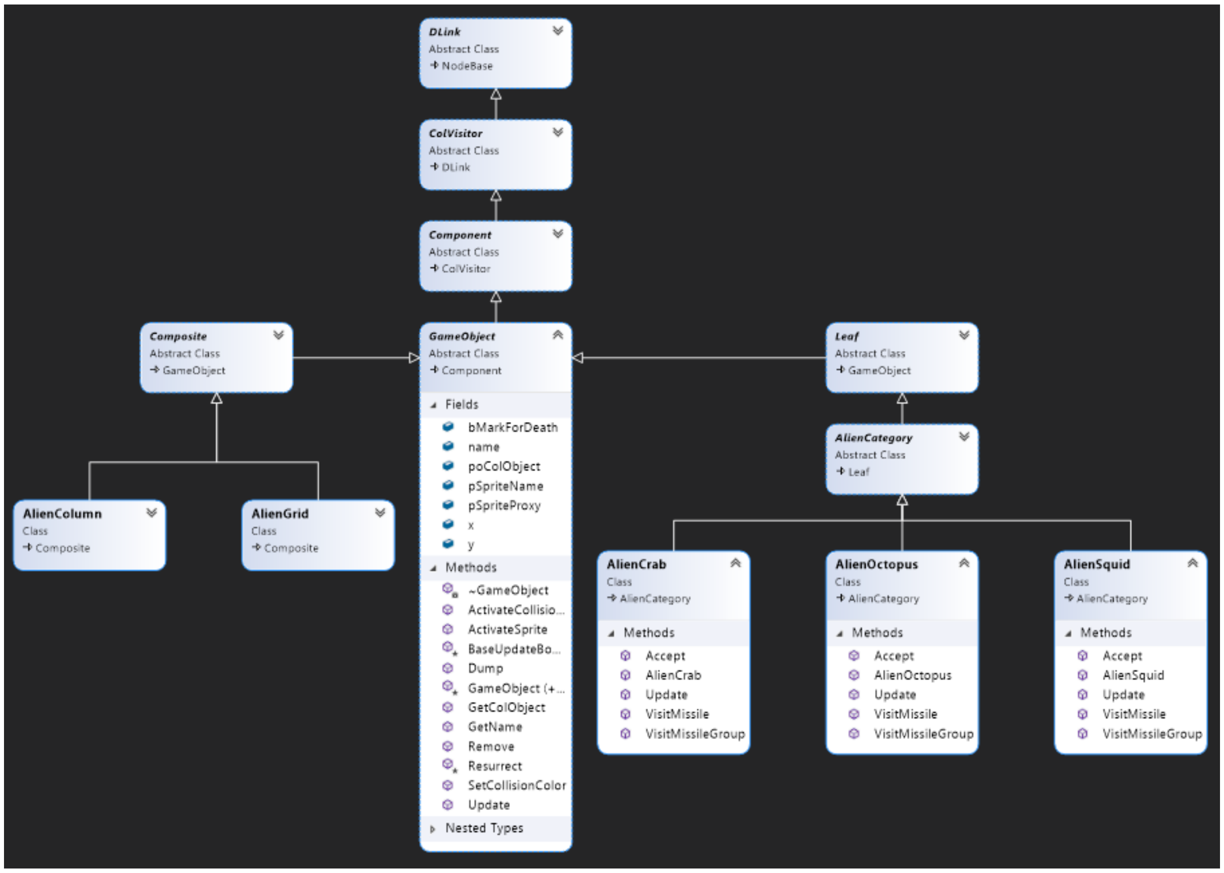
So our alien game object is created, filled with a sprite proxy containing a sprite that holds an image that is displayed to the screen, and a collision box that wraps around the proxy to handle collisions. Now what? This would be all that's needed for the army if the army was one alien strong, but there are 55 up there that need to be registered! To properly handle all aliens as one, I have created a column and grid system that will house our aliens to allow easier manipulation.

The alien grid and columns work similarly. Both are also game objects, but serve a specific purpose. When an alien game object is created, they then get attached to a column. Once 5 aliens are stored in that column, start building out another column. Repeat until all 55 aliens are stored. Then, attach all those columns into the grid. Both the column and grid have collision boxes created when first made and updated in size with each addition. But how are these aliens added to columns and columns to the grid?

This is what makes these two, and others like them, different types of game objects. I need a way to reference each alien game object simply during runtime. To solve this, I created two derisions of game object: a leaf and a composite. Leaves are the actual game objects like the ship, aliens, missile, and bombs, while the composite game objects are what hold the leaves. So the column composite will hold the alien leaves in a double linked list. Then, that composite column is attached to the grid's double linked list. These lists are then referenced during runtime against the collision checks stated above. This allows me to easily check conditions like detecting the left and right walls allowing for the army to shift horizontally and vertically along the screen.

With that, our army is complete! What I have built looks like the attached UML diagram. In it, you can see the relationship between the collision visitor, game object, leaf, composite, column, grid, and aliens. Simple right?! While this does not show the entirety of the game, most other systems function the same as what I have shown above with minor tweaks.

Creating this game has been one hell of a nightmare. I never thought I'd write and edit so much code in just 10 weeks! Sheesh! I left the UML enlarged for easier viewing on the next page.


DLink
Abstract Class
NodeBase
ColVisitor
Abstract Class
DLink
Component
Abstract Class
ColVisitor
Composite
Abstract Class
GameObject
GameObject
Abstract Class
Component
Fields
bMarkForDeath
name
poColObject
pSpriteName
pSpriteProxy
x
y
Methods
~GameObject
ActivateCollisio...
ActivateSprite
BaseUpdateBo...
Dump
GameObject (+...
GetColObject
GetName
Remove
Resurrect
SetCollisionColor
Update
Nested Types
Leaf
Abstract Class
GameObject
AlienCategory
Abstract Class
Leaf
AlienColumn
Class
Composite
AlienGrid
Class
Composite
AlienCrab
Class
AlienCategory
Methods
Accept
AlienCrab
Update
VisitMissile
VisitMissileGroup
AlienOctopus
Class
AlienCategory
Methods
Accept
AlienOctopus
Update
VisitMissile
VisitMissileGroup
AlienSquid
Class
AlienCategory
Methods
Accept
AlienSquid
Update
VisitMissile
VisitMissileGroup