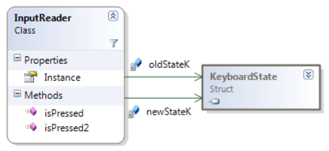
Robert Wascher
Final Project

Final Project: Space Invaders

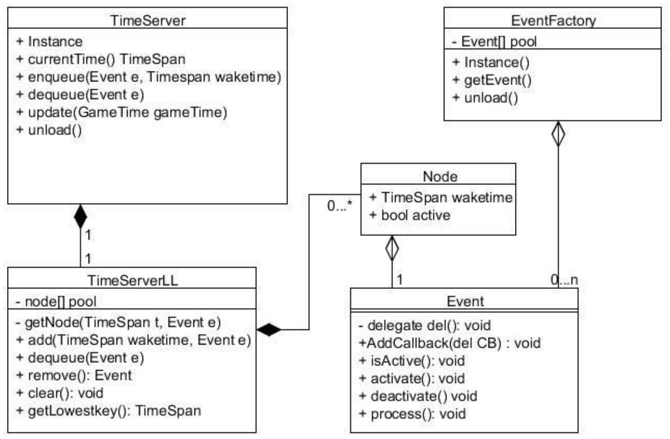
Video: https://www.youtube.com/watch?v=U4qwxUTYVKU

Input



Because of the simplicity of the input of the game, input reading is handled by only one class, InputReader. This class has 2 methods to discern whether a key has been pressed. isPressed() returns whether a button is being pressed. It returns true as long as the key is still be pressed. isPressed2() returns whether a button is pressed. It returns true only the instant it is pressed. A key has to be release and re-pressed for isPressed2() to return true again.

Time Server



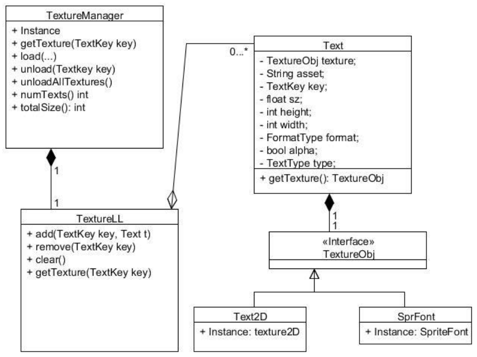
The TimeServer class controls the processing of timed events of the game. The Time Server checks to see if it contains any events with a wake-time <= current time and processes each one it finds that needs to be awoken. To handle the ordered storing of these events I implemented a sorted linked-list of Events sorted by each Events wake-time. The sorting is done upon an Event being added to the list. I used a simple traverse and drop process to handle this sorting. To eliminate dynamic allocation in this linked-list, I initialize the linked-list with a fixed array of created node. The timeserver can handle 50 nodes in the list before it needs to create more through dynamic allocation. To address the same issue, I also added an EventFactory class that holds a pool of pre-created events for reuse.

Robert Wascher
Final Project

Sprite System

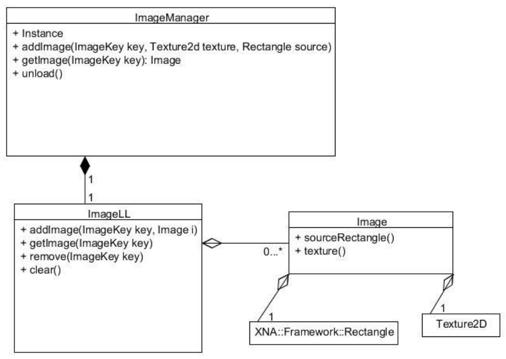
The Sprite system handles the storage and use of classes to handle the graphical nature of the program. It includes the texture manager, image manager, sprite manager, spritebatch manager, and animation manager.

Texture Manager



The TextureManager class controls the loading, storing and access to various Texture objects. To handle the storage I used a linked-list TextureLL. This linked-list control access to various textures using an Enum TextKey. To be able to store any type of texture in this linked list I had to implement an interface TextureObj. TextureObj objects are found and accessed from the Text class. A class I created to solve the problem of needing to store various attributes about each texture. This allows each texture to store the same data regardless of the type of texture it is. The TextureObj interface is implemented by 2 different classes, Text2D and SprFont. These are basically just classes that have a reference to a Texture2D or SpriteFont, respectively. I needed to wrap these classes in order to have them implement the TextureObj interface. The use of the TextureObj interface allows these different types of textures to be stored, but it has a drawback as well. In order to access the underlying texture from a Text class, one has to cast the TextureObj to the appropriate subtype.

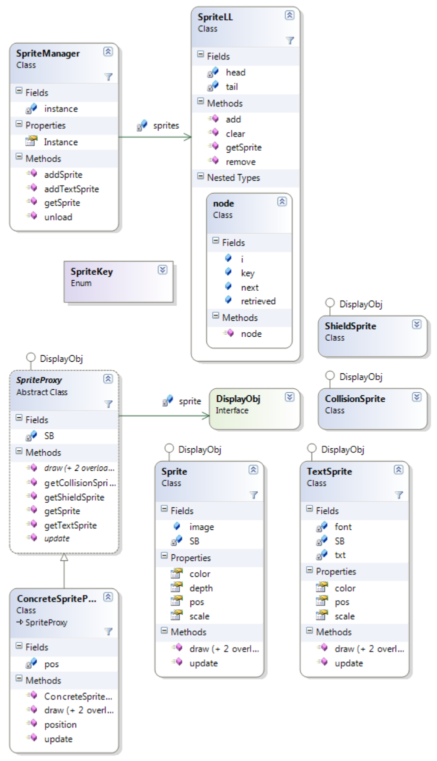
Image Manager


ImageManager
+ Instance
+ addImage(ImageKey key, Texture2d texture, Rectangle source)
+ getImage(ImageKey key): Image
+ unload()
1
1
ImageLL
+ addImage(ImageKey key, Image i)
+ getImage(ImageKey key)
+ remove(ImageKey key)
+ clear()
0...*
Image
+ sourceRectangle()
+ texture()
1
XNA::Framework::Rectangle
1
Texture2D

The ImageManager class controls the loading, unloading, storing, and access to various images. This allows for displaying sub-portions of a texture. An image has a link to a texture and

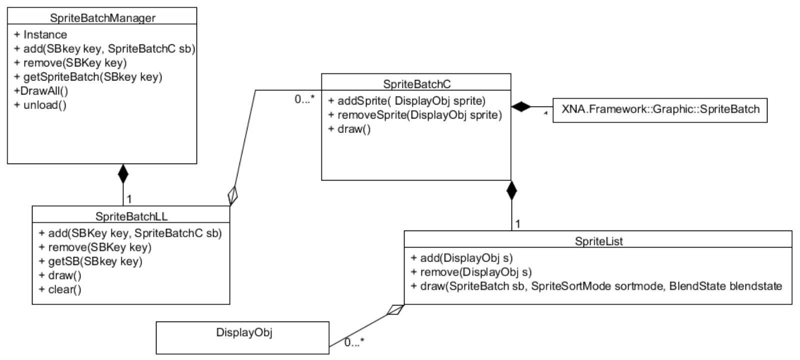
Robert Wascher
Final Project
Rectangle that points to a sub-region of that texture. These are stored within a linked-list with a corresponding ImageKey enum. The ImageManager controls access to this linked-list. No issues were encountered in implementing this manager.

Sprite Manager



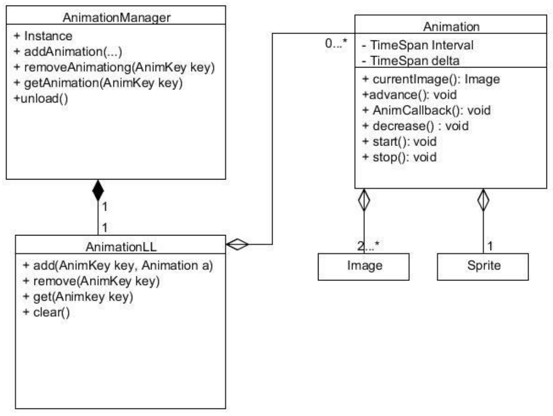
The SpriteManager class controls the loading, unloading, storing and access to various types of sprites. These sprites are stored in a linked-list with a corresponding SpriteKey enum. To be able to handle various types of sprites, such as regular Sprite, TextSprites, CollisionSprites, and ShieldSprites, I implemented a DisplayObj interface that these types implement. The DisplayObj only has 2 methods that need to be implement, both of which are draw methods, one of which is used by the SpriteProxy class(es). In order to be able to display the same sprite in multiple locations without creating a copy of it, I used a flyweight pattern. This is where the SpriteProxy and ConcreteSpritePos come into play. The SpriteProxy class is an abstract class that implements the DisplayObj class as well as having a reference to a Displayobj object. The ConcreteSpritePos class is a subclass of SpriteProxy that has a unique Vector2 position attribute. When a ConcreteSpritePos draws itself it calls draw(SpriteBatch sb, Vector2 pos) on its corresponding DisplayObj. The SpriteManager acts as a flyweight factory in that its getSprite() method returns a SpriteProxy.

Robert Wascher
Final Project
SpriteBatch Manager


SpriteBatchManager
+ Instance
+ add(SBkey key, SpriteBatchC sb)
+ remove(SBKey key)
+ getSpriteBatch(SBkey key)
+DrawAll()
+ unload()
1
SpriteBatchLL
+ add(SBKey key, SpriteBatchC sb)
+ remove(SBKey key)
+ getSB(SBkey key)
+ draw()
+ clear()
0...*
SpriteBatchC
+ addSprite(DisplayObj sprite)
+ removeSprite(DisplayObj sprite)
+ draw()
*
XNA.Framework::Graphic::SpriteBatch
1
SpriteList
+ add(DisplayObj s)
+ remove(DisplayObj s)
+ draw(SpriteBatch sb, SpriteSortMode sortmode, BlendState blendstate
DisplayObj
0...*

The SpriteBatchManager class controls the loading, storing, unloading, and drawing of the content of SpriteBatches. To get this done I implemented a SpriteBatchC class. This class has a reference to a SpriteBatch from the XNA framework as well as a list of DisplayObj objects. DisplayObj objects can be added or removed to/from a SpriteBatchC, which is able to draw each DisplayObj. The SpriteBatchC class will cycle through each DisplayObj in its SpriteList and draw them in the order they were added to it. These SpriteBatchCs are stored in a linked-list SpriteBatchLL. The SpriteBatchManager control access to this linked-list as well as has the ability to call draw on each SpriteBatchC in the linked-list.

Animation Manager

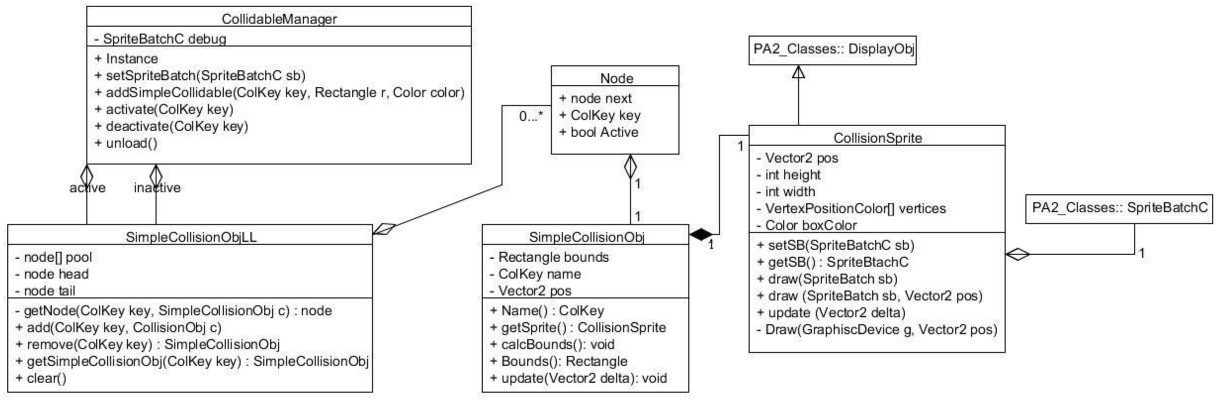


The animation Manager control the loading, unloading, storing, and access to various Animations. An animation is composed of 2 or more Images that are passed to it in an array of images. An animation has a callback function that is added to an Event to be inserted into the TimeServer. It also has the ability to start and stop its animation. Each animation is stored in a linked-list, AnimationLL. The Animation controls the access to this linked-list and can add, remove and retrieve a given Animation with its corresponding AnimKey enum. The biggest issue I had with this class was setting up the Callback function as I have never been exposed to callbacks until now. After prototyping a few callbacks, I took a shot at implementing this into animation. The Callback function changes the image of the Sprite that the animation has a reference to. The new image the Sprite references is the image following the previous image in the array of Images passed to the animation class. The Callback function then enqueues into the TimeServer an event that has a reference to the Callback function as well as a wakeTime equal to currentTime plus the interval specified in the creation of the Animation. The decrease method allows for the interval to be shortened when needed.

Collision System

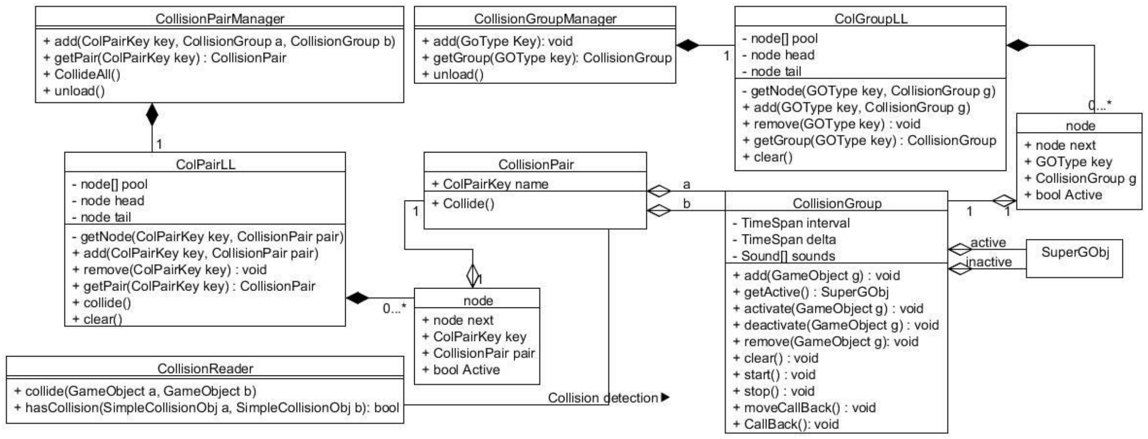
The collision manager controls the efficient detection and handling of collisions between various types of game objects. In order to accomplish this, I have implemented 3 managers: a Collidable Manager, a Collision Group Manager, and a Collision Pair Manager. These managers manage various classes that facilitate the detection of collisions within the game.

Collidable Manager



The CollidableManager class controls the creation and storage of collision objects. These collision objects, represented by the SimpleCollisionObj class, have data about the bounds of the object the collision object's position as well as a CollisionSprite to display the collision box. The CollisionSprite implements the DisplayObj interface from the Sprite System so that it can be added to a SpriteBatchC and drawn using the same commands as any other image. The CollisionSprite is drawn using XNA's linestrip type. This caused some problems as I had to learn how to use XNA's drawPrimitives() method. While this seems trivial, it took me several hours to figure out how to set up the basicEffect class correctly to display 2D graphics. Each SimpleCollisionObj has the ability to update its position by a Vector2. The CollidableManager creates each SimpleCollisionObj and its corresponding CollisionSprite and stores it in a linked-list. It also adds the CollisionSprite to a debug SpriteBatchC so that it can be drawn. To do this, I had to implement a method setSpriteBatch() in the CollidableManager to set the SpriteBatchC to be used for displaying CollisionSprites. The CollidableManager also controls the activation and deactivation of each SimpleCollisionObj. This allows the same SimpleCollisionObj to be used again if needed.

Robert Wascher
Final Project
Collision Group and Collision Pair Managers


CollisionPairManager
+ add(ColPairKey key, CollisionGroup a, CollisionGroup b)
+ getPair(ColPairKey key) : CollisionPair
+ CollideAll()
+ unload()
CollisionGroupManager
+ add(GoType Key): void
+ getGroup(GOType key): CollisionGroup
+ unload()
ColGroupLL
- node[] pool
- node head
- node tail
- getNode(GOType key, CollisionGroup g)
+ add(GOType key, CollisionGroup g)
+ remove(GOType key) : void
+ getGroup(GOType key) : CollisionGroup
+ clear()
node
+ node next
+ GOType key
+ CollisionGroup g
+ bool Active
0...*
1
ColPairLL
- node[] pool
- node head
- node tail
- getNode(ColPairKey key, CollisionPair pair)
+ add(ColPairKey key, CollisionPair pair)
+ remove(ColPairKey key) : void
+ getPair(ColPairKey key) : CollisionPair
+ collide()
+ clear()
CollisionPair
+ ColPairKey name
+ Collide()
a
b
CollisionGroup
- TimeSpan interval
- TimeSpan delta
- Sound[] sounds
+ add(GameObject g) : void
+ getActive() : SuperGObj
+ activate(GameObject g) : void
+ deactivate(GameObject g) : void
+ remove(GameObject g): void
+ clear() : void
+ start() : void
+ stop() : void
+ moveCallBack() : void
+ CallBack(): void
active
inactive
SuperGObj
node
+ node next
+ ColPairKey key
+ CollisionPair pair
+ bool Active
0...*
CollisionReader
+ collide(GameObject a, GameObject b)
+ hasCollision(SimpleCollisionObj a, SimpleCollisionObj b): bool
Collision detection ▶

Collision Group Manager

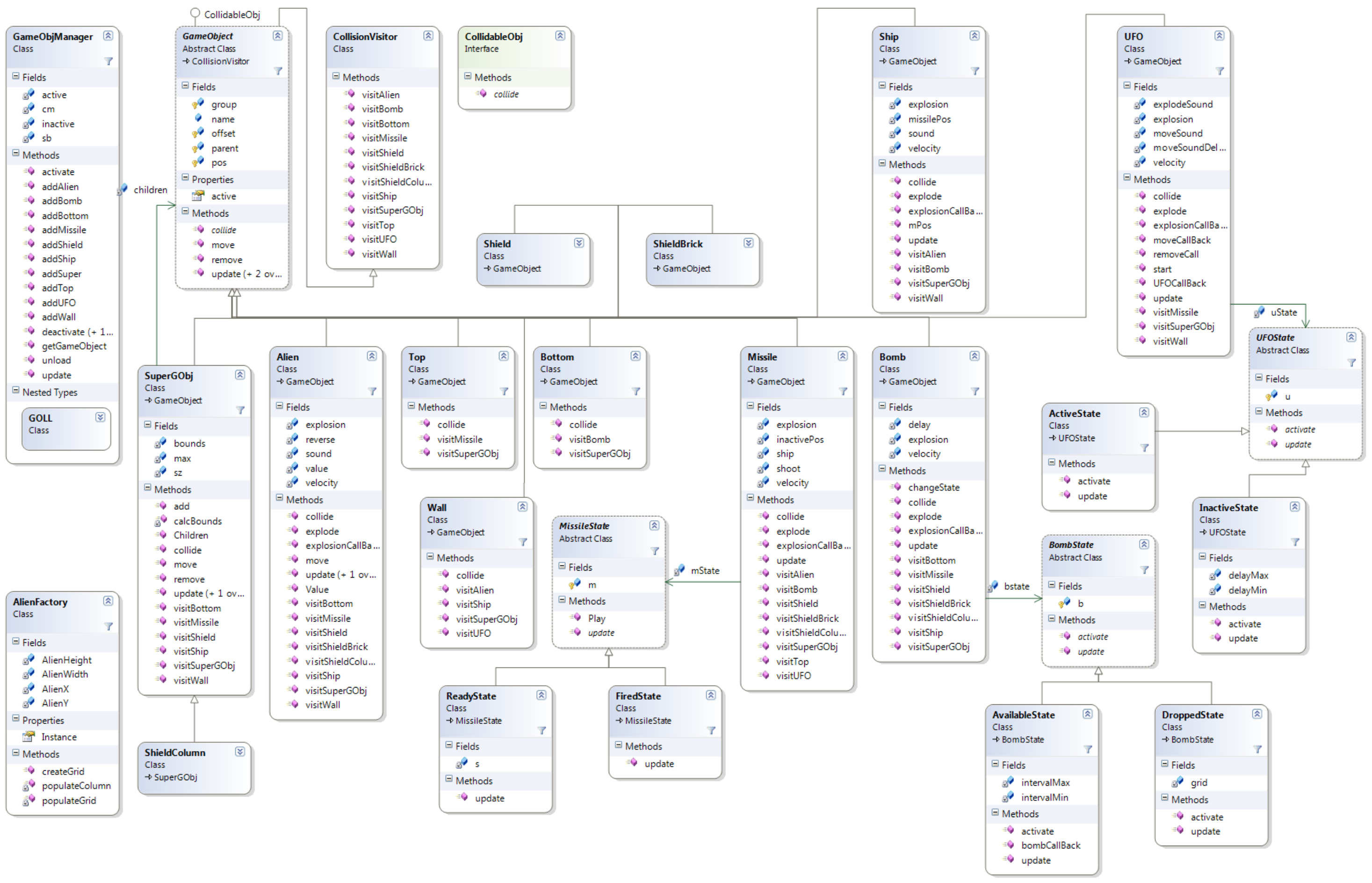
The CollisionGroupManager class controls the creation and storage of CollisionGroups. Each GameObject is added to a CollisionGroup, which allows them to be grouped with other like GameObjects. This is useful to only check for collision against GameObjects that make sense for a given group of GameObjects. The hierarchy implementation found in the GameObjects is useful because it eliminates the need for extra storage knowledge. Each group has 2 SuperGObjs where all the GameObjects for that group are stored. One SuperGObj is for active GameObjects, while the other is for those GameObjects that are currently inactive.

The Last 4 methods in the CollisionGroup class are for use by the Alien Grid. These methods control the lock-step march movement and the accompanying sounds. Start() starts the alien march while stop() stop it. They use the moveCallBack() method along with the TimeServer to accomplish this. The CallBack() method is used to decrease the interval between each march step upon an alien being killed. I put these methods in CollisionGroup because it allowed me to make one call and be able to affect each alien in the grid.

Collision Pair Manager

The CollisionPairManager class controls the creation and storage of CollisionPairs. A CollisionPair contains references to two CollisionGroups and has the ability to test of collisions between GameObjects found within each of its referenced CollisionGroups. To test for collision between game objects it uses the CollisionReader class. This class determines if two GameObjects collide. This is a pretty simple calculation because all collisions found within this game are axis-aligned.

Robert Wascher
Final Project
<u>Game Object Manager</u>



The GameObjectManager class control the creation and storage of various types of GameObjects. A GameObject can be any of the objects found in the game as well as the walls, the top of the screen, the bottom of the screen, and the bounding boxes used for early out collision detection. The GameObject class is an abstract class so only its subclasses can be instantiated. Each GameObject has a reference to a SpriteProxy and a SimpleCollisionObj. It also maintains the offset between these 2 classes. Some classes, such as the walls, top, bottom and SuperGObj, do not have a Sprite associated with it. In these cases the SpriteProxy reference is left as null.

In order to handle collisions between each type of GameObject, I used the visitor pattern. The GameObject class is a subclass of CollisionVisitor as well as implements the CollidableObj interface. These 2 classes allow the double dispatch of the visitor pattern to work. The CollisionVisitor class's methods are all virtual with a default that notifies the user that the method hasn't been implemented for that class. Each subclass overrides the visit methods that are relevant to that class. These visit methods allow each class to have methods that are specific to the type of collision that has occurred. When a collision is registered(specifics on this will be provided later), each class involved calls the appropriate method to get the appropriate outcome of the collision.

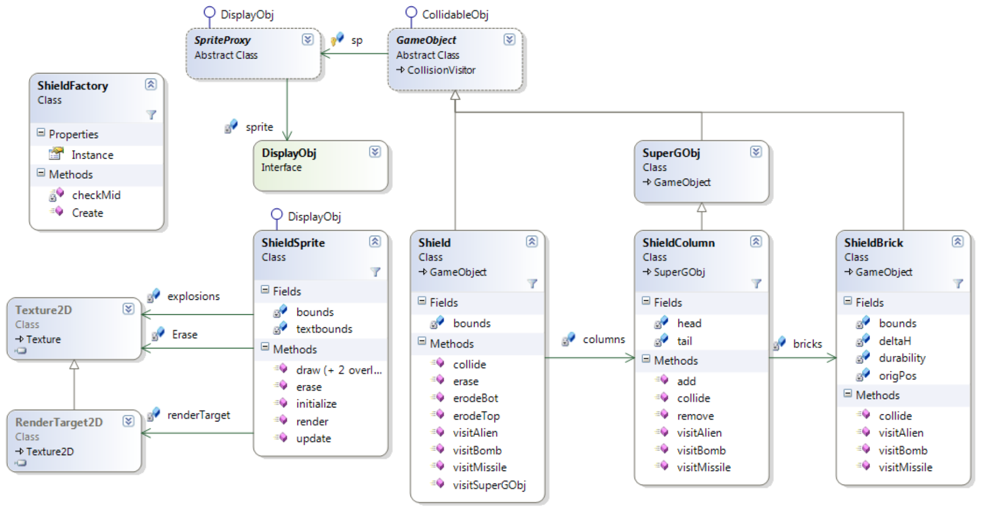
To allow for early outs in the collision detection, I implemented a hierarchy into the GameObject's subclasses. I chose to use the composite pattern for this because it is something I have used before and am comfortable implementing. Other approaches to hierarchy would also have worked though. Each SuperGObj hold a collection of GameObjects. GameObjects are added and removed from the SuperGObj as necessary. The SuperGObj bounds are recalculated each cycle to contain each of its children. The SuperGObj, which is a subclass of GameObject, allows for creation of a hierarchy as deep or shallow as needed. The hierarchy did cause some problems, mainly because I'm stubborn. Even after having several people tell how they made the mistake of implementing the hierarchy at too low of a level by implementing it on the collision object, I still decided to try to implement in on the collision objects. While I did get this approach to work, I soon realized that my code could be simplified by moving the hierarchy to the game object. This detour cost me a good amount of time only to come to the same conclusion as everyone else.

The GameObject Manager class stores each of the GameObjects. It also activates and deactivates GameObjects as necessary, which allows for GameObjects to be reused when needed. This becomes especially helpful when dealing with missiles and bomb as they are reused a lot within the game.

To facilitate the fact that UFOs, bombs, and missile act differently depending on their state, I used the state pattern to allow for simple state control. This is found with the 3 classes BombState, MissileState, and UFOState, as well as their subclasses. The bomb's state is either dropped or available. The missile's state is either fired or ready. The UFO's state is either Active or inactive.

I also created an Alien factory class that allows me to encapsulate the logic needed to create the Alien grid in the game. It has 1 public method and 2 private methods. Its public method, createGrid(), creates a grid and populates it with columns using the private method populateGrid(). This method uses the method populateColumn() to create and add aliens to each column.

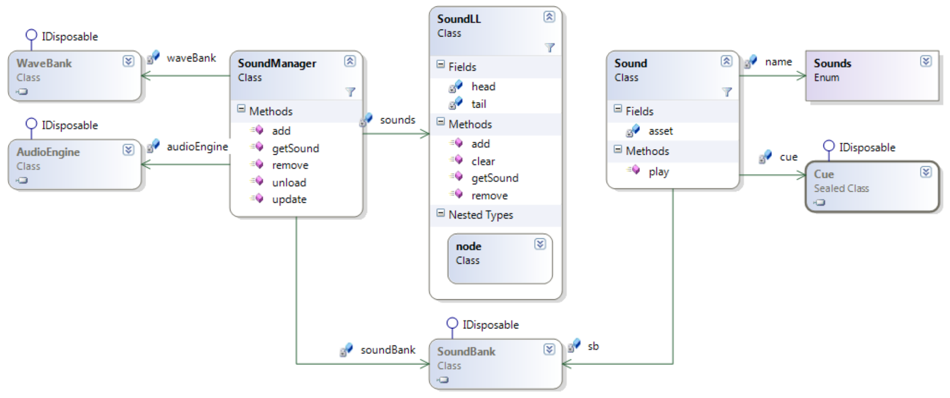
Robert Wascher
Final Project
Shield System



The shield system controls the logic of the Shield game objects in the game. The Shield class is composed of a shield sprite and a collection of ShieldColumns. The ShieldColumns, in turn, have a collection of ShieldBricks. I created a ShieldFactory to encapsulate the logic behind the creation and tying together of these different classes. It has one public method Create() that returns a shield object that has all of the needed objects tied in. the Create method uses the private method checkMid() to remove the middle 2 ShieldBricks on the bottom of the shield so that the shield has an appropriate shape. The ShieldSprite class makes use of XNA's RenderTarget2D class to change as erosion occurs. When render() or erase() is called the RenderTarget2D is stamped with the appropriate explosion or erase textures to create the effect of erosion. The shield class calls these methods in its erase(), erodeTop(), and erodeBot() methods. The erase() method is used when a ShieldBrick is completely destroyed to remove any excess particles that are left of the brick after the 3 explosions occur. The Fact that the erase texture has a fixed size makes this only work for a shield that is 4 bricks wide by 4 bricks tall. The erodeTop() and erodeBot() methods are used to cue erosion when a bomb or missile hits the shield. The ShieldColumn class act very much the same way as the SuperGObj class does. It basically is just a bounding box for ShieldBricks, with a little added logic to only check for collisions at the top or bottom of its children. The ShieldBrick class is the smallest portion of the shield. It has a durability to allow it to be hit multiple times before being destroyed. With each hit, the brick shrinks vertically to simulate deeper missile/bomb penetration. On each hit the ShieldBrick calls the Shield's appropriate erosion/erase method to simulate shield decay.

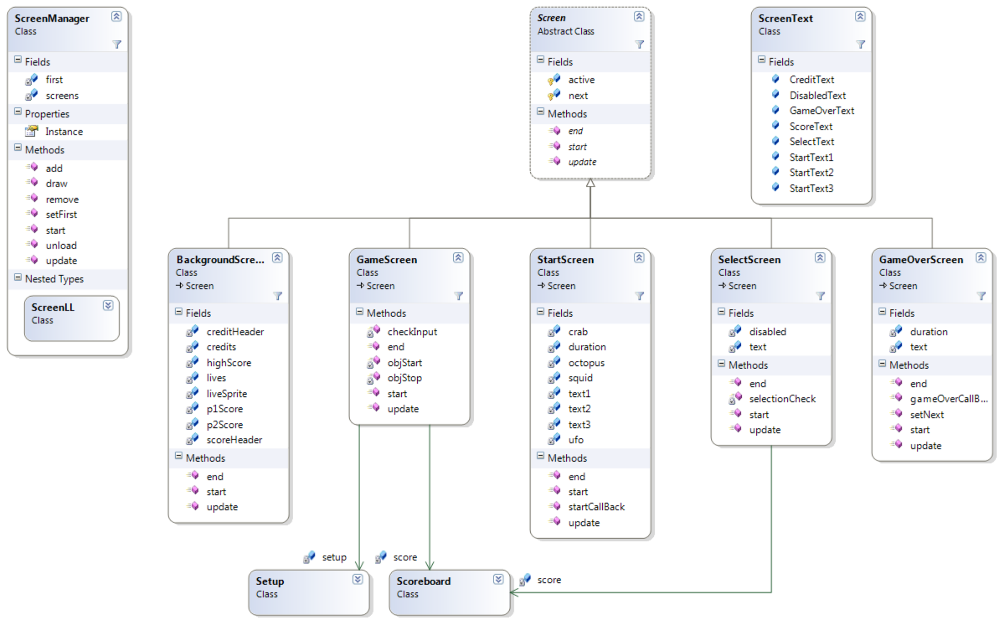
Robert Wascher
Final Project

Sound System



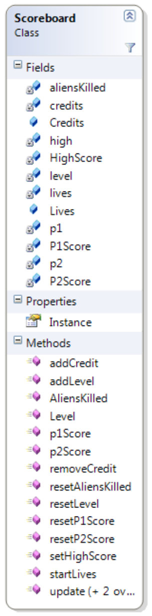
The Sound system controls the storing and playing of sounds. To do this, I made use of the XACT audio tool. The SoundManager class controls the storing of sounds and the updating of the AudioEngine. It has references to the Audio Engine, Wave Bank, and Sound Bank that are created using XACT. Each Sound is stored in a linked-list SoundLL that is owned by the SoundManager class. The Sound class has an enumeration, a string containing its asset name and the ability to play a sound using the Cue it retrieves from the SoundBank. To play a sound all one has to do is call getSound() from the SoundManager (assuming the sound has been properly loaded), and call play() on that Sound class.

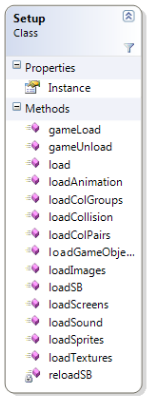
Screen Manager



The ScreenManager class controls the storing and updating of the various screens used in the game. It stores these screens in a linked-list ScreenLL. Each type of screen is a subclass of the asbstract Screen class. This class has the common methods and fields of all classes. Every screen has a start(), end() and update() method. These allow the screen to begin being displayed, end being displayed and update what is being displayed. Each screen also has a reference to the screen that follows it. This allows it to call the start() method of the next screen in its end() method. Because the next screen is passed to the Screen in its constructor, the screens have to be created in reverse order. Also because of the fact that the game cycles, I needed a method in the GameOverScreen to set its next screen so that the cycle can be created. The background screen contains a bunch of SpriteProxies for the various things that are always displayed on the screen. These include the scores at the top as well as the lives and credits at the bottom of the screen. The StartScreen also just displays a bunch of SpriteProxies, these displaying the controls and point values of each type of alien. It, however, also includes a timer event and a Callback to limit the time it is displayed on screen. The SelectScreen has a SpriteProxy prompting the user to choose between 1 or 2 players. It handles the input for this option as well as the input for adding credits. The GameOverScreen has a SpriteProxy displaying "Game Over" as well as a timer event and CallBack function to limit the time it is display on screen as well as updating the high score if necessary. The GameScreen does not contain any SpriteProxies to display however, it does have methods to load and unload all the objects needed for gameplay. In its update method, it reads all relevant user input, updates all the necessary system/managers that handle game logic, as well as drawing all the content needed.

Game, Setup, and Scoreboard





The game class has very little in it. It calls Setup's load() method in loadContent() it calls update() on the TimeServer and ScreenManager in update() and calls ScreenManager's draw(0 method in draw(). Setup contains a bunch of methods to load various types of objects. Basically it can load appropriate content into each of the games managers. Its load() method loads all the constant content in the game, such as textures, images, sprites, sounds, etc. The gameLoad() and gameUnload() methods load and unload all of the game content, such as Animations, collision objects, game object, collision group, collision pairs, etc.

The Scoreboard class handles all of the scoring logic for the game. This includes updating a players score, updating the number of credits, and updating the number of a player’s lives. To accomplish this it holds a collection of ints for the various types of data as well as methods to update these ints. It also hold references to the TextSprites that are used to display each of the types of data so that what is displayed on screen can be updated to match what the Scoreboard holds.