Idea description

At first, we agreed that we would do some sort of a space or cave shoot'em up game. Whether it would be a horizontal or vertical single-player scroller or four-player cave shooter (eg. Wings) was left open. At first, we weren't really familiar with the XNA framework nor the C# language so the game took shape while making it.

Eventually, what turned out was a two-player survival game with a static camera and swarming enemies from all directions. The simple, and quite intuitive, controls we took from an excellent downloadable game called Geometry Wars for the Xbox 360. The left thumbstick controls the movement of the ship and the right stick controls direction of shooting, independent of movement. This proved out to be easy to learn yet challenging at the same time.

Challenges

Our first challenge was to get the ships moving in a proper manner with the keyboard. After that, we decided to make it movable with the Xbox pad controller but our first solution for the moving wasn't working as well as we expected. The problem was that when the thumbstick went back to it's center position it would return so small X and Y coordinate values that our ship (well, the texture) would spin awkwardly. We fixed this by using the length of the vector provided by the thumbstick and assigning it a minimum value. This fixed the texture rotation as well as kept the feel for the analog controls.

After the controls felt alright the next thing on the list was collision checks. This sure is an important part of almost every game. We did this by giving every object in the game their own bounding sphere with a class dependant radius. Every time the object's texture moves, its bounding sphere moves as well.

The actual algorithm for checking collisions proved out to be quite difficult as well. We put all the enemy ships and the players' projectiles in their respective lists and looped through both to check if they were touching each other. Adding the player's own ships to the mixture (so they could die) wasn't pretty since the ships were independent objects and not in a list. Due to the algorithm's complicated nature we couldn't get rid of randomly disappearing enemies but since it at least didn't crash it felt decent enough.

Most of the Thursday was spent on adding the main menu. We got an example source code for it from another group. Integrating it to our game and understanding how the menu works took a lot of time and effort. In the end we decided to strip it down from about 300 lines of code to about 80 and got it working. The actual game loop just checks whether it is in menu or game state and calls the respective update and draw methods. The menu class takes input from the Dpad of the controller and draws the background and menu options. We would have liked to make the menu usable with the left thumbstick as well as the Dpad but scrapped it.

The final challenge was to make the resolution scaling work with Xbox 360. Scaling worked fine with the PC but was horrible on the Xbox. We had used three different classes for determining the screen's width and height in different spots of the code and some weren't working properly on the Xbox. After, we used just one way to find out the width and height our game looked somewhat the way it was supposed to be. Small parts of the area were still stripped off but this seemed to be a problem with the class room's projector.

Technical description

The Xbox gamepad's thumbstick gives an unit vector which tells the direction where the ship is moving and that is multiplied with a constant to get more speed. The right thumbstick controls the firing direction, which is calculated the same way as moving but new projectile object is created and it has constant speed, direction and time to live. Ship class controls the fire rate and other ship features.

The enemies have certain spawn rate and they spawn outside of the screen area on random position on a circle. Spawning position is calculated so that first the screen center is measured and then a circle is calculated that is a bit bigger than the screen size. The enemies are instances of the enemy class and each enemy calculates the distances to both ships and checks which of them are closer and the enemy will go to that direction. The direction can be calculated by creating an unit vector and going to that direction, meaning that the unit vector is multiplied with a constant that illustrates speed.

Collision detection works with bounding spheres and they have a built-in method in them that checks if a bounding sphere intersects another. Our collision algorithm first updates all the projectile's position. Then it independently checks whether either player's ship is in contact of an enemy. If they are, they are made invisible and uncontrollable. Next there is a nested loop (one loop inside another) that checks the collision between projectiles and enemies. If they do hit, then both instances are destroyed and removed from the lists. This part was the problematic one since deleting a list member while iterating through it can lead to problems. Getting it not to crash wasn't that hard but in the end after lots of debugging effort and rethinking we still weren't able to get it working properly.

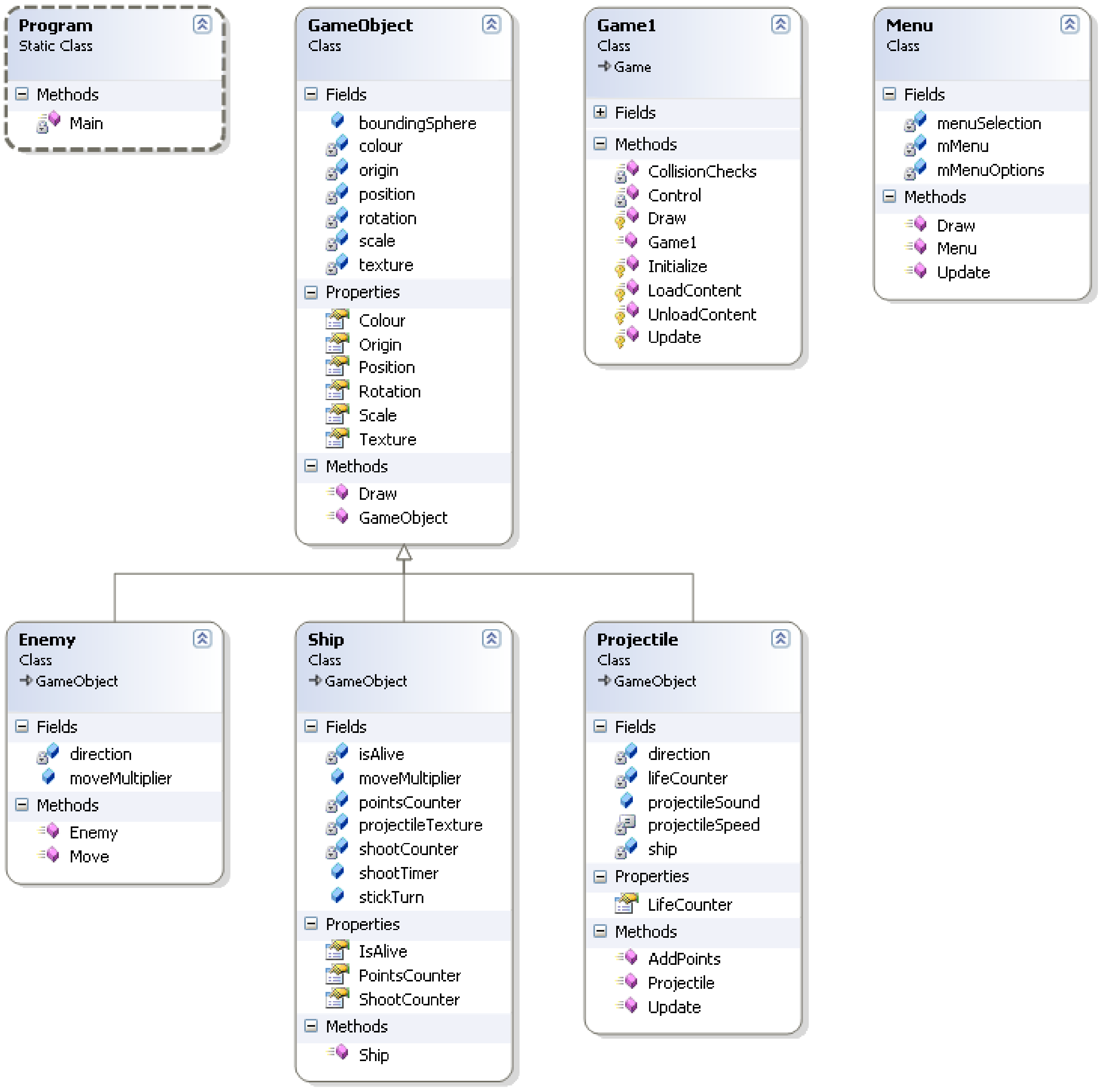
Every projectile knows which ship shot it so adding points to the counter was as simple as adding an amount to the shooter's total. We felt that adding 100 points for every kill seemed too much like a kill counter, so we gave the player 97 points for every kill instead.

Conclusions

The codecamp was a fun experience to take part in we both are looking forward to attending these events again. The XNA Framework made it fast to get into the actual game making instead of focusing on how to handle the passing of time in the game, how to create a game window and those sort of things.

Planning proved out to be a very important part even in a two person project. Adding features took a lot of time when we had to rewrite half the code for it. IntelliSense in the Visual Studio smoothened the C# learning curve a lot and made it unnecessary to have a guide book opened all the time.

A scrolling and/or zooming camera coupled with some particle effects would make the game quite a lot more visually attractive. On the gameplay side, some kind of a dynamic point scoring system and a high score list could hold the player's interest a bit longer than it does now. The final game is somewhat simple and limited, but with just the two of us we are pleased with how the game turned out.



Picture 1. The game’s class diagram. GameObject serves as a base class from which all the other objects are inherited.