Document Information

Document Authors

Author	Position
Aaron Cox	Teacher
Jarrod Dowsey	Programmer, Art.
Heath Burnett	Programmer, Sound, Art.
Richard Murray	Programmer.

Related Documents

- ***Game Design Documentation***
- ***Project Scope Document***
- ***Work Log***
- **Neon Commander - Game Test Results**

Table Of Contents

Introduction

Neon Commander is a single player Space Shooter in which you fight off a variety of enemies to survive as long as possible to put yourself at the top of the leaderboards with a new High Score. Refer to Game Design Doc for full details.

Platforms

Target Platforms

The target platforms for Neon Commander are the PS Vita and PC.

Technical Specifications

PS VITA

For the PS Vita the screen resolution is set to 955 x 544 so that is the screen resolution we will be using. Restrictions with the PS Vita has caused us to go with sprite animations over a particle system so the game remains efficient. The controls include both analog sticks, one for movement and one for shooting. Other buttons will be used as menu navigation.

PC

For the PC the screen resolution will remain 955 x 544 so the game still has the same feel as the PS Vita version (we don't want the PC having a biggest sight radius for gameplay reasons such as where enemies spawn and how many you can see on the screen at once). The controls will be keyboard controls, although they will not feel as fluent than analog stick movement it will still have a good enough feel for consistent gameplay compared to the PS Vita.

Build Process

PS VITA

Deployment is currently done via developer tools (PSM studio), sent via USB from the PC to the PS Vita locally. This is considered a debug build, as a release build would have to be distributed via the Playstation Network.

PC

The project is compiled in Visual Studio, with the executable contained within the release build folder. This group or 'package' of folders contains all the necessary assets and can be copied to other directories with the executable still working.

Code Specification

Class Hierarchy

The Menu State is a main hub for the game. Although it does not load all of the content for the project it acts as the area that everything is accessed from. The Help and Leaderboards States are sub states of the Menu State and between them contain the Instructions, Controls and the Leaderboards.

Next up also coming from the Menu State is the Play State. This is where the core of the game happens. On initialization the content for the game is loaded and then updated as the game progresses. The collisions also are checked within the playstate, each enemy, bullet and player have their own collision circles that are checked against each other to determine collisions.

The GUI is operated through the use of either the GamePad/PS Vita Controls or through the use of the keyboard. Up, Down and Enter(X for PS Vita) are the simple controls for navigating the Menus. The Mouse can also be used to navigate through the menus.

The saving included in the game are two options in which save to a binary file and are then loaded back in on game startup. These two options are player colour and the game volume.

Third Party Libraries

Monogame/XNA framework.

Tools

The Textures in Neon Commander were created in photoshop.
The sprite animations were created through a program written by Aaron Cox in XNA to create a particle effect and save each frame of it.

- The sprite images save to a png file to then be loaded in again as an animation.
- The frames are positioned right next to each other.
- Each frame is 64x64 pixels.

Coding Standards

- Member variables are to be defined as “m_”.
- Every Function written is to have comments above it explaining what the function does.
- All words in a class name start with capital letters.
- All words in a variable name start with capital letters, except for the first word, which is in lower case.
- Delta time is always defined as gT, as a GameTime class.
- All game objects are separated into classes.

Assets Specifications

Asset formats - needs to be done

The textures for Neon Commander are in a .PNG format but are convert to .xnb files for use on multiple platforms.

All animations are stored in larger texture atlases, using a simple UV coordinate traversing object. These atlases are .PNG just like the rest of the images we use. The only animations we currently have within the game are for the enemy death explosions, that are used in place of the particle system.

All sounds are wav files, with no standard bit depth or sample rate. All sounds are under 16 seconds in length, with the music files being the longest. All music files have been specifically chosen because of their ability to be looped, or edited so that it can be looped. This saves on limited memory within the Vita.

Asset Pipeline

All of the images for Neon Commander are created or edited through the use of photoshop. In the case of the sprite animations we used a program mentioned above in tools thats creates an explosion and saves each frame of the explosion side by side to create a sprite sheet.

Through the use of the XNA pipeline, at compile all content files are converted and stored as .xnb files for use on all platforms. An .xnb file is a binary file that is usually a compressed version of the full content file. The conversion of files to .xnb allows us to use these files on the PS Vita.

Folder Structure and file naming conventions

The project is stored in a folder called PSMPROJECTS, and everything necessary for it to function is located in its subfolders. It consists of three sections; the MonoGame-develop folder, which hosts the MonoGame source files, MonoGameProjects, where the NEON COMMANDER

Visual Studio solution and all files necessary for it to compile and run is stored, and MonoGamePSMProjects, where the Playstation Mobile Studio solution is found and is where XNB files are copied to.

Most of the work is done in the MonoGameProjects folder, where the MonoGameProjects solution file is found in the root. The NEON COMMANDER folder is found here, which is the main project. In this folder is the subfolders TestGameOne and TestGamePL. TestGameOne hosts the .cs files which contains the game's source code, and compiled builds are found in the /bin folder.

TestGamePL is the content pipeline. The original assets are found in TestGamePL/TestGamePLContent in the respective folders Fonts (for SPRITEFONT files), Images (for PNG files) and Sounds (for WAVE). At compile time, the xnb files created with the pipeline are sent to respective subfolders Fonts, Images and Sounds in the TestGamePL/TestGamePL/bin/PSM/Content directory, and are then copied into MonoGamePSMProjects/NEON COMMANDER/TestGameOne/bin/debug/TestGameOne-signed and its sister folder, TestGameOne-unsigned.

Save File Formats

For each custom file type your game loads and saves, specify the format of the filetype.
for example,

File Format

We save out two different files during gameplay, both as binary files. We save and load our options, to change player colour, the game volume and whether the game is fullscreen, and we save the top 10 high scores.

The options binary file is used for the players colour when they play and for the overall game volume. The structure that contains all this information is called OptionsObject and contains, in order, three floats (red, green, blue) for colour, another float for volume and a bool for fullscreen. The first 12 bytes relate to the colour (4 bytes per float), the next 4 are for the volume and the final byte is for fullscreen. Both files use the custom file extension .emu, a namesake of a group name “Dancing Emu Studios”.

The high scores file is used to keep track of the players highest scores, saving and loading the top 11 scores ever achieved. The structure that contains the high scores information is called HighScoresObject. This object contains a list of HighScoreEntry’s, which is another structure containing a string (name) and int (score). The list of entries is sorted by descending score, so highest score is first. The file contains 11 scores and 11 names, with the order being name, score, name, score… In total, the file is 88 bytes long.