

OPERATION PLANET X

BY: TEAM JETPACK

Design Document

**William Hu, Robert Olson, Stephen Mitchell,
Nivardo Melchor, and Abi Rajive-Shanker**

Table of Contents

Section	Page
Introduction	4
Overview	5
Artwork and Inspiration	5
Game Specifications	6
Interactions	7
Player and World	7
Player and Enemies	7
Characters	8
Player: Mr. Astronaut	8
Enemy 1	10
Enemy 2	11
Sample Level Design	12
Rationale for certain features	13
XML File Format for Level Storage	15
Movement and Control	16
Technical Specifications	16
Class Design	17
Project Timeline	27
Project Tasks	29

Introduction

A top secret U.S. government research base requires an extraordinary mineral to complete a never before attempted experiment that may change the world as we know it. Unfortunately, this mineral is unattainable on Earth. Extensive research pushes top secret officials to believe that this strange and wonderful mineral, Nivablium Robphenite, can be found in only one location in the known realm of the universe: Planet X. No human has been to this planet before.

Mr. Astronaut, controlled by the player, has been chosen to lead and execute Operation Planet X. He must go to Planet X, retrieve a sample of Nivablium Robphenite, and safely return to Earth. However, it's not as easy as it is described to be. Top-secret officials reveal that the player may or may not be alone when he arrives at Planet X. For his safety, he will be supplied with a weapon, Bazooka Spacegun 3011, and a refueling jet pack. The player must use these wisely in order to protect himself from attackers and the unknown circumstances of Planet X. Once the player retrieves some Nivablium Robphenite, he must return back to his space shuttle as quickly as possible. The player expects no other difficulties or setbacks at this point.

Upon returning to the original location of his space shuttle, the player has completed Level 1. He returns only to find that Planet X inhabitants have unexpectedly taken his space shuttle. There are only a few remains of his shuttle left behind that indicate its original location. He must now take on a second mission - search for an enemy space vehicle that he can steal and use to safely transport him back to Earth. The search for this enemy space vehicle is the goal of level 2.

Overview

Mission Planet X will be a one player, 2D top down shooter, Windows-based game. It will use the XNA Game Studio 4.0 framework. The game will be based in outer space and will contain many futuristic and space-related graphics and features. For instance, all enemies encountered by the player will be an alien form unknown to mankind; and the background terrains and obstacles that the player interacts with will also be foreign to Earth. A user will control the player using only the left, right, up, down arrow keys for movement and the mouse to choose direction.

At the start of the game, a cut scene along with some text will describe the mission instructions to the player. This will include game rules and mechanics as well, such as how to control the player and his features. Once this is complete, level one will begin, as will the adventure on Planet X. Upon completion of level 1, a second cut scene will play. This cut scene will explain to the player what happened to his space shuttle while executing his mission - Planet X inhabitants stole it. This cut scene will describe mission 2 level instructions.

The idea of this game is that for each level to have a different goal, but for each goal to be generally similar to one another. A player wins by completing the given mission objective during each level. For efficiency purposes and considering our time constraint, we want to incorporate as much of the implementation and graphics used in level one, in level two as well, and it successive levels as we develop them.

Artwork and Inspiration

The inspiration for this game was entirely imagination. A team member requested the theme for this game to be outer space. Combining this suggestion with our preference of a 2D, top down shooter game rooted the creation of Mission Planet X. Over a period of two weeks, we discussed various characters, their qualities, their weaknesses, potential level goals, game rules and mechanics.

There will be no outside artwork used in this game. All artwork will be developed using Adobe Flash CS5, and MS Paint. The inspiration for our characters, their physical features and attributes also came from numerous team meetings and brainstorming creative ideas that we thought would be original, fun and interactive. Multiple sketches of our characters were created. One of each was selected to translate into a computer graphic sprite.

For the music assets of our game we are thinking of having our own made sound effects for things like shooting, jumping, flying..etc. For the level background music we will find open source music online that goes with our game.

Game Specifications

With the main goal of the game being that the player is to infiltrate Planet X in search of a mineral to bring back to earth. Each level will have a mission objective that will have to be met in order to move on to the next level. The player will be able to shoot, jump, and fly around within the level in search of its objective while encountering enemies who will attempt to stop him from completing his mission.

Mission Objectives– Level 1, locate and collect samples of special mineral to take back to earth. Level 2, locate and steal enemy ship to get back to earth.

Dying- Player could die by getting choked or shot multiple times and by falling into a pit. When getting choked or shot the player can take up to 5 attacks until he is dead assuming his health bar is full. Stepping into a pit however will cause immediate death to the player.

Bazooka Spacegun 3011- Player always carries around this gun which shoots bullets and never runs out of ammunition. There are also power-ups which can be picked up in the game that give the gun added perks.

Power-ups- Two types red and yellow. Red allows for instant one shot kill and yellow allows for the players gun to have a faster shooting rate.

Enemies- Two kinds of alien enemies one more intelligently advanced than the other. One uses his tentacle eyes to attack the player and the other one uses a gun.

Jet Pack- Is used to fly or jump over objects or obstacles within the world. For example jump over a pit, enemies or enemy attacks. Player will be able to move faster while using the jet pack but will not be able to shoot.

World Collisions- Visible player collisions within the world are that he will not be able to walk on rocks, walls, or enemies.

Health Bar- Every time a player gets shot or choked he will lose a certain percentage of health. When in FROST MODE player will have unlimited health.

Fuel Bar- Limited amount of fuel but will auto regenerate on its own. When in FROST MODE player has unlimited fuel.

Interactions

Player and World

The player will be able to interact with the world by using a jet pack which enables him to jump over pits enemies or enemy attacks. When using the jet pack the player will move a bit faster than if he were walking and will not be able to shoot. The level's edges will be bounded to a specific size limit to which the player cannot pass. The player can either walk or fly around in the world as long as his jet pack doesn't run out of fuel. When the player comes across the edge of the map there will be a rock-like wall preventing him from going any further and making him aware that it is an edge of the map. There will be two types of enemies within the map that will attack the player. The player will be able to kill these enemies by using his gun. The player will also have the ability to pick up either a yellow or red bullet power-up that will appear randomly when an enemy is killed. The red power-up will allow the player to kill future enemies with one shot. The yellow power-up will give his gun a faster shooting rate. These power-ups not only have a specific color but also a distinct design that goes with what the power-up does. Power-ups have a time limit of one minute. Picking up multiple power-ups of the same type will not cause any extra reactions with the gun.

Our player will be walking around Planet X in search of a mineral or specific item declared to him in the mission objectives. This mineral or object will be placed in specific locations relatively far away from where the start position of the level is. Within the level there will be pits that the player will not be able to walk on without dying. In the world there will also be a pile of impassable rocks lying around to which the player will have to walk around. In other areas within the map the player might need to use his jet pack to fly across a big pit in order to clear it and continue his search within the map.

Player and Enemies

There will be two types of enemies among these there is the enemy with googly-eyed tentacles. Whenever the enemy comes within a specific close range to the player he will begin to do his attack. This attack consists of the tentacles stretching out further away from the enemy and reaching for the player. Whenever the tentacles come in contact with the player they begin to choke the player for a few seconds and then release him. Googly-eyed character has a fairly more simple AI since he will just be chasing the player around the screen in order to perform his attack.

The other type of enemy is an alien-looking one with a more advanced attack weapon. This enemy will shoot at the player with his gun that shoots fiery balls of gas. The player will have to avoid getting shot by this enemy and be able to shoot back to kill him. The alien with big gun has a smarter AI since it stays outside of a certain range from player and tries to avoid getting hit. Whenever the player dies he will return back to the starting point and lose all his power-up if he had one active. In level 1 for example he will return back to the same position where the level started next to the ship

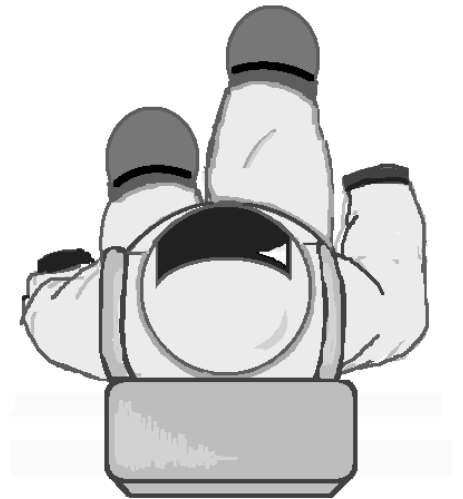
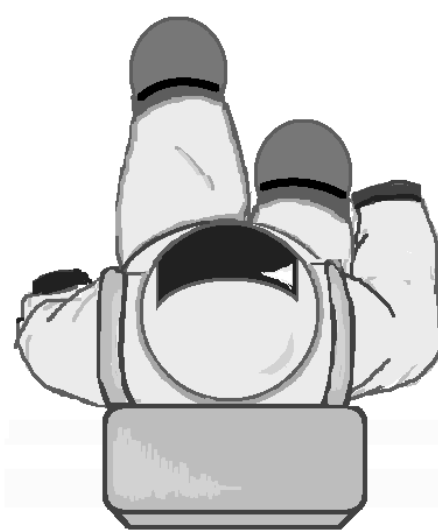
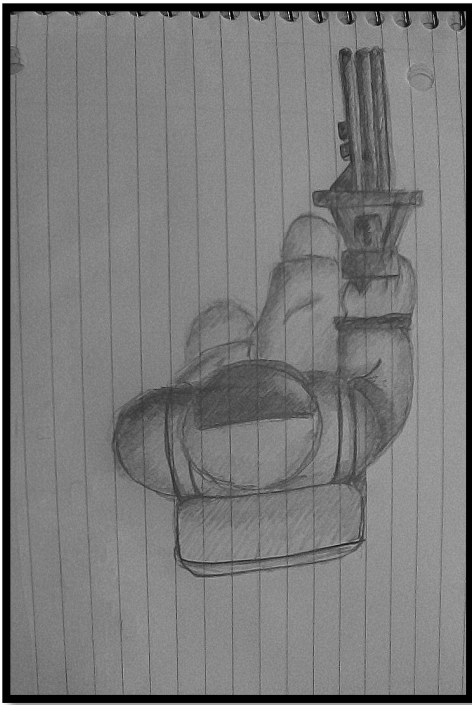
he landed in when he arrived at Planet X. Player will have unlimited continues.

Among these characters there is a possibility of having a third enemy in order to add a bit of difficulty to the game. This will be a flying enemy that will fly from an off screen location ignoring all tile rules to strike the player inflicting damage and fly off screen. Player will not be able to jump it or shoot it down only dodge its attacks.

Characters

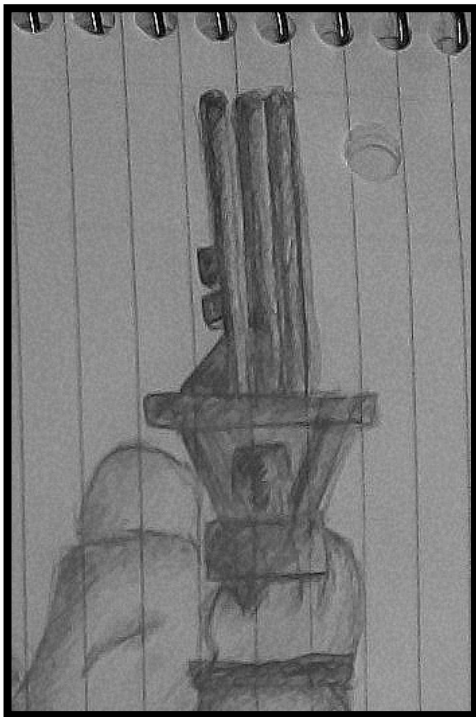
Player: Mr. Astronaut

This is the main player that a user controls. As he moves, his right and left legs will alternate moving inward and outward. This repetitive leg movement will give the appearance of him walking forward; when executed in reverse, it will give the appearance of him walking backward. No other body part will have movement.



Furthermore, this character wears a jet pack on his back. This feature will allow him to essentially jump a few feet or hover over an object while moving in any direction. This ability will become useful when avoiding aliens on Planet X. When the jet pack is in use and the player is in the moment of hovering, the jet pack itself will visually not appear any different. Instead, the entire sprite will be magnified slightly to show that the player has moved up, and a second “flame” or “boost” sprite will appear under the player to show that there is some sort of exhaust involved with the use of the jet pack.

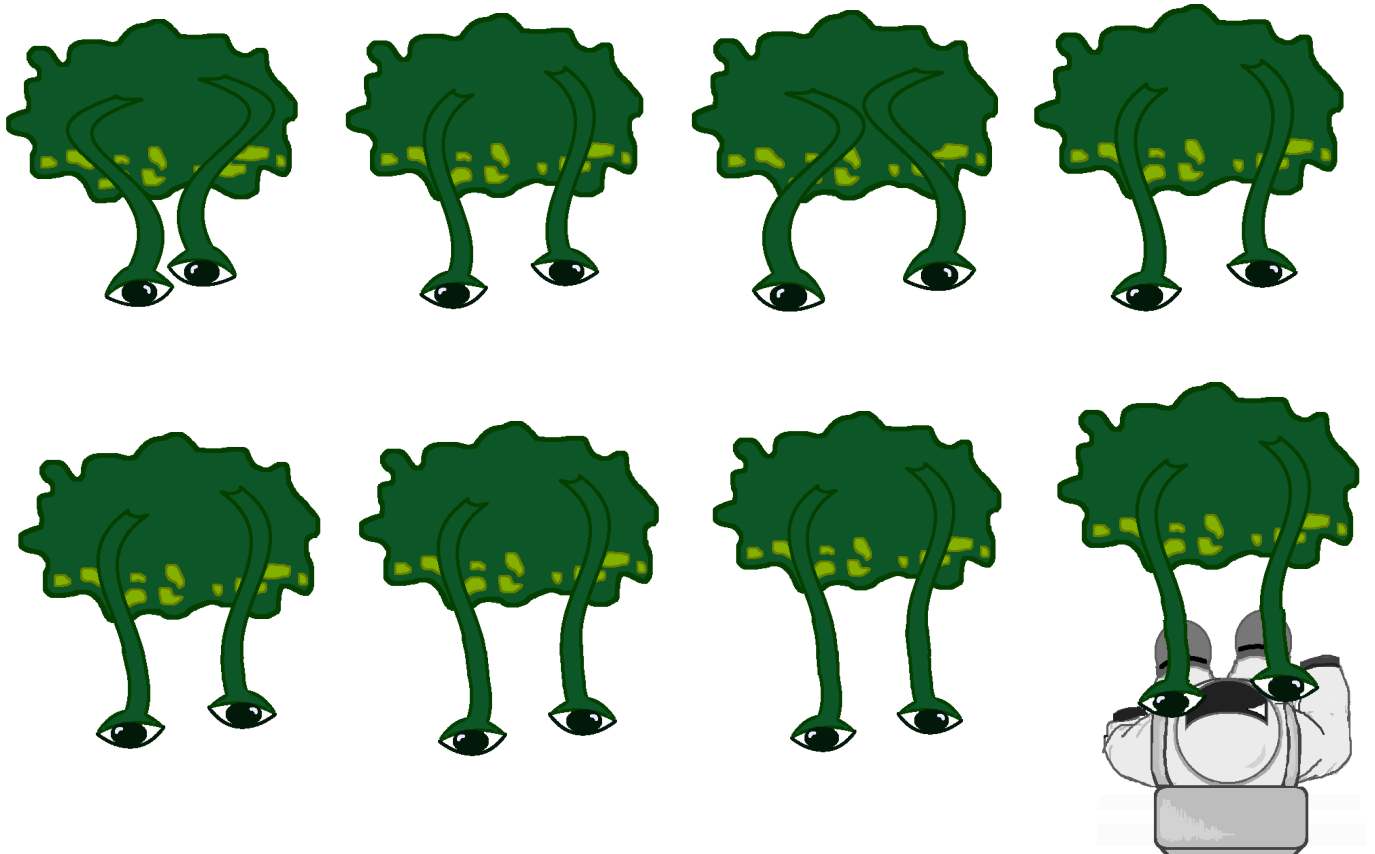
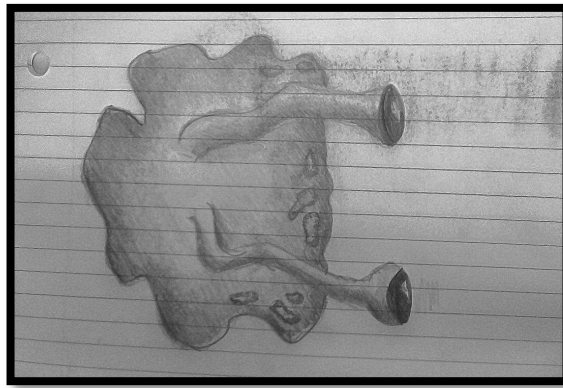
This character is armed with a weapon. He will be holding on to this weapon at all times during the game. This weapon will fire bullets that can be used to eliminate enemies that attack him. Moreover, there exist power-ups on Planet X that the player can pick up by walking over them. These power-ups provide the player with two different types of shots he can fire at an enemy.



Enemy 1

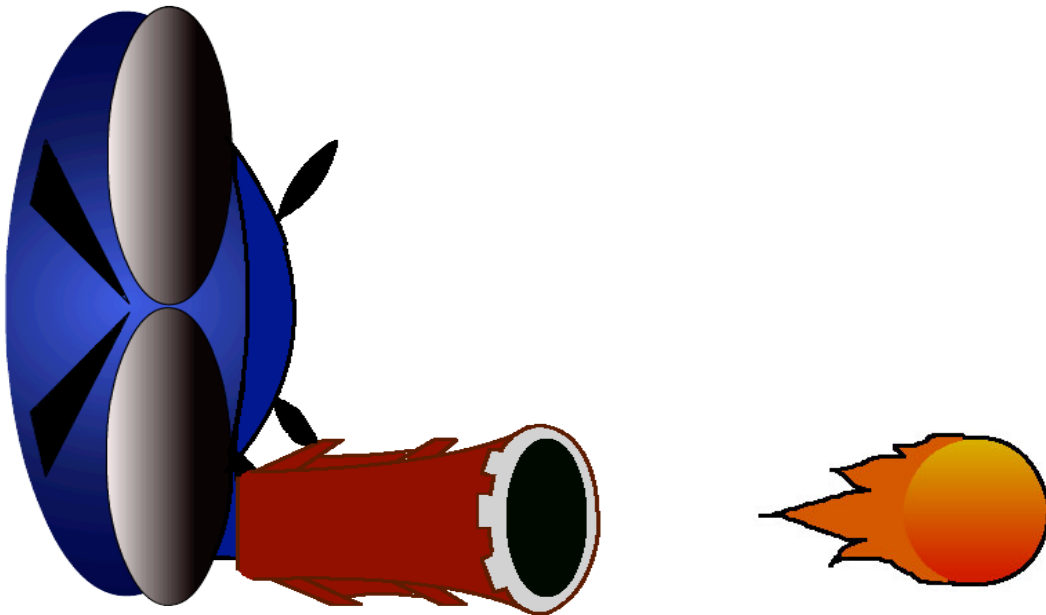
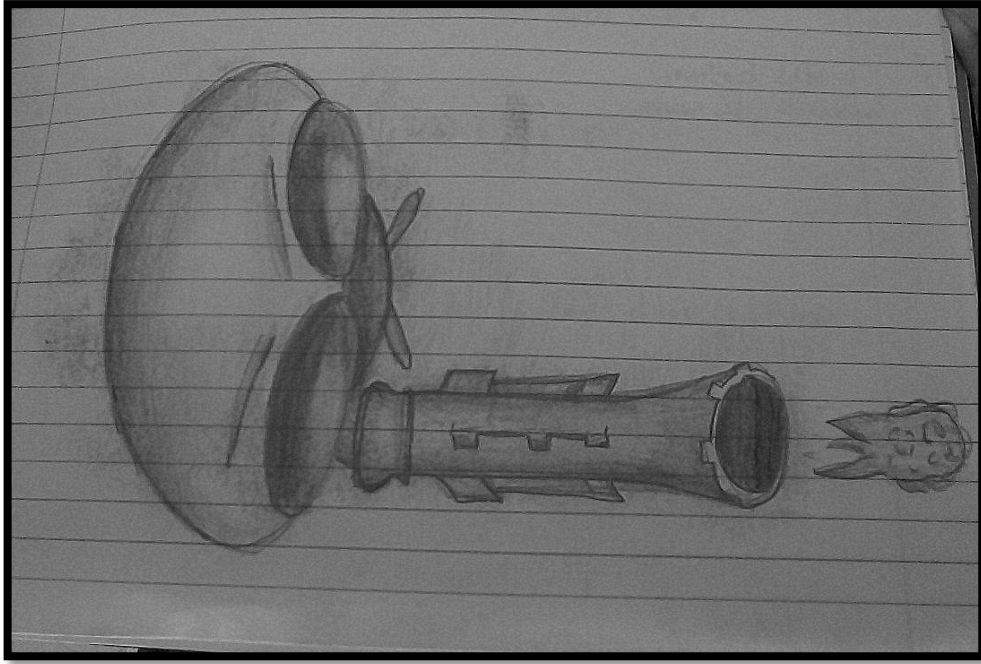
This character is the googly-eyed enemy that is a predator on Planet X. As this enemy approaches the astronaut, his googly tentacle-like eyes wobble from side to side. This is illustrated by a series of sprites on page 10, row one. If the main player, Mr. Astronaut, comes within this enemy's "vision" then this character will reach out with his googly-eyed tentacles and strangle the player for a few seconds. An example of this is illustrated on page 10, row two. The player will be released and lose a percentage of his health every time he is strangled. Further details on the implementation of this character's "vision" will be added at a later time.

The main player can use his jet pack to jump over this enemy and avoid it. The player can also shoot this enemy to destroy it.



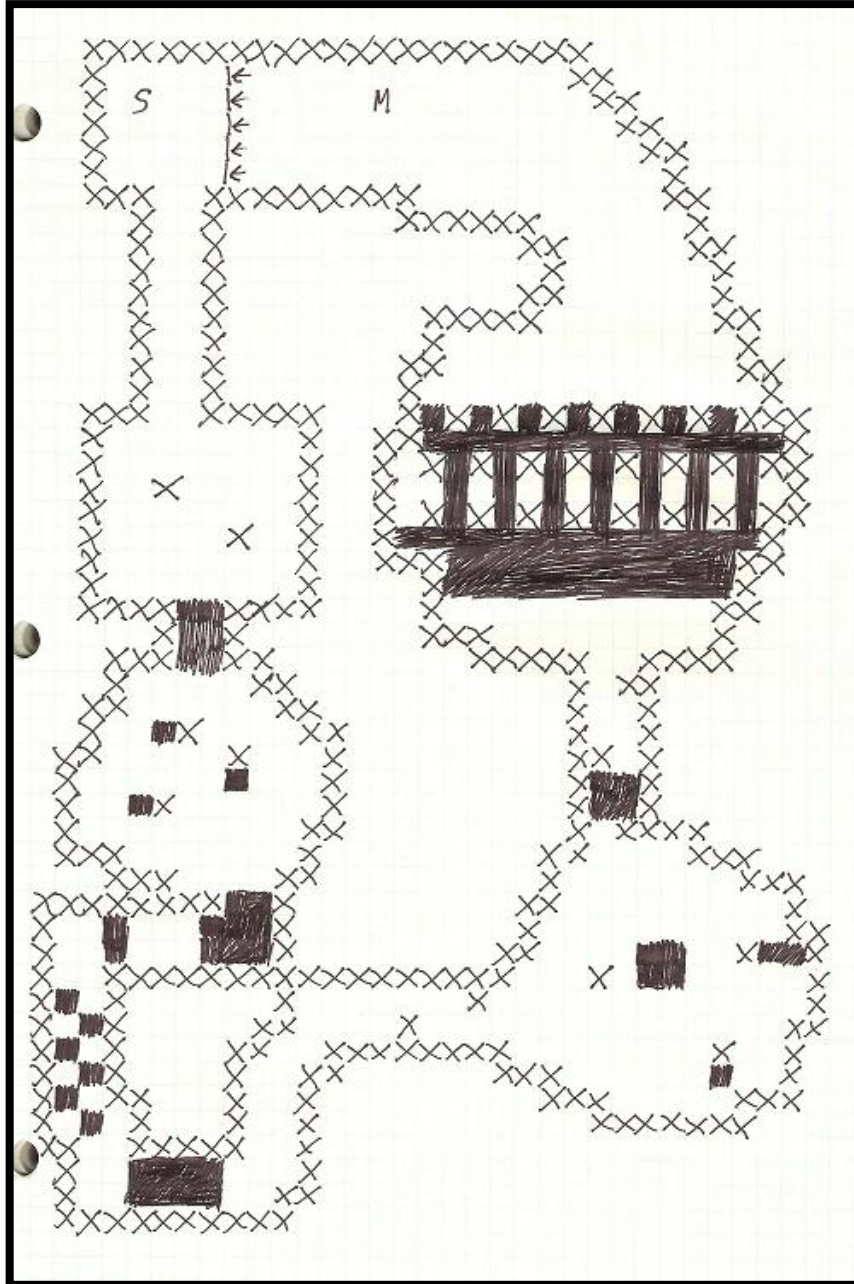
Enemy 2

This character is a second enemy that inhabits Planet X. This character carries a weapon that shoots balls of fiery gas. The inspiration for shots of fiery balls of gas instead of standard bullets for this enemy's weapon was so that we could remain consistent with the futuristic, outer-space theme. The main player can withstand five of these shots before he dies, and the game is over.



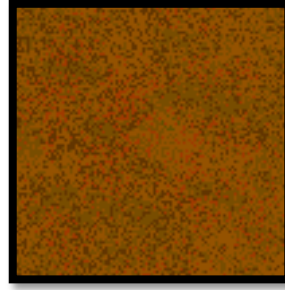
Sample Level Design

Following is a sample level. An “S” Signifies the player’s starting and ending point, an “M” is the mineral that must be acquired to complete the level, an “X” is an impassible tile, a blacked out tile is a pit which player must use the jet pack to fly over, and an arrow with a line signifies that the player may cross this tile only when traveling in the direction of the arrow.

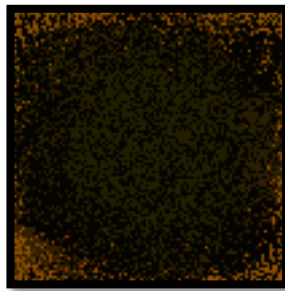




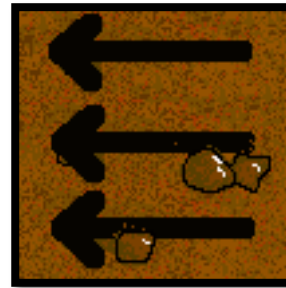
Impassible tile



Passible tile



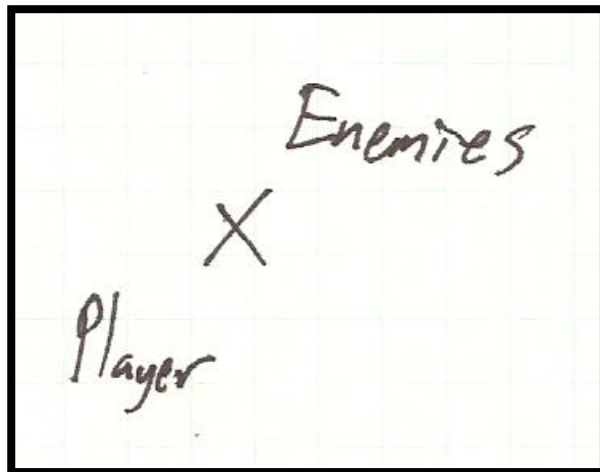
Pit



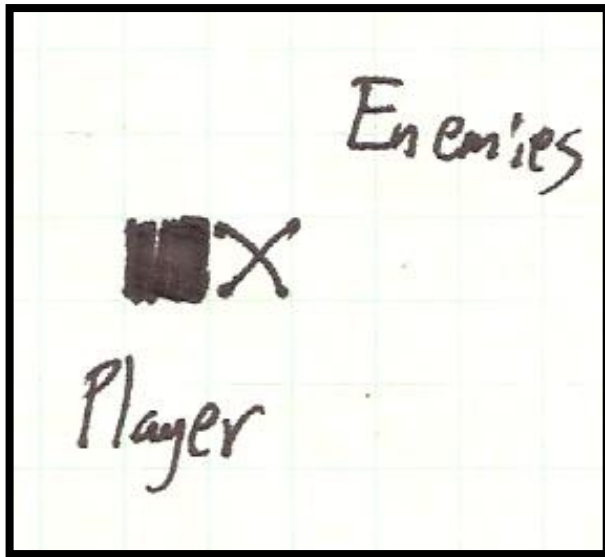
Directional tile

Rationale for certain features

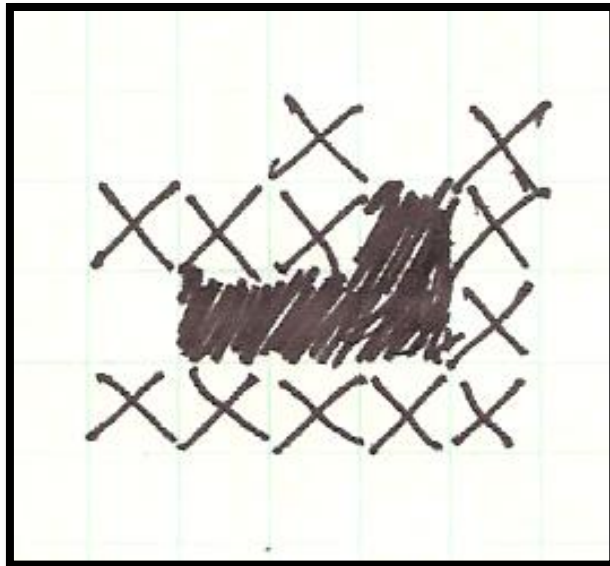
This sort of feature (a lone impassible tile) can be used by the player for cover from enemies.



This feature below (a lone impassible tile adjacent to a pit tile) is similar to the above but only allows partial cover, as the player who ducks behind it too quickly may fall into the pit.



This feature below requires the player to fly around a corner, grappling with the forces of acceleration in the air as he/she does so.



XML File Format for Level Storage

Level files will be generated using a separate level editor utility. This utility (designed expressly for this game) will allow a designer to easily map out a level using GUI tools and then will convert the user input to an XML file which will then be used by the main game to recreate the designer's work.

Levels will be stored in the following XML Format:

```
<level>(the opening tag for a level, there will only be one per file)
  <size>(tag indicating that this section of the document contains the number of
rows and columns of level tiles, there will only be one per file)
    <rownumber></rownumber>(tag containing the number of rows in the
grid; one per file)
    <columnnumber></columnnumber>(tag containing the number of
columns in the grid; one per file)
  </size>
  <rows>(tag containing a number of <row> tags equivalent to that contained in
<rownumber>; one per file)
    <row>(tag representing a row of tiles (as many tiles as the number in
<columnnumber>); many per file)
      <tile>(tag representing a tile; many per file)
        <type></type>(tag containing the tile's type; one per
<tile>)
        <unit></unit>(tag indicating what sort of unit is on the tile,
if at all; one per <tile>)
      </tile>
    </row>
  </rows>
</level>
```

Movement and Control

As the game is to be a 2D top down shooter, it is important that the movement and controls of the game give the player the ability to quickly and accurately react to events in the game world. Since the game is to feature a full range of movement (meaning the player will be able to move at any angle they choose) it is critical that this mechanic be easy enough for the player to use without it negatively affecting their game experience. Tentatively it is proposed that the game will use a compound movement controls system where both mouse and keyboard will be used. The mouse or more specifically the mouse cursor will control the player's direction while the keyboard will control movement and other game actions.

More specifically a player's current direction will be derived from getting a vector from where the player's mouse cursor is in the game world to where the player is in the game world. This vector will only be the direction that the player faces in game. This vector will be used to determine how the player's sprite should be rotated and in which direction his projectiles will be fired. Mouse movement will only change this direction as it will alter the vector and it will not make the player move in that direction. Only the movement buttons forward, back, left and right will effect the player's motion. All these motions will be done relative to this direction vector. Mouse buttons will be assigned to control critical player game actions, such as firing his weapon with left click and activating his jet pack with right click. The player's mouse cursor will be replaced with a cross hair while inside the game window to help make this mechanic more obvious and intuitive to the player.

It is hoped that this relatively simple mouse and keyboard movement control scheme will be familiar to most PC shooter game players. Alternatively, should this scheme prove to be difficult to use, the game might be controlled with a more traditional game controller, such as an XBOX controller, with an analog stick controlling movements such as forward and back and joystick controlling the player's direction. In this case, instead of getting a vector from the cursor position in the game world, the player direction will just come from the orientation on the joystick.

Non movement player controls such as activating the jet-pack and firing weapons will be control separately in the form of either mouse button presses, keyboard key presses, or buttons on the game control. These are likely to be hard coded in prototype builds but be customizable in the final product.

Technical Specification

Programming Language: C#

Hardware: PC

Operating Systems Supported: Windows

Framework: XNA Game Studio 4.0

Class Design

Note: Variables and private functions will not appear in the design document.

1. Entity:

Entity class is an abstract class. Any other entity-like class must derive from this abstract class. This class encapsulates essential attributes and properties of entities in game.

XNA Framework used:

Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Graphics;

abstract class Entity

```
{
    //Constructor 1: base constructor
    /**parameter list**
    //1. Texture2D – sprite or sprite sheet of image being drawn
    //2. Vector2 – where to draw this entity
    //3. Point – size of individual frames in the sprite sheet
    //4. int – collision offset value, used for checking collision
    //5. Point – index of current frame in the sprite sheet
    //6. Point – number of columns and rows in the sprite sheet
    //7. Vector2 – speed at which entity moves in both one to another directions
    //8. int – number of milliseconds to wait between frame changes
    //9. string – for collision sound effect
    //10. int – amount of life
    /*******
    public Entity(Texture2D, Vector2, Point, int, Point, Point, Vector2, int, string,
    int) {}

    //Constructor 2
    //This constructor does not take the value of millisecond per frame (8), and will
    just
    //use the default value
    public Entity(Texture2D, Vector2, Point, int, Point, Point, Vector2, string, int) :
this(...) {}

    //Constructor 3
    //This constructor is as same as constructor 2 but take scale ratio (float) as an
    extra //parameter in the end
    public Entity(Texture2D, Vector2, Point, int, Point, Point, Vector2, string, int,
float) : this(...) {}

    //Abstract definition of getting direction property
    public abstract Vector2 direction { get; }
}
```

```

//Getting current position of the entity
public Vector2 GetPosition { get {} }

//Getting current amount of life
public int GetLifeAmount { get; protected set; }

//Getting sound information
public string collisionSoundName { get; private set; }

//Getting collision rectangle based on position, frame size, scale, offset, etc
public Rectangle collisionRectangle { get {} }

//Virtual definition: update the logic of this entity such as movement, input,
collision //detection, etc.
public virtual void Update(GameTime, Rectangle) {}

//Virtual definition of drawing this entity based on current variable values
public virtual void Draw(GameTime, SpriteBatch) {}

//Checking if entity is out of game window
public bool IsOutOfWindow(Rectangle) {}

//Modifying size of entity
public void ModifyScale(float) {}

//Resetting size of entity
public void ResetScale() {}

//Modifying speed of entity
public void ModifySpeed(float) {}

//Resetting speed of entity
public void ResetSpeed() {}

//Damage taken
public void TakeDamage(int) {}

//Removing this entity
public void KillThisEntity() {}
}

```

2. ControlledEntity:

ControlledEntity derives from Entity class. It creates entities that players can control. In game, Mr. Astronaut is classified here.

XNA Framework used:

Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Graphics;
Microsoft.Xna.Framework.Input;

class ControlledEntity : Entity

```
{  
    //Constructor 1  
    //This constructor is based on Entity constructor 2  
    public ControlledEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2) :  
    base(...) {}  
  
    //Constructor 2  
    //This constructor is based on Entity constructor 1  
    public ControlledEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,  
    int) :  
    base(...) {};  
  
    //Getting direction of entity based on input and speed  
    public override Vector2 direction { get } }  
  
    //Overriding Update method of base class  
    public override void Update(GameTime, Rectangle) {}  
  
    //Overriding Draw method of base class  
    public override void Draw(GameTime) {}  
  
    //Checking life of this entity  
    public boolean IsAlive() {}  
  
    //Shooting ammo (laser beam)  
    public void FireShots() {}  
  
    //Checking if jetpack is used  
    public boolean IsFloating() {}  
  
    //Using jetpack  
    public void UsingJetpack() {}  
}
```

3. AutomatedEntity:

AutomatedEntity is designed for creating entities that doesn't change direction and speed. Ammos, laser beams, and other projectiles are classified. The class also derives from Entity class.

XNA Framework used:

```
Microsoft.Xna.Framework.Graphics;
```

class AutomatedEntity : Entity

```
{  
    //Constructor 1  
    //based on base constructor 2  
    public AutomatedEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,  
string, int) : base(...) {}  
  
    //Constructor 2  
    //based on base constructor 1  
    public AutomatedEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,  
int, string, int) : base(...) {}  
  
    //Constructor 3  
    //based on base constructor 3  
    public AutomatedEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,  
string, int, float) : base(...) {}  
  
    //Direction is same as speed since entity is automated  
    public override Vector2 direction { get {} }  
  
    //Overriding Update method of base class  
    public override void Update(GameTime, Rectangle) {}  
  
    //Checking friend or foe to make hit  
    public boolean IsFoe(Entity) {}  
}
```

4. EntityManager:

EntityManager class is going to be used for integrating different logical pieces of code into the game. This class also derives from Microsoft.Xna.Framework.DrawableGameComponent, so it can invoke Draw methods of all the entities that it manages by being wired up to the game's Draw method.

XNA Framework used:

```
Microsoft.Xna.Framework;  
Microsoft.Xna.Framework.Audio;  
Microsoft.Xna.Framework.Content;  
Microsoft.Xna.Framework.GamerServices;  
Microsoft.Xna.Framework.Graphics;  
Microsoft.Xna.Framework.Input;
```

Microsoft.Xna.Framework.Media;

public class EntityManager : Microsoft.Xna.Framework.DrawableGameComponent

```
{
    //Constructor. All child components will be constructed here
    public EntityManager(Game) : base(...) {}

    //Overriding
    public override void Initialize() {}

    //Overriding
    public override void Update(GameTime) {}

    //Overriding
    public override void Draw(GameTime) {}

    //Updating and Managing all entities in the manager
    protected void UpdateEntities() {}

    //Adjusting enemies' spawn times based number of enemies on screen
    protected void ModifySpawnTimes(GameTime) {}

    //Checking expiration of any power-up abilities
    protected void CheckPowerUpExpiration(Game) {}

    //Updating positions of ammos appearing on the screen
    protected void UpdateShots() {}

    //Updating necessary elements when any enemy is eliminated
    protected void UpdateDeath () {}

    //adding shots in the screen
    //parameters are position, direction of ammo, and type of entity
    public void AddShots(Vector2, Vector2, AutomatedEntity) {}
}
```

5. RangeAIEntity:

RangeAIEntity, deriving from Entity class, is designed for creating enemies who can do range attack. The AI makes this entity keep itself away from the player in safe distance when the player approaches. Moreover, this entity is allowed to fire ammos while moving. The class derives from Entity class.

XNA Framework used:

Microsoft.Xna.Framework;

Microsoft.Xna.Framework.Graphics;

```

public class RangeAIEntity : Entity
{
    //Constructor 1
    //based on the base constructor 2, parameters are the same except the last three
    ones
    //The last three parameters are following:
    //EntityManager – need to update EntityManager once AI’s decision is made, as
    well as
    //          get player’s current position in game for decision making
    //float – speed modifier
    //int – setting the minimum distance to avoid the player

    public MeleeAIEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,
string, int, EntityManager, float, int) : base(...) {}

    //Constructor2
    public MeleeAIEntity(Texture2D, Vector2, Point, Point, int, Point, Point,
Vector2, int, string, int, EntityManager, float, int) : base(...) {}

    //Getting entity’s speed
    public override Vector2 direction { get {} }

    //Overriding Update method
    //The algorithm of AI for melee enemy will be used and (or) coded here, such as
    //moving, shooting, hiding, etc
    public override void Update(GameTime, Rectangle) {}

    //checking if entity needs removing
    public boolean IsAlive() {}
}

```

6. MeleeAIEntity:

The function of MeleeAIEntity, deriving from Entity class, is the same with RangeAIEntity. Instead, MeleeAIEntity takes use the last parameter to make valid melee attack possible. Also, the AI makes this entity more aggressive chasing player. The class derives from Entity class.

XNA Framework used:

```

Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Graphics;

```

```

public class MeleeAIEntity : Entity
{

```

```

    //Constructor 1

```

```

//based on the base constructor 2, parameters are the same except the last three
ones
//The last three parameters are following:
//EntityManager – need to update EntityManager once AI’s decision is made, as
well as
//                get player’s current position in game for decision making
//float – speed modifier
//int – setting valid melee range to attack the player
public MeleeAIEntity(Texture2D, Vector2, Point, int, Point, Point, Vector2,
string, int, EntityManager, float, int) : base(...) {}

//Constructor2
//based on the base constructor 1.
//same with Melee AI Entity constructor 1 but milliseconds per frame is specified
public MeleeAIEntity(Texture2D, Vector2, Point, int, Point, Point, Point,
Vector2, int, string, int, EntityManager, float, int) : base(...) {}

/Getting entity’s speed
public override Vector2 direction { get {} }

//Overriding Update method
//The algorithm of AI for melee enemy will be used and(or) coded here such as
//chasing, avoiding traps, melee attacking, etc
public override void Update(GameTime, Rectangle) {}

//checking if entity needs removing
public boolean IsAlive() {}
}

```

7. CamView:

CamView allows to control the movement of the Camera. Since the game is a top-down 2D game, the camera is always on the top of player’s character’s head. Moreover, the game intends to keep player’s character in the middle of screen, so the camera will automatically move to the position where the character is. The camera has a fixed direction and lookup. The position X,Y of camera are always the same with character’s position X, Y, and the position Z of the camera is set as constant (character’s position Z is always 0 because the game is based on 2D platform). CamView class derives from Microsoft.Xna.Framework.GameComponent.

XNA Framework used;

```

Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Audio;
Microsoft.Xna.Framework.Content;
Microsoft.Xna.Framework.GamerServices;
Microsoft.Xna.Framework.Graphics;

```

```
Microsoft.Xna.Framework.Input;
Microsoft.Xna.Framework.Media;
Microsoft.Xna.Framework.Net;
Microsoft.Xna.Framework.Storage;
```

```
public class CamView : Microsoft.Xna.Framework.GameComponent
{
    //Constructor
    //Passing EntityManager to get player's current position
    //The last parameters in order are cam's direction and lookup
    public CamView(Game, EntityManager, Vector3, Vector3) : base(...) {}

    //Returning camera position. X and y-values are as same as player current position
    x,y, //and z-value is constant
    public Vector3 GetCameraPosition { get } {}

    //Overriding methods following:
    public override void Initialize() {}

    public override void Update(GameTime) {}
}
```

8. HUD:

HUD (head-up display) class is created for displaying player hp bar and fuel bar in game process, and is responsible for updating values when certain conditions are met. Also, other components, such as pictures or avatars, can be added by using this class. Because this class draws, so it derives from Microsoft.Xna.Framework.DrawableGameComponent.

XNA Framework used:

```
Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Graphics;
```

```
public class HUD : Microsoft.Xna.Framework.DrawableGameComponent
{
    //Constructor
    public HUD(Game) {}

    //Display player life
    public int PlayerLife { set; }

    //Display fuel
    public int Fuel { set; }

    //Display "frost" mode
```

```

public boolean FrostMode { set; }

//Display using jetpack
public boolean UsingJetpack { set; }

//Overriding, drawing UI
public override void Draw(SpriteBatch) {}
}

```

9. PromptScreen:

PromptScreen class is designed for displaying menu screen, cut scenes, game-over screen, etc in certain game states. The class derives from Microsoft.Xna.Framework.DrawableGameComponent.

XNA Framework used:

```

Microsoft.Xna.Framework;
Microsoft.Xna.Framework.Audio;
Microsoft.Xna.Framework.Content;
Microsoft.Xna.Framework.GamerServices;
Microsoft.Xna.Framework.Graphics;
Microsoft.Xna.Framework.Input;
Microsoft.Xna.Framework.Media;

```

```

public class PromptScreen : Microsoft.Xna.Framework.DrawableGameComponent
{
    //Constructor
public PromptScreen(Game) {}

    //draw texts corresponding to game states
public void SetTexts(string, OPX.GameState) {}

    //draw BGs corresponding to game states
public void SetBG(string, OPX.GameState) {}

    //Overriding methods following
public override void Initialize() {}

protected override void LoadContent() {}

public override void Update(GameTime) {}

public override void Draw(GameTime) {}
}

```

10. LevelData:

This class is designed for XML parsing. Based on the design document, all level designs are stored in XML files; then the data in XML format is converted to appropriate data types for C#. In the other words, the purpose of this class only does XML parsing – retrieving raw data from XML file, creates a package (collection), and then send the package to C#. Therefore, this class contains many private functions, which are not shown in the paper,” for XML parsing, and is indirectly relevant to the game engine.

XNA Framework used: None

```
public class LevelData  
{  
    //Constructor  
    //parameter takes a string as a name of XML file  
    public LevelData(string) {}  
  
    //add all contents from XML into collection that C# uses  
    public Collection add() {}  
}
```

11. OPX:

OPX stands for “Operation Planet X”. This class is just as same as “Game1”, which Microsoft Visual Studio C++ creates by default, when a new project is created. The slight difference is that “Game1” is renamed as “OPX”, and the game related components are added into this class.

XNA Framework used:

```
Microsoft.Xna.Framework;  
Microsoft.Xna.Framework.Audio;  
Microsoft.Xna.Framework.Content;  
Microsoft.Xna.Framework.GamerServices;  
Microsoft.Xna.Framework.Graphics;  
Microsoft.Xna.Framework.Input;  
Microsoft.Xna.Framework.Media;  
Microsoft.Xna.Framework.Net;  
Microsoft.Xna.Framework.Storage;
```

```
public class OPX : Microsoft.Xna.Framework.Game  
{  
    //Setting state machine for the game  
    //START – menu before game starts  
    //PLAY – gameplay in process  
    //NEXTLEVEL – loading next level inform  
    //END – credit list and game over  
    public enum GameState { START, PLAY, NEXTLEVEL, END };
```

```

//constructor
public OPX() {}

//default method
protected override void Initialize() {}

//default method
protected override void LoadContent() {}

//default method
protected override void UnloadContent() {}

//default method
protected override void Update(GameTime) {}

//default method
protected override void Draw(GameTime) {}

//Playing cue
public void PlayCue(string) {}
}

```

Project Timeline

Week 1-2

Discuss member weaknesses and strengths
 Brainstorm game ideas
 Submit Statement of Team Composition
 Develop game theme, story, and general game specifications

Week 2-3

Pick framework and programming language
 Become familiar with framework and language
 Sketch potential characters
 Develop tile ideas

Week 3-4

Complete sprites of main player and enemy 1
 Sketch four tiles: passable, impassable, pit, arrows

Week 5-6

Complete four tile sketches
Complete tile and character interaction illustrations
Establish version control

Week 7

Complete four tiles: 60x60 or 100x100 mp
Sketches of main menu including Frost mode option
Build prototype for study and research
Begin collision detection coding, level coding
Begin implementation of character control and enemy path finding

Week 8

Implement bullet coding(player and enemy), test with collision
Implement jet pack control feature, test with tiles
Attempt particle jet pack emission coding
Implement tentacles attack, test with player
Implement enemy spawning system
Add background music to game
Add sound-effects to specific features(jet pack, bullets,..)
Playable mini version of game ready

Week 9

Replace all sprite-place holders with final versions of sprites
Test implemented features and collisions within the world
Assign static enemy locations within the levels
Debugging

Week 10

Testing
More debugging, if necessary
Finalize game

Week 11

Present game

Project Tasks

First Half of Quarter

- Nivardo**
- Character-character interaction
 - Bullet types and controls
- Stephen**
- Level design
 - Develop tile design and player interaction with tiles
- Robert**
- Methods to control player
 - Determine implementation of player control
- William**
- Rough class design
 - Collision detection implementation research
- Abi**
- Artwork:
- Main player sprite
 - Enemy 1 sprite
 - Enemy 2 sprite

Second Half of Quarter

- Nivardo**
- Enemy path finding implementation
 - Bullet system implementation for enemy and player
 - Enemy spawning system implementation
 - Background music and sound-effects
- Stephen**
- Level design & tile placement implementation
 - Fuel health bar concept implementation
 - Tile design and interaction
- Robert**
- Character movement and control implementation
 - Jet pack control feature implementation
 - Attempt jet pack particle emission system implementation
- William**
- Collision detection implementation
 - To adjust and modify the class design based on implementation
- Abi**
- Artwork:
- Main menu that includes Frost mode option
 - Cut scene between start and level 1
 - Cut scene between level 1 and 2
 - Final credit list
 - Fire sprite (under jet pack)
 - 4 Tiles

