Progetto di Laboratorio Sistemi & Reti

SPEARPOINT GALAXY

Paolo Imbriani 4°F

A.S 2021 / 2022

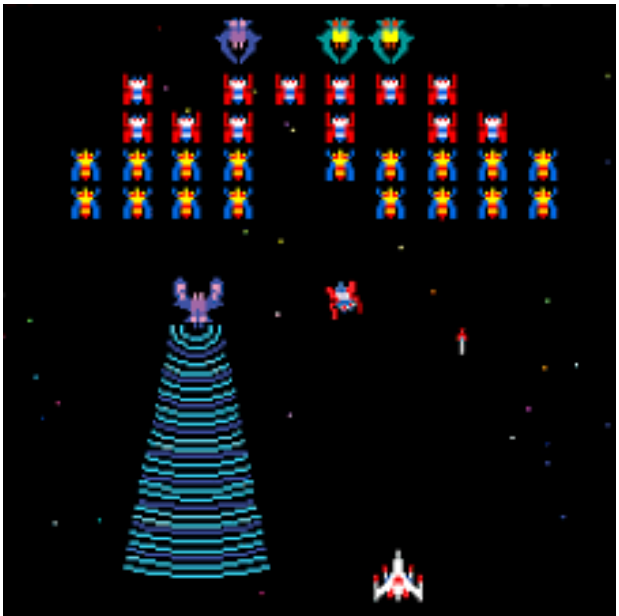
INDICE

Introduzione

Quello che è andrete a leggere è una relazione creata per il progetto di linguaggio C# da creare con XNA o con Unity. Tutto il gioco è composto da circa **25 Classi**. Ovviamente la descrizione e lo scandire di tutte le informazioni sulla creazione del videogioco sarà piuttosto prolissa, quindi spero di non essere noioso nella mia esposizione. Questo documento è diviso in *Descrizione del gioco*, *Comandi di gioco*, *Scene di gioco* e *il codice di programmazione* che sono tutte e quattro abbastanza esplicative.

I. Descrizione del gioco

SpearPoint Galaxy è un videogioco originale che vuole dare omaggio ai grandi giochi retrò nati negli 70/80, come "*Galaga*" (uscito nell'81) o il classico *"Space Invaders"*, tutti giochi che venivano giocati nei classici *Arcade (*per poi essere presenti in quasi tutte le console esistenti). *Galaga* per esempio è stato per *Namco (*la casa videoludica che ha creato Galaga e molti altri giochi) la prima hit personale nel mondo dei videoludico. Nonostante I vari spunti presi da questi giochi, *SpearPoint Galaxy* si vuole differenziare per la sua natura più da *Survival Arcade*.



Schermata di gioco di "Galaga"

Il gioco consiste nel buttarsi nello spazio più tetro e cercare *di resistere il più possibile* in modo da formare un nuovo e sempre più grande ***Record Imbattibile***. Il tempo in questo caso, non solo scandisce per quanto tempo si è sopravvissuti in quel momento, ma serve anche per creare un punteggio che si basa proprio sul *tempo di sopravvivenza.* Asteroidi e persino Ufo, arriveranno in contro all'astronave e l'unico modo per avere successo e schivare quello che ci si trova davanti e provare a sparare agli oggetti imminenti. Più si procederà nel gioco, più sopravvivere diventerà ostico, e sarà sempre più difficile battere il record più alto. Una scia di asteroidi e ufo frenetica che non permetteranno all'astronave di avere vita lunga.

Se immaginiamo un ipotetica trama per questo videogioco questa sarebbe più o meno così:

Anno 2117. Gli **SpearPointers** sono degli assi dell'aeronautica spaziale. A loro è stata affidata una missione di assoluta importanza. Una missione che comporterà la salvezza della terra: sterminare gli ufo e asteroidi che si stanno per approcciare sulla terra: lo **SpearPointer77** (uno dei più abili SpearPointer della terra) dopo una lunga battaglia, si trova da solo, dopo che tutti I compagni sono stati devastati dagli asteroidi o polverizzati dagli ufo. Alcuni resti lasciati dai compagni potrebbero tornare utili, sfruttali a tuo vantaggio! Ogni speranza per il pianeta sembra perduta: quanto a lungo riuscirà a resistere lo SpearPoint77, *in modo tale da riuscire a guadagnare tempo per la terra che non sembra avere via di scampo?*

Schermata di gioco di *"Spearpoint Galaxy"*

2. COMANDI DEL GIOCO

I comandi di gioco sono I seguenti:

I tasti **UP, DOWN, RIGHT e LEFT** servono per spostare l'astronave in verticale o in orizzontale senza nessun limite nella mappa.

Il tasto **E** per sparare (il proiettile apparirà sulla punta dell'astronave) tuttavia ci sarà un delay che limiterà lo sparare dell'astronave.

3. Scene di gioco

Il videogioco contiene cinque scene principali che adesso andremo a spiegare in dettaglio.

3.1 Scena di start

La scena di start è composta da un background, un logo e un menu interagibile grazie alla classe GameMenu che verrà spiegata in seguito. Purtroppo avrei voluto inserire **le song** ma non sono riuscito ad inserirle a causa di un errore in fase di lancio del programma. In cima al menu, si trova il logo del gioco, totalmente creato da me, che mostra proprio lo Spearpointer. Il menu mostra le seguenti opzioni:

- Nuova Partita: Crea una nuova scena di gioco e permette di riniziare la partita.
- Comandi: Mostra la scena di aiuto per i comandi.
- Classifica: Mostra la classifica
- Esci: Esce dal gioco.

3.2 Scena di gioco

La scena di gioco è sicuramente la scena più piena di contenuti del videogioco, essendo che è dove si concentra tutto: sono presenti l'astronave, l'hud (che mostra il tempo e il punteggio del gioco), I proiettili che vengono sparati dall'astronave, Il "Bolt-PowerUp", gli asteroidi e le collisioni. Aggiungo anche un effetto sci-fi per quando si spara.

Nella scena di gioco ho dovuto inserire una proprietà dell'astronave in modo tale da poter usare le variabili dell'astronave anche nelle altre classi.

```
public Spaceship Spaceship
{
    get { return spaceship;}
}
```

Ovviamente ho dovuto inserire un "if" aggiuntivo nell'ai metodi Update e Draw dove controllo se la scena di play è attiva al momento: se la scena di gioco non è attiva, non ha senso aggiornare o disegnare gli elementi nella scena di gioco.

3.3 Scena di Help

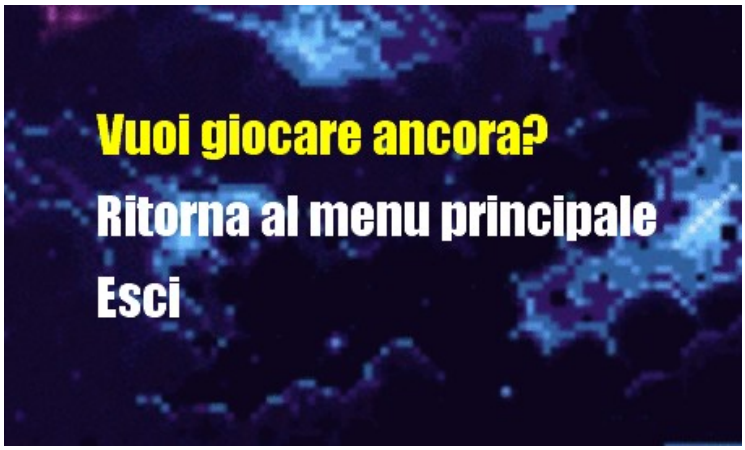
Una scena semplicissima dove sono mostrati i semplici comandi di gioco.

3.4 Scena di Classifica

Grazie alla classe HighScores che andremo a spiegare nella parte di codice, la schermata di classifica consiste nel mostrare I tre punteggi più alti mai registrati nel gioco. Si parte dal primo che è più alto e così via. Inizialmente avevo utilizzato una soluzione **col Common**, ma alla fine sono riuscito ad utilizzare i file, riprendendo dal progetto scorso sempre fatto con C#.

3.5 Scena di GameOver

Una semplice scena che permette di ritentare il gioco una volta morti: si può creare una nuova partita oppure ritornare al menu principale per controllare se si ha fatto un nuovo record. Oppure direttamente uscire.



4. Codice di programmazione

Il codice del videogioco ha avuto uno sviluppo abbastanza contorto. L'idea del videogioco era chiara fin da subito, ma col passare del tempo, sono andato a modificare alcune delle idea base da dove avevo preso fondamento. Per esempio, la "alfa" del mio videogioco, era soltanto una navicella che si muoveva nello spazio e che era in grado di sparare questi proiettili. Subito dopo mi venne l'idea di creare un gioco stile *survival Arcade* con delle ondate che mano a mano diventava sempre più difficile. Anche i solo powerup, che nell'idea principale non erano presenti, sono stati inseriti molto dopo.
Per la creazione del videogioco sono state implementate le seguenti classi:

- **Background**: gestisce lo sfondo.
- **BackgroundAnimated**: gestisce lo sfondo animato.
- **Bullet:** gestisce il singolo proiettile
- **Bullets:** gestisce i proiettili

- **Collision**: gestisce le collisioni.
- **Common:** contiene variabili e oggetti globali di utilizzo generale.
- **Enemies:** gestisce i nemici.
- **Enemy:** gestisce il nemico singolo.
- **Game1:** gestisce la classe principale
- **GameMenu:** gestisce I menu del gioco.
- **HighScores:** gestisce la classifica da file
- **HUD:** gestisce la HUD
- **KeyboardInput:** gestisce I comandi da tastiera.
- **Logo:** gestisce il logo.
- **PowerUp**: gestisce il powerup singolo.
- **PowerUps:** gestisce I powerup
- **Spaceship:** gestisce l'astronave
- **Ufo:** gestice l'ufo singolo
- **Ufos:** gestisce gli ufo

Ma bando alle ciance e iniziamo!

4.1 Background

La classe **Background** gestisce lo sfondo. L'unico attributo memorizzato nella classe è la texture di sfondo.

```
private Texture2D texture;
```

Il costruttore inizializza il precedente attributo, impostando la texture.

```
        public Background(string textureName)
        {
            texture = Common.Content.Load<Texture2D>(textureName);
        }
```

Il metodo Draw disegna lo sfondo.

```
public void Draw()
        {
            Common.SpriteBatch.Draw(texture, Vector2.Zero, Color.White);
        }
```

4.2 BackgroundAnimated

La classe BackgroundAnimated gestisce lo sfondo animato. Non avendo sprite decenti per utilizzare un animazione del gioco ho deciso di creare una classe che almeno animasse lo sfondo di inizio. Gli attributi che ho utilizzato sono:

```
private Texture2D texture;
private double elapsed = 0;
```

Il costruttore è un costruttore vuoto poiché tutta l'animazione verrà fatta nell'**Update()**. Il problema di questa animazione, è che non ho trovato altri modi per sviluppare questa animazione, non essendo un vero e proprio asset di frame. Ho splittato una GIF in frames, e ho ottenuto questi frame! Quindi ho optato per questa opzione pur essendo parecchio lunga.

```
public void Update()
        {
            elapsed += Common.GameTime.ElapsedGameTime.TotalMilliseconds;
            if(elapsed > 0)
            {
                texture = Common.Content.Load<Texture2D>("frame1");
            }
            if(elapsed > 0 && elapsed < 100)
            {
                texture = Common.Content.Load<Texture2D>("frame2");
            }
            if(elapsed > 100 && elapsed < 200)
            {
                texture = Common.Content.Load<Texture2D>("frame3");
            }

            […]

            if (elapsed > 3600 && elapsed < 3700)
            {
                texture = Common.Content.Load<Texture2D>("frame36");
            }
            if(elapsed > 3700)
            {
                elapsed = 0;
            }
      }
```

Ovviamente nel **Draw** si disegna tutto come è stato per il Background normale.

4.3 BULLET

La classe **Bullet** gestisce il proiettile singolo. Gli attributi memorizzati nella classe sono la texture, la posizione, la velocità e una flag per controllare se il proiettile è vivo o morto.

```
        private Texture2D texture;
        private Vector2 position;
        private int speed;
        private bool alive;
```

Il **costruttore** inizializza gli attributi e dobbiamo passargli come parametri la texture e la posizione.

```
 public Bullet(string textureName, Vector2 position)
        {
            texture = Common.Content.Load<Texture2D>(textureName);
            speed = 20;
            this.position = position;
            alive = true;
        }
```

Per ogni oggetto nella PlayScene() ho voluto che ogni classe avesse le proprietà e quindi un **get e un set** per ogni attributo. Un esempio:

```
public Texture2D Texture
        {
            set { texture = value; }
            get { return texture; }
        }
```

In modo tale da impostare la texture in ogni punto del codice.

Essendo che tornerà utile nelle collisioni, ho deciso di dargli una hitbox grazie alla classe **Rectangle**.

```
public Rectangle Hitbox
        {
            get
            {
                return new Rectangle((int)position.X, (int)position.Y, texture.Width,
texture.Height);
            }
        }
```

Nella funzione **Update()** invece, non facciamo altro che farlo mandare dal basso verso l'alto.

```
public void Update()
        {
            position.Y -= speed;
        }
```

4.4 BULLETS

La classe **Bullets** serve per gestire la lista di proiettili. Nella classe possiamo trovare i seguenti attributi.

```
        private List<Bullet> projectiles;
        private Spaceship spaceship;
        Sound effect;
```

Nel costruttore invece gli passiamo come parametro l'astronave e l'effetto che farà quando il proiettile verrà sparato. Ovviamente creiamo anche la lista di proiettili.

```
public Bullets(Spaceship spaceship, Sound effect)
        {
            this.spaceship = spaceship;
            projectiles = new List<Bullet>();
            this.effect = effect;
        }
```

Ovviamente aggiungiamo anche una funzione di aggiunta, che quindi aggiunge **i Bullet** nella lista dei **projectiles** (la lista di proiettili) passandogli come parametro un oggetto **Bullet**.

```
 public void Add(Bullet b)
        {
            projectiles.Add(b);
        }
```

Nell'**Update** invece facciamo un veloce controllo per vedere se l'astronave sta attacando. Se l'astronave è in fase d'attacco, allora spara il proiettile e riproduce il suono. E ovviamente aggiorniamo il proiettile per ogni proiettile che sta nella lista.

```
public void Update()
        {
            if(spaceship.Is_attacking)
            {
               effect.Play();
               projectiles.Add(new Bullet("laserr", new Vector2(spaceship.Position.X +
spaceship.HitBox.Width/2 - 5, spaceship.Position.Y)));
            }

            foreach(Bullet b in projectiles)
            {
                b.Update();
            }
        }
```

Nel **Draw** invece andremo a disegnare il proiettile solo è soltanto se è vivo nella lista.

```
public void Draw()
        {
            foreach(Bullet b in projectiles)
            {
                if(b.Alive)
                {
                    b.Draw();
                }

            }
        }
```

4.5 COLLISION

La classe **Collision** gestisce tutte le collisioni nel gioco. Intanto vengono creati i seguenti attributi che servono per gestire le collisioni di tutti gli oggetti nella PlayScene.

```
        private Spaceship spaceship;
        private Enemies enemies;
        private Bullets bullets;
        private Ufos ufos;
        private PowerUps powerups;
        Sound Bolt;
```

Il costruttore invece cerca di copiare ogni cosa passata come parametro agli attributi della classe in modo da poterli gestire. Creo anche il suono per il PowerUp.

```
public Collision(Spaceship spaceship, Enemies enemies, Bullets bullets, Ufos ufos, PowerUps
powerups)
        {
            this.spaceship = spaceship;
            this.enemies = enemies;
            this.bullets = bullets;
            this.ufos = ufos;
            this.powerups = powerups;
            Bolt = new Sound("PowerUp", SoundType.Effect);
        }
```

Le collisioni non essendo visibili non avranno un metodo **Draw** bensì solo un metodo **Update** dove tutte le collisioni del caso verranno gestite. Una volta che un qualcosa viene colpito verrano flaggate tutte le variabili booleane dell'oggetto colpito: per i powerup invece, da un boost che permette all'astronave di sparare più velocemente. Quando un asteroide viene colpito dal proiettile, ho settato che una volta colpito, non solo non sarà visibile, ma scomparirà anche dalla schermata, essendo che pur non essendo visibile, il proiettile si sarebbe comunque fermato su di esso.

```
 public void Update()
        {

            foreach(Ufo u in ufos.Aliens)
            {
                if(u.isVisible)
                {
                    if(spaceship.HitBox.Intersects(u.Hitbox))
                    {
                        spaceship.Is_alive = false;
                    }
                }

                foreach (Bullet b in bullets.Projectiles)
                {
                    if (b.Alive)
                    {
                        if (b.Hitbox.Intersects(u.Hitbox))
                        {
                            u.isVisible = false;
                            b.Alive = false;
                        }
                    }
                }


            }

            foreach (Enemy e in enemies.Nemici)
            {

                if (e.isVisible)
```

```
            {
                if (spaceship.HitBox.Intersects(e.Hitbox))
                {
                    spaceship.Is_alive = false;
                }
            }

            foreach (Bullet b in bullets.Projectiles)
            {
                if (b.Alive)
                {
                    if (b.Hitbox.Intersects(e.Hitbox))
                    {
                        e.position = new Vector2(Common.GameWidth + e.texture.Width, 0);
                        e.isVisible = false;
                        b.Alive = false;
                    }
                }
            }
        }

        foreach(PowerUp p in powerups.Potenziamenti)
        {
            if(spaceship.HitBox.Intersects(p.Hitbox))
            {
                p.isVisible = false;
                spaceship.Powerup_activated = true;
                spaceship.PowerupCooldown = 4500;
                Bolt.Play();
            }
        }
    }
```

4.6 COMMON

La classe **Common** contiene variabili e oggetti globali di utilizzo generale. Gli attributi memorizzati nella classe sono la larghezza e l'altezza della finestra, il titolo del videogioco, l'oggetto SpriteBatch per il disegno delle texture, l'oggetto Content per la gestione dei contenuti multimediali, l'oggetto gametime per la gestione del tempo di gioco, il nome del gioco e una stringa che serve per il nome del file.

```
public static string GameTitle = "Spearpoint";
public static int GameWidth = 640;
public static int GameHeight = 1000;
public static string Filedati = "filedati";
public static SpriteBatch SpriteBatch;
public static ContentManager Content;
public static GameTime GameTime;
```

4.7 ENEMIES

La classe **Enemies** gestisce la lista di asteroidi (la lista si chiama “Enemies” poiché inizialmente pensavo fossero gli unici nemici del gioco). Al suo interno vengono dichiarati i seguenti elementi:

```
        private List<Enemy> nemici;
        private double spawnrate;
        private double upgrade = 15000;
        private double upgrade2 = 50000;
        private Random random = new Random();
```

Lo spawnrate serve per scandire i tempi di spawn degli asteroidi. I due upgrade servono per dare un countdown all’upgrade che subiscono gli asteroidi col passare del tempo.

Il costruttore crea la lista di nemici.

```
public Enemies()
        {
            nemici = new List<Enemy>();
        }
```

E anche qui facciamo una funzione che aggiunga gli asteroidi alla lista.

```
public void Add(Enemy e)
        {
            nemici.Add(e);
        }
```

Nell’**Update** della lista andiamo a impostare i due cooldown per l’upgrade che mano a mano andranno a scalare. Invece nello spawnrate, una volta che scade, viene aggiunto un nuovo proiettile alla lista e per ogni proiettile che viene creato ed è presente nella PlayScene() viene aggiornato.
Creaiamo una variabile RandX che servirà per decidere la X dell’asteroide, essendo che la peculiarità dell’asteroide è quella che apparirà sempre dall’alto

```
public void Update()
        {
            int randX = random.Next(0 + 40, Common.GameWidth - 40);

            if(upgrade > 0 && upgrade2 > 0)
```

```
        {
            upgrade -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
            upgrade2 -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
        }

        if(upgrade <= 0)
        {
            foreach (Enemy e in nemici)
            {
                e.speed = 15;
            }
        }

        if (upgrade2 <= 0)
        {
            foreach (Enemy e in nemici)
            {
                e.speed = 20;
            }
        }

        if (spawnrate > 0)
        {
            spawnrate -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
        }

        if(spawnrate <= 0)
        {
            nemici.Add(new Enemy("asteroid1", new Vector2(randX, 0)));
            spawnrate = 500;
        }

        foreach (Enemy e in nemici)
        {
            e.Update();
        }
    }
```

4.7 ENEMY

La classe **Enemy** invece gestisce il nemico singolo e ha questi attributi nella sua classe:

```
    public Texture2D texture;
    public Vector2 position;
    public int speed;
    public bool isVisible;
    private Random random = new Random();
```

Il costruttore del nemico, chiede come parametro la texture e la posizione del nemico. La velocità del nemico inizialmente, ho deciso di metterla a 0, in modo che sia un buon livello 0 per abituarsi alla velocità.

```
public Enemy(string textureName, Vector2 position)
        {
            isVisible = true;
            texture = Common.Content.Load<Texture2D>(textureName);
            this.position = position;
            speed = 10;
        }
```

Al contrario dei proiettili, i nemici andranno verso il basso.

```
public void Update()
        {
            position.Y += speed;
        }
```

4.8 Game1

La classe Game1 gestisce il programma principale. Gli attributi memorizzati nella classe sono la scena iniziale, la scena di aiuto, la scena di gioco, la LeaderBoard, la schermata di GameOver e il suono di effetto, insieme agli oggetti graphics e SpriteBatch.

```
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        StartScene startScene;
        HelpScene helpScene;
        PlayScene playScene;
        LeaderboardScreen leaderboardScreen;
        GameOverScreen gameoverScreen;
        Sound menueffect;
```

Il costruttore imposta la larghezza e l'altezza della finestra e il titolo del videogioco insieme all'effetto.

```
public Game1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
            Common.Content = Content;
            Window.Title = Common.GameTitle;
            graphics.PreferredBackBufferWidth = Common.GameWidth;
            graphics.PreferredBackBufferHeight = Common.GameHeight;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            menueffect = new Sound("openmenu_effect", SoundType.Effect);
        }
```

Il metodo **LoadContent** crea e inizializza i precedenti oggetti. Il metodo crea e imposta tutte le scene di gioco.

```
protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);
            Common.SpriteBatch = spriteBatch;
            startScene = new StartScene();
            helpScene = new HelpScene();
            playScene = new PlayScene();
            gameoverScreen = new GameOverScreen();
            leaderboardScreen = new LeaderboardScreen(playScene);
        }
```

Il metodo **Update** aggiorna lo stato delle scene.

```
protected override void Update(GameTime gameTime)
        {
            Common.GameTime = gameTime;
            KeyboardInput.Update();
            startScene.Update();
            HandleScenes();
            playScene.Update();
            gameoverScreen.Update();
            base.Update(gameTime);
        }
```

Il metodo **Draw** disegna tutte le scene.

```
protected override void Draw(GameTime gameTime)
        {
            GraphicsDevice.Clear(Color.Black);
            Common.SpriteBatch.Begin();
            startScene.Draw();
            helpScene.Draw();
            playScene.Draw();
            gameoverScreen.Draw();
            leaderboardScreen.Draw();
            Common.SpriteBatch.End();
            base.Draw(gameTime);
        }
```

Il metodo **HandleScenes** è un metodo che abbiamo utilizzato per gestire l'apparizione delle scene in base a cosa si sceglie nel Menu. Ovviamente per comprendere a pieno questo metodo bisogna andare a vedere la classe delle **Scene** e il resto delle scene. Per vedere cosa fanno le scene basta tornare nelle prime pagine e vedere il loro utilizzo.

```
private void HandleScenes()
        {
            if (Scene.Active == ActiveScene.Start)
            {
                if (KeyboardInput.KeyPressed(Keys.Enter))
                {
                    if (startScene.Menu.SelectedIndex == StartScene.SEL_NEW_GAME)
                    {
                        menueffect.Play();
                        Scene.Active = ActiveScene.Play;
                        playScene = new PlayScene();
                    }
                    else if (startScene.Menu.SelectedIndex == StartScene.SEL_HELP)
                    {
                        menueffect.Play();
                        Scene.Active = ActiveScene.Help;
                    }
                    else if(startScene.Menu.SelectedIndex == StartScene.SEL_LEADERBOARD)
                    {
                        menueffect.Play();
                        leaderboardScreen = new LeaderboardScreen(playScene);
                        Scene.Active = ActiveScene.Leaderboard;

                    }
                    else if (startScene.Menu.SelectedIndex == StartScene.SEL_EXIT)
                    {
                        Exit();
                    }
                }
            }
            else if (Scene.Active == ActiveScene.Play || Scene.Active == ActiveScene.Help ||
Scene.Active == ActiveScene.Leaderboard)
            {
                if (KeyboardInput.KeyPressed(Keys.Escape))
                {
                    menueffect.Play();
                    startScene.Initialize();
                    Scene.Active = ActiveScene.Start;
                }
            }
            else if(Scene.Active == ActiveScene.GameOver)
            {
                if(KeyboardInput.KeyPressed(Keys.Enter))
                {
                    if(gameoverScreen.Menu.SelectedIndex == GameOverScreen.SEL_CONTINUE)
                    {
                        menueffect.Play();
                        Scene.Active = ActiveScene.Play;
                        playScene = new PlayScene();
                    }
                    else if(gameoverScreen.Menu.SelectedIndex == GameOverScreen.RETURN_TO_MENU)
                    {
                        menueffect.Play();
                        Scene.Active = ActiveScene.Start;

                    }
                    else if(gameoverScreen.Menu.SelectedIndex == GameOverScreen.SEL_EXIT)
                    {
```

```
                    Exit();
                }
            }
        }
    }
```

4.9 GAMEMENU

La classe **GameMenu** serve per gestire i menu del gioco. Questa classe è composta da i seguenti attributi: il font, la posizione del menu, il colore standard, il colore selezionato, l'effetto, l'indice dell'Item selezionato, la lista di Item e una variabile per l'incremento.

```
    private SpriteFont font;
    private Vector2 position;
    private Color standardColor;
    private Color selectedColor;
    private Sound menueffect;
    private int selectedIndex;
    private List<string> items;
    private int increment;
```

Il costruttore inizializza la posizione a zero e il colore standard bianco e quello selezionato ciano. Viene caricato il nome del font e creato l'effetto.

```
public GameMenu(string fontName)
    {
        font = Common.Content.Load<SpriteFont>(fontName);
        items = new List<string>();
        position = Vector2.Zero;
        standardColor = Color.White;
        selectedColor = Color.Cyan;
        selectedIndex = 0;
        increment = 50;
        menueffect = new Sound("menu_effect", SoundType.Effect);
    }
```

Nell'Update invece, viene gestita la navigazione del menu.

```
    public void Update()
    {
        if (KeyboardInput.KeyPressed(Keys.Down))
        {
            menueffect.Play();
            selectedIndex++;
            if (selectedIndex == items.Count) selectedIndex = 0;
        }

        else if (KeyboardInput.KeyPressed(Keys.Up))
```

```
            {
                menueffect.Play();
                selectedIndex--;
                if (selectedIndex == -1) selectedIndex = items.Count - 1;
            }
        }
```

il **Draw** disegna il menu in base a quanti sono stati inseriti.

```
 public void Draw()
        {
            Color color;
            float y = position.Y;
            for (int i = 0; i < items.Count; i++)
            {
                color = standardColor;
                if (i == selectedIndex) color = selectedColor;
                Common.SpriteBatch.DrawString(font, items[i], new Vector2(position.X, y),
color);
                y += increment;
            }
        }
```

4.10 HIGHSCORES

La classe HighScore serve per gestire i punteggi più alti che sono salvati nell'apposito file dati. HighScore possiede soltatno due attributi, un array di punteggi e una costante che definisce quanti dati ci possono essere nella classifica.

```
        public int[] score;
        const int max_scorable = 3;
```

Il costruttore di HighScores crea l'array e legge il file dati attuale in modo tale che quando si riapre il gioco, i punteggi non verranno persi.

```
        public HighScores()
        {
            score = new int[max_scorable];
            LeggiFileDati(Common.Filedati);
        }
```

Le funzioni su cui mi sono appoggiato sono le stesse che abbiamo utilizzato nella prima relazione di laboratorio con le Windows Form. Queste due funzioni sono **LeggiFileDati** e **ScriviFileDati** che come si può capire dal nome servono corrispettivamente a leggere e a scrivere il file dati.

```
public void ScriviFileDati(string nomeFile)
        {
            FileStream file = new FileStream(nomeFile, FileMode.Create);
            BinaryFormatter bf = new BinaryFormatter();
            bf.Serialize(file, score);
            file.Close();
        }


public bool LeggiFileDati(string nomeFile)
        {
            bool leggi = false;
            if (File.Exists(nomeFile))
            {
                FileStream file = new FileStream(nomeFile, FileMode.Open);
                BinaryFormatter bf = new BinaryFormatter();
                score = (int[])bf.Deserialize(file);
                file.Close();
                leggi = true;
            }
            return leggi;
        }
```

4.11 HUD

La classe HUD gestisce l'HUD del gioco che si trova in alto a sinistra per il il punteggio e il tempo. Come attributi possiamo trovare:

```
        private SpriteFont font;
        private Vector2 position;
        private Color color;
        private Spaceship spaceship;
        private double elapsed;
        public string text;
```

Il costruttore della classe HUD chiede come parametro il nome del font e dell'astronave (essendo che dobbiamo prendere i suoi attributi).

```
        public HUD(string fontName, Spaceship spaceship)
        {
            font = Common.Content.Load<SpriteFont>(fontName);
            position = Vector2.Zero;
            color = Color.White;
            this.spaceship = spaceship;
        }
```

Il metodo **Draw** in questo caso, fa aumentare la variabile **elapsed** che abbiamo dichiarato all'inizio, solo che per questi numeri ho deciso di utilizzare **TotalSeconds** in modo da ottenere un intero e non un double lunghissimo. E poi ovviamente disegna la HUD.

```
public void Draw()
        {
            elapsed += Common.GameTime.ElapsedGameTime.TotalSeconds;
            text = "Time: " + (int)Math.Truncate(elapsed) + " | Score: " + spaceship.Timealive;
            Common.SpriteBatch.DrawString(font, text, position, color);
        }
```

4.12 KeyboardInput

Con la classe KeyboardInput gestiamo il modo in cui vengono i presi da tastiera. Come attributi abbiamo due oggetti di tipo KeyboardState.

```
        private static KeyboardState oldKey;
        private static KeyboardState newkey;
```

Non ha un costruttore ma solo tre metodi: l'**Update** aggiorna lo stato della tastiera.

```
        public static void Update()
        {
            oldKey = newkey;
            newkey = Keyboard.GetState();
        }
```

E poi ci sono due funzioni che servono per rilevare due momenti del tasto diversi. **KeyPressed** serve per rilevare se il tasto è stato premuto e rilasciato.

```
public static bool KeyPressed(Keys key)
        {
            return (oldKey.IsKeyDown(key) && newkey.IsKeyUp(key));
        }
```

KeyDown invece serve per rilevare solo se il tasto è stato rilasciato.

```
public static bool KeyDown(Keys key)
        {
            return (newkey.IsKeyDown(key));
        }
```

4.13 Logo

Logo è una classe che ho dovuto inserire per inserire un immagine diversa dal background e forse inserire due background non avrebbe avuto molto senso.

Di sé la classe è molto semplice: ha come attributi la texture e la posizione. Nel costrutte vengono richiesti come parametri, il nome della texture, la x e a la y su dove andrebbe messo il logo e poi con il metodo **Draw** viene disegnato.

```
private Texture2D texture;
private Vector2 position;

public Logo(string textureName, int x, int y)
{
    texture = Common.Content.Load<Texture2D>(textureName);
    position.X = x;
    position.Y = y;
}

public void Draw()
{
    Common.SpriteBatch.Draw(texture, position, Color.White);
}
```

4.14 POWERUP

La classe PowerUp gestisce il potenziamento singolo nel gioco. Nella classe PowerUp vi sono presenti gli attributi texture, posizione, velocità, la visibilità, e una variabile per randomizzare.

```
private Texture2D texture;
private Vector2 position;
public int speed;
public bool isVisible;
private Random random = new Random();
```

Il costruttore richiede il nome della texture e la posizione. La velocità ho deciso di impostarla a 20 per rendere più difficile e ardua la cattura del powerup. Tuttavia non cambierà velocità nel corso del gioco.

```
public PowerUp(string textureName, Vector2 position)
{
    isVisible = true;
    texture = Common.Content.Load<Texture2D>(textureName);
    this.position = position;
    speed = 20;
}
```

L'**Update** sarà un semplice update e solito update, preso direttamente dalla classe **Enemy.**

```
public void Update()
{
    position.Y += speed;
}
```

4.15 POWERUPS

La classe **PowerUps** è simile a tutte le altre liste che abbiamo fatto in precedenza e serve per gestire la lista di powerups. Abbiamo i seguenti attributi:

```
private List<PowerUp> potenziamenti;
private double spawnrate = 1000;
private Random random = new Random();
```

Il costruttore crea la lista di PowerUps. La variabile spawnrate serve – come abbiamo fatto per **Enemy** e **Ufo** – per scandire il tempo di spawn dei PowerUp.

```
public List<PowerUp> Potenziamenti
{
    get { return potenziamenti; }
}
```

Il metodo **Update** è simile a quelli usati i precedenza. Lo Spawnrate una volta esaurito aggiunge un nuovo potenziamento risettando lo spawnrate a 7 secondi. 7 secondi perché era abbastanza per non abusare del PowerUp durante il gioco. E ovviamente per ogni powerup nella lista, viene aggiornato. Il **Draw** invece, disegnerà il potenziamento solo nel caso in cui fosse visibile.

```
public void Update()
{

    int randX = random.Next(0, Common.GameWidth);

    if (spawnrate > 0)
    {
        spawnrate -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
    }

    if (spawnrate <= 0)
    {
```

```
            potenziamenti.Add(new PowerUp("boltsprite", new Vector2(randX, 0)));
            spawnrate = 7000;
        }

        foreach (PowerUp p in potenziamenti)
        {
            p.Update();
        }
    }

    public void Draw()
    {
        foreach (PowerUp p in potenziamenti)
        {
            if (p.isVisible)
            {
                p.Draw();
            }

        }
    }
```

4.16 SCENE

La classe **Scene** serve per gestire le scene in tutto. Usufruiamo dell'enumerazione **ActiveScene** che ci permette di selezionare poi il tipo di scena grazie alla variabile statica di tipo **ActiveScene** che all'inizio viene impostata a Start.

```
enum ActiveScene { Start, Play, Help, Leaderboard, GameOver }

    class Scene
    {
        public static ActiveScene Active = ActiveScene.Start;
```

4.17 SOUND

Anche qui, come nella classe **Scene** utilizziamo un enumerazione, in modo da far capire se stiamo utilizzando un effetto o una song.

```
enum SoundType { Effect, Song }
```

Gli attributi saranno: un oggetto **SoundEffect e un oggetto Song** presi dal Framework di XNA dell'Audio. E poi ovviamente il nostro SoundType.

```
        private SoundEffect effect;
        private Song song;
        private SoundType type;
```

Con il costruttore di Sound vogliamo passargli il nome del suono e il tipo di suono che vogliamo: se è un effetto verrà caricato l'effetto e la song messa null mentre se è una song, tutto il contrario.

```
public Sound(string soundName, SoundType type)
{
    this.type = type;
    if (type == SoundType.Effect)
    {
        song = null;
        effect = Common.Content.Load<SoundEffect>(soundName);
    }
    else if (type == SoundType.Song)
    {
        song = Common.Content.Load<Song>(soundName);
        effect = null;
    }
}
```

Il metodo **Play,** avrà due funzioni diverse a seconda di quale tipo di audio è stato scelto. Quindi se è un effetto non verrà ripetuto, mentre se è una song verrà ripetuta una volta finita.

*

Il problema che ho riscontrato nel mio codice, è che a quanto pare non vuole accettarmi le Song come estensione ".mp3". Ho riprovato in tutti i modi ma da quanto ho capito dovrebbe essere un problema all'interno dei file che non riesco a sistemare.

Il metodo **Stop ferma la canzone.**

```
public void Stop()
{
    if (type == SoundType.Song)
    {
        MediaPlayer.Stop();
    }
}
```

4.18 SPACESHIP

La classe **Spaceship** è forse la la classe più lunga del videogioco. Contiene variabili come la texture, la posizione, la classifica, la velocità, tutti i tasti che servono per muovere la spaceship e il tasto per sparare, flag per vedere se l'astronave sta sparando, se è viva o se il powerup è attivato, due variabili che definiscono il cooldown di sparo e di powerup e infine una variabile che conta il tempo di vita.

```
private Texture2D texture;
private Vector2 position;
private HighScores classifica;
private int speed;
private Keys keyUp;
private Keys keyDown;
private Keys keyRight;
private Keys keyLeft;
private Keys shoot;
private bool is_attacking;
private bool is_alive;
private bool powerup_activated;
private double cooldown;
private double powerupCooldown;
private int timeAlive;
```

Il costruttore inizializza tutte le variabili precedenti, impostando la velocità a 10 di base.

```
public Spaceship(string textureName)
{
    texture = Common.Content.Load<Texture2D>(textureName);
    speed = 10;
    position = Vector2.Zero;
    keyDown = Keys.Down;
    keyUp = Keys.Up;
    keyRight = Keys.Right;
    keyLeft = Keys.Left;
    shoot = Keys.E;
    is_attacking = false;
    cooldown = 0;
    is_alive = true;
    timeAlive = 0;
    powerup_activated = false;
    classifica = new HighScores();
}
```

Con la funzione **Update** l'ho divisa in due: se l'astronave è viva allora può usare i comandi e scalano i cooldown dello sparo e del powerup. Se invece

non è vivo vuol dire che non può utilizzare più i comandi e viene attivata la scena di GameOver. Col GameOver viene descritto un For che per tre volte controlla se è stato registrato un punteggio più alto di quello che si trova nella classifica attuale. Se lo è allora lo sostituisce. Questo per le tre posizioni che grazie a delle variabili temporanee tengono i punteggi rimanenti. In ogni caso, riscriverò il file dati con la funzione **ScriviFileDati.**
Ovviamente per evitare che l'astronave esca dai confini facciamo un Clamp per limitarla.

```
public void Update()
{
    is_attacking = false;

    if (cooldown > 0)
    {
        cooldown -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
    }

    if(is_alive)
    {
        timeAlive++;

        if (KeyboardInput.KeyDown(KeyUp))
        {
            position.Y -= speed;
        }
        else if (KeyboardInput.KeyDown(keyDown))
        {
            position.Y += speed;
        }
        else if (KeyboardInput.KeyDown(keyRight))
        {
            position.X += speed;
        }
        else if (KeyboardInput.KeyDown(keyLeft))
        {
            position.X -= speed;
        }

        if (KeyboardInput.KeyPressed(shoot) && cooldown <= 0)
        {
            is_attacking = true;
            if(!powerup_activated)
            {
                cooldown = 300;
            } else
            {
                cooldown = 100;
            }
        }

        if(powerup_activated && powerupCooldown > 0)
```

```
            {
                powerupCooldown -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
            }

            if(powerupCooldown < 0)
            {
                powerup_activated = false;
            }

        }

        if(!is_alive)
        {
            Scene.Active = ActiveScene.GameOver;

            //tempo che tiene il secondo posto
            int temp = -1;
            //tempo che tiene il terzo posto
            int SubTime = -1;

            for (int i = 0; i < classifica.Max_scorable; i++)
            {
                if(timeAlive > classifica.Score[i] && temp < 0)
                {
                    temp = classifica.Score[i];
                    classifica.Score[i] = timeAlive;
                }
                else if(temp >= 0 && SubTime < 0)
                {
                    SubTime = Classifica.Score[i];
                    classifica.Score[i] = temp;

                } else if(temp >= 0)
                {
                    Classifica.Score[i] = SubTime;
                }
            }

            classifica.ScriviFileDati(Common.Filedati);
        }

        position.Y = MathHelper.Clamp(position.Y, 0, Common.GameHeight - texture.Height);
        position.X = MathHelper.Clamp(position.X, 0, Common.GameWidth - texture.Width);
    }
```

Col **Draw** lo disegniamo soltanto se l'astronave è viva.

```
    public void Draw()
    {
        if (is_alive)
        {
            Common.SpriteBatch.Draw(texture, position, Color.White);
        }
    }
```

4.19 UFO

La classe Ufo gestisce l'ufo singolo. Avevo il bisogno di inserire un nuovo nemico che fosse leggermente diverso da quell'asteroide. Nei suoi attributi troviamo quelli identici usati per l'Enemy.

```
private Texture2D texture;
private Vector2 position;
public int speed;
public bool isVisible;
private Random random = new Random();
```

Il costruttore dell'ufo prende il nome della texture e la posizione:

```
public Ufo(string textureName, Vector2 position)
{
    isVisible = true;
    texture = Common.Content.Load<Texture2D>(textureName);
    this.position = position;
    speed = 10;
}
```

Nell'Update diversamente dall'asteroide lo faremo scorrere sulla X.

```
public void Update()
{
    position.X += speed;
}
```

E poi verra disegnato col **Draw.**

```
public void Draw()
{
    Common.SpriteBatch.Draw(texture, position, Color.White);
}
```

4.20 UFOS

La classe Ufos, gesisce la lista di Ufo. Come attributi abbiamo i soliti:

```
private List<Ufo> aliens;
private double spawnrate;
private double delay = 30000;
private Random random = new Random();
```

Il Delay serve per dare una sorta di ritardo per quando appariranno gli UFO (quindi al secondo 30) Con il costruttore della classe vogliamo creare la lista:

```
public Ufos()
{
    aliens = new List<Ufo>();
}
```

La classica funzione **Add** che aggiunge un ufo alla lista, passandogli come parametro un ufo.

```
public void Add(Ufo u)
{
    aliens.Add(u);
}
```

L'**Update** è pressochè simile a quello dell'asteroide con l'unica differenza che ci sarà un delay e al posto di spawnare in cima alla finestra, gli ufo usciranno da lato, complicando la vita dello **SpearPointer77.**

```
public void Update()
{
        if(delay > 0)
        {
            delay -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
        }

        int randY = random.Next(0, Common.GameHeight);

        if (spawnrate > 0)
        {
            spawnrate -= Common.GameTime.ElapsedGameTime.TotalMilliseconds;
        }

        if (spawnrate <= 0 && delay <= 0)
        {
            aliens.Add(new Ufo("ufo", new Vector2(0, randY)));
            spawnrate = 3000;
        }

        foreach (Ufo u in aliens)
        {
            u.Update();
        }
}
```

Infine il **Draw** disegnerà l'ufo solo è vivo.

```
public void Draw()
{
    foreach (Ufo u in aliens)
    {
        if (u.isVisible)
        {
            u.Draw();
        }

    }
}
```

5. SITOGRAFIA

Non ho consultato molti siti per aiutarmi a fare il videogioco. Utilissimi sicuramente sono stati:

https://docs.microsoft.com/
https://stackoverflow.com/

Che mi hanno aiutatato per quando avevo un dubbio su qualche classe e soprattutto se avevo problemi nella creazione del codice. (senza contere l'incredibile generosità del prof, con le sue email di supporto).