

Konstantinos Sideris | CS188 | December 1, 2013

Galactic Wars

An introduction to Directx 2d Game programming

# Game objects

The game engine contains the following types of objects:

* **Characters** (Player/Enemies):
  + HorizontalSliderPlayer.cpp is the player Class
  + Enemy.cpp is the enemy class
* **Backgrounds**
  + SlidingBackgroundSprite.cpp is the main moving background class
  + FlashingBackground.cpp handles the health indicator background
* **Game Elements**:
  + GameElements.cpp handles the collectible items, the finish line and the powerups
* **Particle Effects**:
  + Handles the projectiles, the rocket fuel and the explosions

The hierarchy for the Game Engine is shown in figure [\*]

Sprite

Colliding Object

GameObject

Background

The hierarchy for GameObjects is shown in figure [\*]

Most of these objects are initialized in the Levels.cpp file with exception of the Player object (SpriteGame.cpp). If you just want to modify the appearance of the player, all you have to is change the Spaceship.png icon in the Assets.

# Collision detection/handling

The main collision class is the **CollidingObject** class (in CollidingObject.cpp). It defines a collision polygon and a set of functions to detect collision. Every GameObject inherits from the class, so technically every GameObject can collide with other objects.

Collision detection relies on the Separating Axis Theorem and works with all convex polygons.

The renderer (SpriteGame.cpp) is responsible for calling the collision detection appropriately (in function void SpriteGame::DetectCollisions(); , Ln 209, SpriteGame.cpp )

Collision Handling is also taken care of by void SpriteGame::HandleCollisions(); Ln 252, SpriteGame.cpp. Each object is responsible for reporting the result of a collision trough the GameObject function ProcessHit().

# Level loading

The level loading can be broken down to two parts. The file Levels.xml contains information about the levels that are going to be loaded into the engine. That information is used by DirectXPage.xaml.cpp to create a level selection UI in function void DirectXPage::LoadLevels(String^ levels\_xml) , Ln 741.

To add a new level, you need to declare it in the Levels.xml with a name of your choosing and the filename of the xml for that level. E.g

<Level Name="Andromeda Galaxy" Data="Andromeda" > </Level>

Where the attribute Name defines the display name and the attribute Data contains the filename for that level. New xml files should be stored in the folder Level Data ( in visual studio make sure to include it in the project).

The second part is the level itself. Obviously we have one XML for each level. This looks like this:

Background music for the level (has to be stored in Assets/Music and included in the Visual Studio project)

<?xml version="1.0"?>

<Level>

Background image for the level (has to be stored in Assets/GameObjects and included in the Visual Studio project). Think of it as the sky in a game.

The foreground consists of two textures. Think of it as the clouds.

<Theme track="DST-10Class.wav" ></Theme>

<Background Texture="m32.png"></Background>

<Foreground Texture="GeometrySprite.png"

Texture2="GeometrySprite2.png">

Textures for the rest of the Objects

</Foreground>

<Asteroid Texture="ida.dds"> </Asteroid>

<RingTexture Texture="ring.png"> </RingTexture>

<Projectile Texture="particle.png"> </Projectile>

Textures and EnemyTypes.

<EnemyType Texture="Enemy1.png"></EnemyType>

<EnemyType Texture="Enemy2.png"></EnemyType>

<EnemyType Texture="Enemy3.png"></EnemyType>

<EnemyType Texture="Enemy4.png"></EnemyType>

<HealthTexture Texture="life\_upgrade.png"></HealthTexture>

Enemy spawning. The attribute Pos is relative to the total time passed. Type is one of the types registered above. Movem defines the movement type. Right now , RANDOM (0) and UP&DOWN(1) are supported

<PowerupTexture Texture="weapons\_upgrade.png"></PowerupTexture>

<Enemy Type="0" Pos="3" Movem="0"></Enemy>

Asteroid Fields refer to a cluster of asteroids. The attribute count is the number of asteroids to generate

Rings are the collectible stars

Health is the health powerup, Powerup is the weapons power and Level End is the finish line

<Enemy Type="0" Pos="15" Movem="1"></Enemy>

<Enemy Type="0" Pos="22" Movem="0" ></Enemy>

<Enemy Type="0" Pos="30" Movem="0"></Enemy>

<AsteroidField Pos="10" Count="10"> </AsteroidField>

<AsteroidField Pos="20" Count="30"> </AsteroidField>

<Ring Pos="1" Count="10"> </Ring>

<Ring Pos="20" Count="10"> </Ring>

<Powerup Pos="10" ></Powerup>

<Powerup Pos="67" ></Powerup>

<Health Pos="5" ></Health>

<LevelEnd Pos="200" Texture="finishline.png"></LevelEnd>

</Level>

# User Interface

The UI is based on XAML. There are three root elements in the DirectXPage.xaml that define the UI. The GamePlayGrid contains the Heads Up Display. The MenuButtonsGrid and LevelButtonsGrid define the Start Page and level selection page respectively. Since XAML in a DirectXPage is limited to one single page, switching between those interfaces is achieved by hiding and showing them appropriately (in DirectXPage.xaml.cpp)

There are also the Popup elements, which are the views of the settings page, loaded based on the settings command that was selected.

# Settings

Settings and high scores persist. In the case of the settings, this is achieved by using LocalSettings. ApplicationDataContainer^ localSettings. Check the following helper functions for more details:

void DirectXPage::Save(Platform::String^ key, Platform::Object^ value);

Platform::Object^ DirectXPage::Read(Platform::String^ key);bool DirectXPage::Exists(Platform::String^ key);

On the other hand, high scores are stored and loaded from a comma separated text file. See for more details.

void DirectXPage::CheckNCreateFile(String^ Filename);

void DirectXPage::LoadHighScores(String^ Filename);

void DirectXPage::UpdateScores();

# Audio

Audio in the galactic wars game engine is managed by the AudioManager class (in Audio/AudioManager.cpp).

Audio initialization happens in the LoadSettings (Line 552, Audio/AudioEngine.h). Function of DirectXPage.xaml.cpp

There are 2 types of audio:

Sound Effects

To define and load a new sound effect, three steps are required:

1. Define a new Sound Effect name in the enum SoundEvent (Line 19, Audio/AudioEngine.h).

E.g. Shoot

1. Add the following line to the Audio::CreateResources() function (Line 147, Audio/AudioEngine.cpp):

E.g. CreateSourceVoice (Shoot);

1. Add the location of the sound effect file in void Audio::CreateSourceVoice(SoundEvent sound) (Line 216, Audio/AudioEngine.cpp):

E.g.

case Shoot: soundEffectStream.Initialize(L"Assets\\Sounds\\Shoot.wav"); break;

Hint: refrain from using file types other than .wav since the extra decoding can cause performance issues.

To use the sound effect you just registered, add the following lines where you want the sound to play:

AudioManager::AudioEngineInstance.StopSoundEffect(AudioEngine::SoundEvent::Shoot); AudioManager::AudioEngineInstance.PlaySoundEffect(AudioEngine::SoundEvent::Shoot);

Background Music

Background music is handled by the AudioManager. You need to call (Line 52, Audio/AudioManager.cpp):

void AudioManager::SetGamePlayMusic(const wchar\_t\* musicurl)

to set a new background music file. E.g.:

AudioManager::SetGamePlayMusic ("Assets\\Music\\maintheme.wav");

Currently, Galactic Wars loads music in the level loader by reading the following property in the level xml:

<Theme track="DST-10Class.wav" ></Theme>

The file needs to be stored under "Assets\\Music” to be correctly loaded. GameOver music is handled by calling void AudioManager::SetGameOverMusic(), so you need to modify the loaded music there or use SetGamePlayMusic(“”)

Note: For the background music to be correctly played, the main rendering loop needs to call

AudioManager::AudioEngineInstance.Render();