

UGE

Developer's Reference

A game engine for Universally Accessible Games



Franco Eusébio Garcia

Last Update:

March 13, 2014

Contents

[Contents 2](#_Toc384138985)

[Figures 6](#_Toc384138986)

[Tables 8](#_Toc384138987)

[Listings 9](#_Toc384138988)

[1. Introduction 12](#_Toc384138989)

[1.1. Introduction 12](#_Toc384138990)

[1.2. Context and Motivation 12](#_Toc384138991)

[1.2.1. Game Accessibility Guidelines and Best Practices 12](#_Toc384138992)

[1.2.2. Universally-Accessible Games (UA-Games) 12](#_Toc384138993)

[1.2.3. Designing UA-Games: The Unified Design 12](#_Toc384138994)

[1.3. Goals 12](#_Toc384138995)

[1.4. Design and Architecture Considerations 13](#_Toc384138996)

[2. UGE Overview 15](#_Toc384138997)

[2.1. Introduction 15](#_Toc384138998)

[2.2. UGE Engine 15](#_Toc384138999)

[2.3. UGE IO 16](#_Toc384139000)

[2.4. UGE Core 16](#_Toc384139001)

[2.5. UGE Utilities 17](#_Toc384139002)

[3. UGE Engine 18](#_Toc384139003)

[3.1. Introduction 18](#_Toc384139004)

[3.2. Game Application Layer 18](#_Toc384139005)

[3.2.1. Game Application Functionality 18](#_Toc384139006)

[3.2.2. Game Application Architecture 19](#_Toc384139007)

[3.2.3. Game Application Run-Time 20](#_Toc384139008)

[3.2.4. Remarkable Methods 22](#_Toc384139009)

[3.3. Game Logic Layer 23](#_Toc384139010)

[3.3.1. Game Logic Functionality 23](#_Toc384139011)

[3.3.2. Game Logic Architecture 30](#_Toc384139012)

[3.3.3. Game Logic Run-Time 32](#_Toc384139013)

[3.3.4. Remarkable Methods 35](#_Toc384139014)

[3.4. Game View Layer 36](#_Toc384139015)

[3.4.1. Game View Functionality 36](#_Toc384139016)

[3.4.2. Game View Architecture 41](#_Toc384139017)

[3.4.3. Game View Run-Time 43](#_Toc384139018)

[3.4.4. Remarkable Methods 45](#_Toc384139019)

[4. UGE Core 47](#_Toc384139020)

[4.1. Introduction 47](#_Toc384139021)

[4.2. Entities (Actors) and Components 47](#_Toc384139022)

[4.2.1. Functionality 50](#_Toc384139023)

[4.2.2. Architecture 50](#_Toc384139024)

[4.2.3. Run-Time 54](#_Toc384139025)

[4.2.4. Example 55](#_Toc384139026)

[4.3. Events 58](#_Toc384139027)

[4.3.1. Functionality 60](#_Toc384139028)

[4.3.2. Architecture 60](#_Toc384139029)

[4.3.3. Run-Time 62](#_Toc384139030)

[4.3.4. Example 63](#_Toc384139031)

[4.4. Game Commands 67](#_Toc384139032)

[4.4.1. Functionality 68](#_Toc384139033)

[4.4.2. Architecture 69](#_Toc384139034)

[4.4.3. Run-Time 69](#_Toc384139035)

[4.4.4. Example 70](#_Toc384139036)

[4.5. Physics 71](#_Toc384139037)

[4.5.1. Functionality 71](#_Toc384139038)

[4.5.2. Architecture 71](#_Toc384139039)

[4.5.3. Run-Time 72](#_Toc384139040)

[4.5.4. Example 73](#_Toc384139041)

[4.6. Player Profiles 81](#_Toc384139042)

[4.6.1. Functionality 81](#_Toc384139043)

[4.6.2. Architecture 82](#_Toc384139044)

[4.6.3. Run-Time 83](#_Toc384139045)

[4.6.4. Example 84](#_Toc384139046)

[4.7. Resources 89](#_Toc384139047)

[4.7.1. Functionality 89](#_Toc384139048)

[4.7.2. Architecture 89](#_Toc384139049)

[4.7.3. Run-Time 91](#_Toc384139050)

[4.7.4. Example 91](#_Toc384139051)

[4.8. Scenes 92](#_Toc384139052)

[4.8.1. Functionality 92](#_Toc384139053)

[4.8.2. Architecture 92](#_Toc384139054)

[4.8.3. Run-Time 96](#_Toc384139055)

[4.8.4. Example 96](#_Toc384139056)

[4.9. Scripting 96](#_Toc384139057)

[4.9.1. Functionality 96](#_Toc384139058)

[4.9.2. Architecture 96](#_Toc384139059)

[4.9.3. Run-Time 97](#_Toc384139060)

[4.9.4. Example 97](#_Toc384139061)

[4.10. Tasks 99](#_Toc384139062)

[4.10.1. Functionality 99](#_Toc384139063)

[4.10.2. Architecture 99](#_Toc384139064)

[4.10.3. Run-Time 101](#_Toc384139065)

[4.10.4. Example 101](#_Toc384139066)

[5. UGE IO 104](#_Toc384139067)

[5.1. Introduction 104](#_Toc384139068)

[5.2. Input 104](#_Toc384139069)

[5.2.1. Input Mapping 104](#_Toc384139070)

[5.3. Output 108](#_Toc384139071)

[5.3.1. Audio 111](#_Toc384139072)

[5.3.2. Graphics 116](#_Toc384139073)

[6. UGE Utilities 117](#_Toc384139074)

[6.1. Introduction 117](#_Toc384139075)

[6.2. Debug 117](#_Toc384139076)

[6.3. File 118](#_Toc384139077)

[6.4. Macros 121](#_Toc384139078)

[6.5. Math 121](#_Toc384139079)

[6.6. String 121](#_Toc384139080)

[6.7. System Information 121](#_Toc384139081)

[6.8. Templates 122](#_Toc384139082)

[7. UGE Tutorials 123](#_Toc384139083)

[7.1. Setting Up the Development Environment 123](#_Toc384139084)

[7.1.1. Microsoft Visual Studio 2012 123](#_Toc384139085)

[7.2. Creating a Game 143](#_Toc384139086)

[7.2.1. Unified Design 143](#_Toc384139087)

[7.2.2. Implementation: Initial Header 144](#_Toc384139088)

[7.2.3. Main 148](#_Toc384139089)

[7.2.4. Game Application Layer 149](#_Toc384139090)

[7.2.5. Game Logic Layer 156](#_Toc384139091)

[7.2.6. Game View Layer 202](#_Toc384139092)

[7.2.7. Defining the Gameplay 218](#_Toc384139093)

[7.2.8. Tailoring the Game to Different Interaction Abilities 233](#_Toc384139094)

[7.2.9. Results and Discussion 233](#_Toc384139095)

[References 234](#_Toc384139096)

# Figures

[**Figure 1.** UGE’s layers: GameApplication, GameLogic and GameView. 15](#_Toc384139097)

[**Figure 2.** The Game Application Layer. 19](#_Toc384139098)

[**Figure 3.** Game states. 27](#_Toc384139099)

[**Figure 4.** The Game State Factory. 29](#_Toc384139100)

[**Figure 5.** The Game Logic Layer. 31](#_Toc384139101)

[**Figure 6.** Game command mapping. 37](#_Toc384139102)

[**Figure 7.** UGE Input and Output: the IGameView abstract interface and the HumanGameView. 41](#_Toc384139103)

[**Figure 8.** The IGameView dataflow. 42](#_Toc384139104)

[**Figure 9.** A lower level IGameView dataflow. 42](#_Toc384139105)

[**Figure 10.** An actor with two components attached. 48](#_Toc384139106)

[**Figure 11.** The actor from **Figure 10** with a DrawableComponent. 48](#_Toc384139107)

[**Figure 12.** The actor from **Figure 10** with an AudibleComponent. 49](#_Toc384139108)

[**Figure 13.** The actor from **Figure 10** with an AudibleComponent and a DrawableComponent. 49](#_Toc384139109)

[**Figure 14.** UGE’s entity: the Actor. 50](#_Toc384139110)

[**Figure 15.** The component base class. 51](#_Toc384139111)

[**Figure 16.** Different types of components deriving from ActorComponent. 52](#_Toc384139112)

[**Figure 17.** The ActorFactory. 54](#_Toc384139113)

[**Figure 18.** A high-level event. 58](#_Toc384139114)

[**Figure 19.** Different layers handling the high-level event of **Figure 18**. 59](#_Toc384139115)

[**Figure 20.** The event abstract interface and base implementation. 60](#_Toc384139116)

[**Figure 21.** Some default events. 61](#_Toc384139117)

[**Figure 22.** The event manager. 62](#_Toc384139118)

[**Figure 23.** Low and high-level game commands. 68](#_Toc384139119)

[**Figure 24.** The low-level game command is a simple value. 69](#_Toc384139120)

[**Figure 25.** The Physics subsystem interface. 72](#_Toc384139121)

[**Figure 26.** The Unified Design for a Space Invaders clone (extracted from [8]). 144](#_Toc384139122)

[**Figure 27.** Checking the game Application: the initial window. 149](#_Toc384139123)

[**Figure 28.** The tutorial game window after setting the Game Application layer. 156](#_Toc384139124)

[**Figure 29.** Finite state machine for the game states. 157](#_Toc384139125)

[**Figure 30.** The first actor created into the tutorial game. 173](#_Toc384139126)

[**Figure 31.** A new iteration on the first actor – adding new components to the archetype. 174](#_Toc384139127)

[**Figure 32.** Using the Physics subsystem debugger to visualize the game world. 175](#_Toc384139128)

[**Figure 33.** The result of changing the collision shape in **Listing 76**. 177](#_Toc384139129)

[**Figure 34.** The game world defined in **Listing 81**. 183](#_Toc384139130)

[**Figure 35.** Using an actor XML resource as an archetype to create many similar actors. 186](#_Toc384139131)

[**Figure 36.** A positive value assigned to InitialHealthPoints (100). 193](#_Toc384139132)

[**Figure 37.** A zero value assigned to InitialHealthPoints (0). 193](#_Toc384139133)

[**Figure 38.** The spaceship (in red) after several random game commands. 197](#_Toc384139134)

[**Figure 39.** The game logic actors, after being overridden by the player profile. 202](#_Toc384139135)

[**Figure 40.** The graphical actor. 205](#_Toc384139136)

[**Figure 41.** Firing some projectiles. 228](#_Toc384139137)

[**Figure 42.** Shooting some projectiles at the enemies. 231](#_Toc384139138)

# Tables

**No table of figures entries found.**

# Listings

[**Listing 1.** The BaseGameApplication’s initialization. 20](#_Toc384139139)

[**Listing 2.** The BaseGameApplication’s game loop. 21](#_Toc384139140)

[**Listing 3.** A sample implementation of the BaseGameApplication’s method vInitOutputSystems(). 23](#_Toc384139141)

[**Listing 4.** A sample game state. 28](#_Toc384139142)

[**Listing 5.** A sample implementation of the method vInitFactory(). 29](#_Toc384139143)

[**Listing 6.** The default initialization of the BaseGameLogic. 32](#_Toc384139144)

[**Listing 7.** Updating the BaseGameLogic. 33](#_Toc384139145)

[**Listing 8.** Updating the physics subsystem and task manager. 33](#_Toc384139146)

[**Listing 9**. Changing game states. 34](#_Toc384139147)

[**Listing 10.** Finishing the game: the default de-initialization of the BaseGameLogic. 34](#_Toc384139148)

[**Listing 11.** A player profile. 39](#_Toc384139149)

[**Listing 12.** The specifications for the PrimaryOutput of **Listing 11**. 40](#_Toc384139150)

[**Listing 13.** Initializing the HumanGameView. 43](#_Toc384139151)

[**Listing 14.** Attaching a human controlled actor to the HumanGameView. 44](#_Toc384139152)

[**Listing 15.** Updating the HumanGameView. 44](#_Toc384139153)

[**Listing 16.** Rendering the HumanGameView. 44](#_Toc384139154)

[**Listing 17.** De-initializing the HumanGameView. 45](#_Toc384139155)

[**Listing 18.** Adding and removing SceneRenderers to the Scene. 45](#_Toc384139156)

[**Listing 19.** After attaching a human actor (**Listing 14**), it might be convenient to set the camera to its position. 46](#_Toc384139157)

[**Listing 20.** A data-driven Actor. 52](#_Toc384139158)

[**Listing 21.** A sample Actor resource. 55](#_Toc384139159)

[**Listing 22.** Creating a game Actor from the resource of **Listing 21**. 55](#_Toc384139160)

[**Listing 23.** A XML resource modifying for modifying an Actor. 56](#_Toc384139161)

[**Listing 24.** Using the Actor as an archetype and overriding its data. 57](#_Toc384139162)

[**Listing 25.** Event subsystem example. 63](#_Toc384139163)

[**Listing 26.** An example of using game commands. 70](#_Toc384139164)

[**Listing 27.** A sample general settings file for the Physics subsystem. 73](#_Toc384139165)

[**Listing 28.** An actor with Physics related components. 74](#_Toc384139166)

[**Listing 29.** Another actor for the simulation: a simple plane. 75](#_Toc384139167)

[**Listing 30.** A simple physics simulation: a bouncing ball. 75](#_Toc384139168)

[**Listing 31.** Adding events to the simulation of **Listing 30**. 78](#_Toc384139169)

[**Listing 32.** The PlayerProfile class. 82](#_Toc384139170)

[**Listing 33.** The PlayerProfile manager. 83](#_Toc384139171)

[**Listing 34.** A list of player profiles. 84](#_Toc384139172)

[**Listing 35**. A sample profile. 84](#_Toc384139173)

[**Listing 36.** The general settings resource. 85](#_Toc384139174)

[**Listing 37.** Input mapping resource. 85](#_Toc384139175)

[**Listing 38.** The graphics subsystem configuration. 86](#_Toc384139176)

[**Listing 39.** The audio subsystem configuration. 87](#_Toc384139177)

[**Listing 40.** A list of game events. 87](#_Toc384139178)

[**Listing 41.** A list of entities to specialize in the profile. 87](#_Toc384139179)

[**Listing 42.** A specialized actor. 88](#_Toc384139180)

[**Listing 43.** Loading the profile data into an application. 89](#_Toc384139181)

[**Listing 44.** The Spaceship game GameApplication. 145](#_Toc384139182)

[**Listing 45.** The Spaceship game GameLogic. 145](#_Toc384139183)

[**Listing 46.** The Spaceship game GameView. 146](#_Toc384139184)

[**Listing 47.** The entry point of the tutorial game. 148](#_Toc384139185)

[**Listing 48.** An initial list of player profiles. 150](#_Toc384139186)

[**Listing 49.** Several player profiles in the list. 150](#_Toc384139187)

[**Listing 50.** The player profile for average users. 150](#_Toc384139188)

[**Listing 51.** The settings for the graphical renderer. 151](#_Toc384139189)

[**Listing 52.** The settings for audio playback. 152](#_Toc384139190)

[**Listing 53.** Creating the output subsystems for the GameApplication. 152](#_Toc384139191)

[**Listing 54.** Creating the game logic. 153](#_Toc384139192)

[**Listing 55.** Creating the game view. 153](#_Toc384139193)

[**Listing 56.** Setting the game title. 154](#_Toc384139194)

[**Listing 57.** Creating and initializing a custom resource cache. 154](#_Toc384139195)

[**Listing 58.** Initializing the Application. 155](#_Toc384139196)

[**Listing 59.** Creating the game Physics. 156](#_Toc384139197)

[**Listing 60.** The game states header. 158](#_Toc384139198)

[**Listing 61.** The initialization game state. 162](#_Toc384139199)

[**Listing 62.** A GameStateFactory for the tutorial game. 163](#_Toc384139200)

[**Listing 63.** Defining the GameStateFactory for the Game Logic layer. 164](#_Toc384139201)

[**Listing 64.** Brainstorming the components data-driven layout. 165](#_Toc384139202)

[**Listing 65.** Defining the HealthComponent. 166](#_Toc384139203)

[**Listing 66.** Implementing the data-driven HealthComponent. 167](#_Toc384139204)

[**Listing 67.** The tutorial game's ActorFactory. 169](#_Toc384139205)

[**Listing 68.** Registering the components to the ActorFactory. 169](#_Toc384139206)

[**Listing 69.** Creating the custom factory in the tutorial’s game logic. 170](#_Toc384139207)

[**Listing 70.** Creating the first actors. 171](#_Toc384139208)

[**Listing 71.** Initializing the game actors. 171](#_Toc384139209)

[**Listing 72.** A draft for the first game actor. 171](#_Toc384139210)

[**Listing 73.** Creating and adding the actor to the game logic. 172](#_Toc384139211)

[**Listing 74.** Attaching a few more components into the actor. 173](#_Toc384139212)

[**Listing 75.** Adding the actor the Physics subsystem. 174](#_Toc384139213)

[**Listing 76.** Changing the collision shape to a box. 175](#_Toc384139214)

[**Listing 77.** The XML resource for enemy actors. 177](#_Toc384139215)

[**Listing 78.** The XML resource for bullet actors. 178](#_Toc384139216)

[**Listing 79.** The XML resource for bomb actors. 179](#_Toc384139217)

[**Listing 80.** Refactoring the Running game state. 180](#_Toc384139218)

[**Listing 81.** Implementing the new methods of the refactor. 181](#_Toc384139219)

[**Listing 82.** Defining many enemies at once using the alien actor archetype. 183](#_Toc384139220)

[**Listing 83.** Creating many enemies at once using the alien actor archetype. 184](#_Toc384139221)

[**Listing 84.** A sample event to communicate the destruction of aliens. 187](#_Toc384139222)

[**Listing 85.** Registering game events handlers. 188](#_Toc384139223)

[**Listing 86.** Removing destroyed actors from the game. 190](#_Toc384139224)

[**Listing 87.** Adding a destroyed alien to test the event. 191](#_Toc384139225)

[**Listing 88.** Defining a game command to move actors. 194](#_Toc384139226)

[**Listing 89.** Handling a game command. 195](#_Toc384139227)

[**Listing 90.** Dispatching a game command with a random direction. 196](#_Toc384139228)

[**Listing 91.** Setting a constant speed to the actor. 197](#_Toc384139229)

[**Listing 92.** Implementing player profile support to the game logic. 199](#_Toc384139230)

[**Listing 93.** Tailoring the entities. 201](#_Toc384139231)

[**Listing 94.** The PlayerPreferences section of the player profile, describing general settings. 203](#_Toc384139232)

[**Listing 95.** Adding actors to the game scene. 203](#_Toc384139233)

[**Listing 96.** Adding an output component to the actor. 204](#_Toc384139234)

[**Listing 97.** The default camera used in the tutorial. 206](#_Toc384139235)

[**Listing 98.** Defining a follow-up camera. 206](#_Toc384139236)

[**Listing 99.** A GameController using mouse and keyboard. 207](#_Toc384139237)

[**Listing 100.** A sample implementation of the InputCallback() method. 208](#_Toc384139238)

[**Listing 101.** Low-level game commands: actions, states and ranges. 209](#_Toc384139239)

[**Listing 102.** A list of input contexts. 210](#_Toc384139240)

[**Listing 103.** All the low-level game commands for a context. 210](#_Toc384139241)

[**Listing 104.** The input mapping for the low-level game commands. 211](#_Toc384139242)

[**Listing 105.** User specific input mapping. 211](#_Toc384139243)

[**Listing 106.** A class to provide aural feedback. 212](#_Toc384139244)

[**Listing 107.** A factory for providing event feedback. 216](#_Toc384139245)

[**Listing 108.** Initializing the aural feedback. 216](#_Toc384139246)

[**Listing 109.** A list of event feedback resources. 218](#_Toc384139247)

[**Listing 110**. A sample configuration file for event feedback. 218](#_Toc384139248)

[**Listing 111.** A game command to fire projectiles (bullets and bombs). 219](#_Toc384139249)

[**Listing 112.** Creating and adding the projectile to the game world. 220](#_Toc384139250)

[**Listing 113.** Dispatching the game command’s event triggers its execution in the game logic. 222](#_Toc384139251)

[**Listing 114.** Register the new game commands to their low-level representation. 223](#_Toc384139252)

[**Listing 115.** Register the new low-level game commands. 223](#_Toc384139253)

[**Listing 116.** Default input mapping with the new game commands. 224](#_Toc384139254)

[**Listing 117.** Profile specific input mapping with the new game commands. 224](#_Toc384139255)

[**Listing 118.** Updating the InputCallback() to add the new game events. 225](#_Toc384139256)

[**Listing 119.** Removing obsolete code. 225](#_Toc384139257)

[**Listing 120.** The new spaceship archetype. 226](#_Toc384139258)

[**Listing 121.** The physical-level specialization for the average user profile. 227](#_Toc384139259)

[**Listing 122.** Handling the collision between an actor and a projectile. 228](#_Toc384139260)

[**Listing 123.** Handling the side effects of the collision. 231](#_Toc384139261)

[**Listing 124.** Moving the game enemies using the game command from **Listing 88**. 232](#_Toc384139262)

# Introduction

## Introduction

This document briefly describes UGE, an open-source game engine to develop accessible games, targeting especially Universally-Accessible games (UA-Games) [9,10] – games that follow the principles of the Design for All [17], aiming to enable as many users as possible, regardless of their (dis)abilities, to play.

UGE combines different approaches to aid game designers and programmers to develop flexible UA-Games, allowing them to tailor the game stimulus to match the users’ interaction needs and abilities. This is possible due to design and architectures choices for the engine. For instance, UGE uses a data-driven architecture with an entity-component approach – thus, in a UGE game, everything is data, including game entities. As such, it is possible to change, tweak and specialize many features of the game without requiring recompilations of the code base.

In UGE, it is possible to define various player profiles to customize the game settings. However, these profiles are not just regular configuration files: they even allow overriding the input and output of the game completely. Thus, it is possible to implement the game logic once, with all its rules and mechanics, and create as many physical-level specializations as desired.

UGE is implemented in C++, using several of the C++11 standard features. By default, developers can implement games in the C++ and/or Lua programming languages – it also is possible to bind new scripting languages to the engine. Although currently Windows only, most of the dependencies are cross-platform, which eases ports to different platforms in the future.

This section is organized as follows: …

## Context and Motivation

Game accessibility importance is increasing…

### Game Accessibility Guidelines and Best Practices

Average user.

### Universally-Accessible Games (UA-Games)

### Designing UA-Games: The Unified Design

## Goals

As mentioned in the previous section, various papers and resources discuss the design and implementation of accessible games. However, there are only a few frameworks and game engines created to ease the development of accessible games – many of which discontinued or not publicly available.

As play testing is very important for the development of accessible games, quick iterations is a key aspect in the development cycle. Thus, UGE tries to fill this void. Developers are full of nice ideas to design games; UGE offers a foundation to prototype and implement the design.

Besides, UA-Games must be flexible from the design to the implementation to support the wide difference in interaction needs. UGE offers the developers an input-output free approach to implement the game logic. This allow the developers to choose, define and even change the input and output of the game to adapt the game commands and presentation and suit them to the user abilities.

## Design and Architecture Considerations

The design and architecture of UGE aims to offer flexibility to tailor the game interaction during run-time. Thus, many design choices favor flexibility over performance; customization and fast iterations over rigid definitions or immutable data.

The main inspirations for UGE are Unity[[1]](#footnote-1), Gameplay[[2]](#footnote-2) and the Game Coding Complete[[3]](#footnote-3) [12,13] engines. UGE uses the Game Coding Complete 4 (GCC4) [12] engine as its foundation, extending and improving its functionality and flexibility to a game accessibility context.

UGE combines data-driven and event-based architectures to entity-component and game command approaches to decouple the game’s logic from its presentation. Thus, it is possible to implement different presentations and input schemes to the game – users are free to choose the combinations that best suit their interaction needs during run-time. This way, the developers are able to offer an accessible and adequate playing experience to user: the engine matches the game logic to the user’s chosen combination and presents the game accordingly.

Contrarily to most game engines available, UGE does not aim to excel in the presentation. In fact, the input and output subsystems are fully abstract in UGE. We believe it is more important to let the developers to decide which technology is the most adequate to the target users. The developers implement the engine’s interfaces and choose which renderer the game will use.

By default, UGE uses Object-Oriented Graphics Rendering Engine[[4]](#footnote-4) (OGRE) as the graphical rendering subsystem (covering OpenGL and DirectX); YSE[[5]](#footnote-5) and TinyOAL[[6]](#footnote-6) as options for the audio subsystem (both using OpenAL); and Object Oriented Input System[[7]](#footnote-7) (OIS) for the input subsystem. It is possible to add new subsystems by extending the corresponding programming interfaces and choose the desired ones in run-time.

UGE is designed and implemented as a game engine. However, due to the three-tier architecture employed, it is possible to use many features independently. Thus, UGE can also be seen as a game framework to develop UA-Games – its core components can be the foundations for other engines.

# UGE Overview

## Introduction

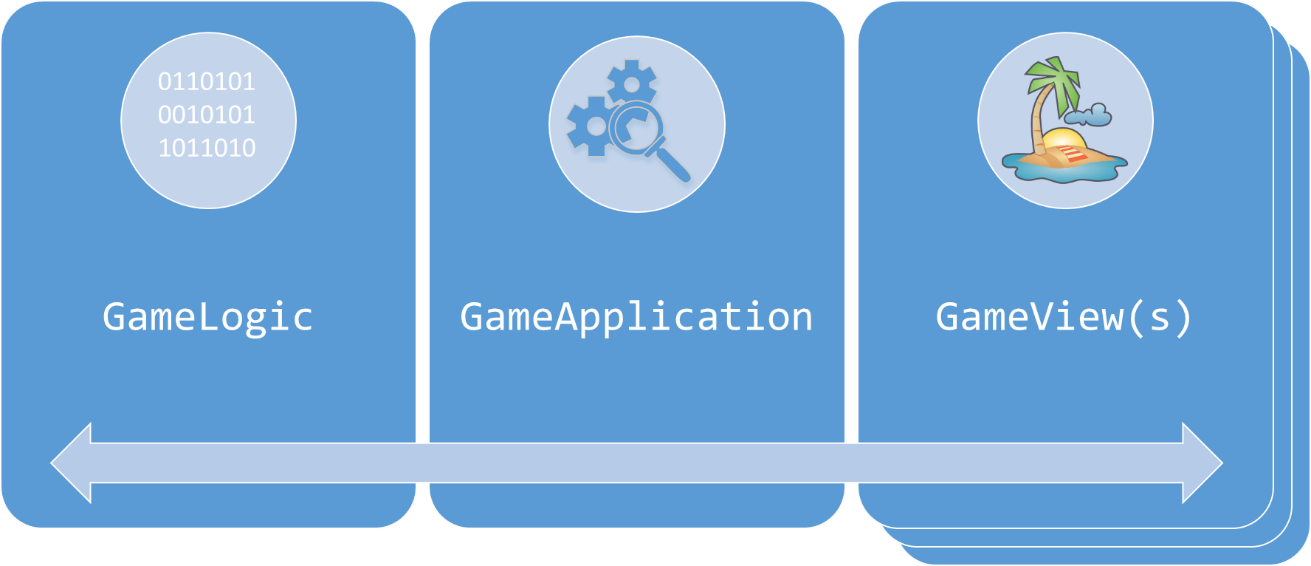
This section describes the structure and architecture of UGE. The subsections are organized by module abstraction level, in a descending order. Using a top-down approach, we start discussing the engine and its layers. This will provide insights on how the remaining subsections and their features fit in the engine.

A more detailed description for each abstraction level is available with more detail in its specific section – Section 3 for the Engine, Section 4 for the Core, Section 5 for Input-Output (IO) and Section 6 for Utilities.

## UGE Engine

The highest abstraction level structure of UGE is the engine itself. The engine groups all the codebase features in a few classes, abstracting and easing many of the complexities of the development in a cohesive way. The engine manages and orchestrate the execution of the game – including its events, physics, logic and presentation. To implement a game, developers extend the engines classes and implement the game logic and presentation.

As illustrated in **Figure 1**, the engine has three layers (GameApplication, GameLogic and GameView). As the layer names suggests, the engine uses a three-tier architecture to isolate the game logic (GameLogic) from the presentation (GameView). This way, it is possible to change the presentation data without changing the logic.



**Figure 1.** UGE’s layers: GameApplication, GameLogic and GameView.

The GameApplication provides an abstraction layer between the game and the user’s operating system. It creates and manages the game input and output subsystems, game resources, player profiles and both the GameLogic and GameView. The application also handles the game loop, updating and rendering the game and all subsystems in an appropriate order.

The GameLogic provides an abstraction the game implementation. This is where the game world is simulated with all its rules, logic and behaviors. The GameLogic handles the life cycle (creation, update and removal) of game entities during every tick of the game. It also has an IO free scene manager, allowing the use of hierarchical entities. A new game project has an empty, blank scene; when an entity is attached to the scene, its components will define its behaviors in the game world simulation. The GameLogic is defined by the subclass implementation and event handlers.

The GameLogic does not handle any input-output data for the game entities. This is the role of the GameView. The GameView abstracts the human interaction with the game. It defines how the user perceive and interact with the game – thus, controllers, cameras and output devices are created and chosen in the GameView.

It depends on all the other levels of the engine – most notably on the core functionality. The Engine is discussed with more details in Section 3.

## UGE IO

The GameView provides the IO specialization for the GameLogic. In a UA-Game, the interaction needs may vary greatly – from a graphical user interface with mouse and keyboard to an average user to an aural user interface with one-button controllers to a visually and motor impaired user. Therefore, it is very important to allow the use many different IO devices in a UA-Game. As stated in Section 2.2 (and detailed below in Section 2.4), the GameLogic has an IO free scene manager. Thus, it must be specialized to offer appropriate IO to the player.

This is the goal of the IO level. This level provides abstraction for Input and Output subsystem for the engine. By overriding the available interfaces (for instance, IAudio and IGraphics for audio and graphics output subsystems, respectively), it is possible to define new subsystems for the engine.

The engine loads the subsystems during run-time; thus, it is possible to choose and change a subsystem when the user starts playing the game. In combination with player profiles, adequate events and components, this allows the game to adapt itself to suit the interaction needs of the user.

The IO functionality depends on the utilities and core levels of the engine – and on whatever other chosen APIs or frameworks for IO specialization. It is discussed with more details in Section 5.

## UGE Core

As stated in Section 1.4, although UGE is envisioned as an engine, it is also possible to use it as a framework. This is possible due to the UGE core, the level that implements most of the engine basic functionality. With the exception of the three-tier architecture mentioned in Section 2.2, all other design and architectural choices are implemented here.

The core consists of the following game features:

* Commands;
* Entities and components;
* Events;
* Physics;
* Player profiles;
* Resources;
* Scene;
* Scripting;
* Tasks.

Each of these features abstracts and manages the game behavior and specialization in constricted, defined ways. The features allow a high-level abstraction of the game implementation – as one can note, there is no references to physical-level interaction in this level. Thus, the definition of every game interaction uses one or more of the available features.

For instance, moving an entity requires the combination of components, events and physics. Let us consider a MoveEntity event. When the event is triggered, the GameLogic might issue a request to the physics subsystem to move the entity in the chosen direction with the chosen velocity and acceleration. The physics subsystem does not care (or even knows) what generated the event – it could be a user key press or speech command or even a non-playable character (NPC) request from the game artificial intelligence (AI) – it only updates the position and rotation data. Thus, the GameView is free to read the request from the most appropriate input device and just dispatch the MoveEntity event when necessary.

The core functionality only depends on the utilities level of the engine. It is discussed with more details in Section 4.

## UGE Utilities

Finally, the last level provide framework and platform-independent utility functions. The utilities level is a general use toolbox, offering abstraction for commonly needed functionalities such as types, debug macros and logging, file handling, string manipulation, mathematics and system information.

The Utilities is discussed with more details in Section 6.

# UGE Engine

## Introduction

Section 2.2 presented an overview of the engine architecture and its layers: GameApplication, GameLogic and GameView. This section presents the UGE engine layers in-depth, discussing its usage and functionality. It also outlines the most relevant methods from each layer and important implementation details.

## Game Application Layer

As stated in Section 2.2, the GameApplication provides an abstraction layer between the game and the user’s operating system. The following subsections discusses some aspects with the layer with more depth.

### Game Application Functionality

The GameApplication class provides the basic functionality that enables a game to run. It brings together all the core functionality into a single class. For convenience, it also manages the both the GameLogic and GameView. Thus, it manages the lifecycle of the GameLogic, the GameView(s), of game resources and events and of the output systems during run-time – the game loop itself.

UGE provides two different classes for creating a game application: the GameApplication itself and the BaseGameApplication. The GameApplication is the lowest level class for managing a game, offering the minimal functionality required and outlining the required implementations with pure virtual functions. This eases the development of a new application case the developers find the default one unsuitable for their needs.

In the general case, however, UGE games implements the BaseGameApplication. It provides a high level framework for developing games, offering support for resource management, output systems, game views and player profiles out of the box. The BaseGameApplication defines a default game loop, creating, updating and destroying all the core functionality in a defined and appropriate way. It provides several helper methods to ease the creation of the game – all with very simple implementation.

Thus, the default settings of the BaseGameApplication allows the creation of a game only by defining the GameLogic and the desired GameView(s).

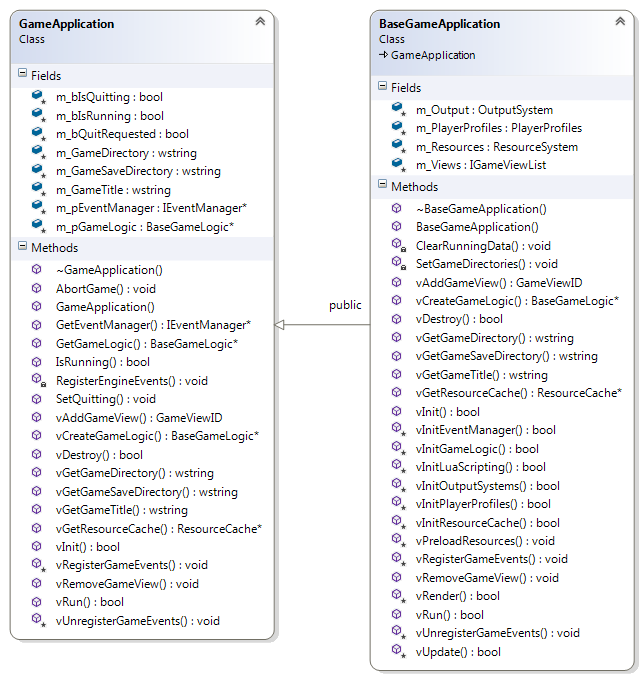
#### Game Application Input-Output Subsystems

The Game Application layer manages the IO subsystems. For convenience, it can use some high-level subsystems’ managers: the OutputSystem and the ResourceSystem. The OutputSystem manages the run-time lifecycle of the output subsystems after their initialization, handling graphical (IGraphics subsystem), aural (IAudio subsystem) and other stimuli for the output of the game (haptic, for instance). The output functionalities are further discussed in Section 5.3. As the name implies, these subsystems are output-only and, thus, do not interfere with the game logic – only convey its data to the user.

The ResourceSystem is the OutputSystem’s equivalent to manage game resources, such as assets. It offers a higher-level API to the core’s level Resource and ResourceCache. These features are discussed in Section 4.7.

### Game Application Architecture

The architecture of the UGE Game Application layer is illustrated in **Figure 2**. It combines all the core functionality with the engine together with the high-level GameLogic and GameView layers. By analyzing their data members, it is possible to see the GameApplication depends on the Game Logic layer and on the Event Manager. On the other hand, being higher-level, the GameApplication also depends on the Game View layer and on the output and resource subsystems. It also provides out of the box support for player profiles.



**Figure 2.** The Game Application Layer.

In **Figure 2**, all methods preceded by the letter ‘v’ are virtual. Thus, especially for the BaseGameApplication, it is possible to see how the methods guide the setup of the application: it is possible to override the methods to create the implementation for the game.

### Game Application Run-Time

The GameApplication is a straightforward class, leaving most of the implementation to the subclass. Its initialization is defined in the vInit() method and its de-initialization, in the vDestroy(). As the class only have basic data types (primitive types and pointers), both methods are simple – just setting the data to the default values.

The most important method is vRun(). It is a pure virtual method, which should be overridden in the subclass. This method should implement the game loop, calling the adequate Update() functions of the core level.

The BaseGameApplication subclass offers a default implementation for the GameApplication. Most of its complexity is hidden in the View layer GameView, IO level OutputSystem and in the core level ResourceSystem and PlayerProfile. As the implementation for all these subsystems may vary according to the chose GameView, the BaseGameApplication offers initialization methods for them. These methods are called in the appropriate time and order, allowing the game to be tailored accordingly the profile definition.

As vInit() method shows a possible sequence for the initialization, its execution is outlined in **Listing 1** below.

**Listing 1.** The BaseGameApplication’s initialization.

|  |
| --- |
| bool BaseGameApplication::vInit()  {  // Initializes the super class data.  if (!GameApplication::vInit())  {  return false;  }  ClearRunningData();  // Loads the player profiles.  if (!vInitPlayerProfiles())  {  return false;  }  SetGameDirectories();  // Initializes the Event Manager, used for dispatching events  // for the engine and the game.  if (!vInitEventManager())  {  return false;  }  // Initializes the resource cache – a structure which hold the  // assets and configuration files which will be used by the game.  if (!vInitResourceCache())  {  return false;  }  // Initializes the output systems  // (for instance, Graphics and Audio).  if (!vInitOutputSystems())  {  return false;  }  // Initializes the game logic (the game implementation itself).  if (!vInitGameLogic())  {  return false;  }  // Initializes the scripting bindings. The default one is Lua.  if (!vInitLuaScripting())  {  return false;  }  // Register events required by the game logic.  vRegisterGameEvents();  // Preload engine and game resources, such as assets  // and initial scripts.  vPreloadResources();  m\_bIsRunning = true;  return true;  } |

In **Listing 1**, it is possible to observe how early the player profiles are loaded for the BaseGameApplication. This provides the required data for the resource and output subsystems to tailor the game presentation and interactions to the user’s interaction needs.

The BaseGameApplication also implements the GameApplication’s method is vRun(), outlined in **Listing 2**.

**Listing 2.** The BaseGameApplication’s game loop.

|  |
| --- |
| bool BaseGameApplication::vRun()  {  // Main game loop.  Time::TimePoint startTime = Time::GetTime();  while (true)  {  Time::TimePoint currentTime = Time::GetTime();  unsigned long deltaNanoseconds =  Time::GetDeltaAsNanoseconds(currentTime,  startTime);  startTime = currentTime;  if (!vUpdate(currentTime, deltaNanoseconds))  {  // The user requested to quit the game.  return true;  }  if (!vRender(deltaNanoseconds))  {  // Some error occurred whilst rendering the game.  return false;  }  }  // Should never reach this.  return false;  } |

**Listing 2** shows a basic game loop. It fetches the current time, calculates the time elapsed since the last update, update the GameLogic and game subsystems (in vUpdate()) and render the game to the user (in vRender()). The Event subsystem is one of the update targets vUpdate() – thus, it is during this call that the previously triggered events are processed and dispatched for the registered listeners.

### Remarkable Methods

The most important methods for the Game Application layer classes (vInit(), vDestroy() and vRun()) were described in Section 3.2.

Other important methods are vAddGameView() and vRemoveGameView(), used to attach and detach a GameView to the GameApplication, respectively, and vRegisterGameEvents() and vUnregisterGameEvents(), which allows registering and unregistering game events to the Event Manager. As it will be discussed in Section 4.3, events can play a very important role in the implementation of UA-Games.

For the BaseGameApplication, all the initialization methods are also important. They offer a simple sequence for initializing the application, setting up the player profiles (vInitPlayerProfiles()), the resources (vInitResourceCache()), the event manager (vInitEventManager()), the game logic (vCreateGameLogic() and vInitGameLogic()) and the output subsystems (vInitOutputSystems()). These methods can be implemented by a subclass to initialize the respective features – they are usually a few lines of code long. Only vCreateGameLogic() and vInitOutputSystems() do not offer a default implementation. **Listing 3** illustrates an implementation of vInitOutputSystems().

**Listing 3.** A sample implementation of the BaseGameApplication’s method vInitOutputSystems().

|  |
| --- |
| virtual bool MyGameApplication::vInitOutputSystems()  {  // Graphics  uge::IGraphicsSharedPointer pGraphics(  LIB\_NEW uge::OgreGraphics(vGetGameTitle(),  m\_CurrentPlayerProfile.GetGraphicalPreferences()));  // Audio  const int TOTAL\_BUFFERS = 32;  uge::IAudioSharedPointer pAudio(LIB\_NEW  uge::OpenALSoftAudio(TOTAL\_BUFFERS));  return m\_Output.Init(pGraphics,  pAudio);  } |

In **Listing 3**, the MyGameApplication created two output subsystems: one for graphics and another for audio. A more complex example could load the desired systems from the player profile preferences.

## Game Logic Layer

As stated in Section 2.2, the GameLogic layer provides an abstraction the game implementation and contains the simulation of the game world with all its rules, logic and behaviors. The following subsections discusses some aspects with the layer with more depth.

### Game Logic Functionality

The GameLogic layer handles the run-time lifetime of the game. It orchestrates everything that exists in and interacts with the game world, managing the entities and game commands, events, physics and tasks.

This layer self-sufficient, i.e., it does not depend on any other layer of the engine; it depends only on UGE’s core features. Thus, it is possible to define the logic of a game without mentioning any input or output interactions – the game world can run and simulate itself. This gives great flexibility for polymorphic physical-level specialization: conveying the game information to the user is like snapping a picture of the current game state and describing it to the user. In the other hand, sending a command to the game just require dispatching an event when the users performs an action.

The GameLogic groups many of the core functionalities into a single structure. It uses entities and components; the Physics subsystem; the Scene Manager; the Task Manager and the Player Profiles. As the game logic complexity can vary depending on the game – from very simple to very complex –, UGE provides flexible and extensible Game States.

Each of these features are briefly discussed in the subsections below and detailed with more depth in Section 4.

#### Game Entities (Actors)

In a UGE game, everything that participates in the game world is an entity. Characters, vehicles, objects, scenery, projectiles… Everything is an entity. In some games, entities are also known as game objects or actors.

The engine uses a flexible entity approach: an entity-component approach [1,2,6,11,12,14,15,16]. An entity-component approach relies into smaller, self-constricted components to decouple the data and behaviors of an entity from its implementation [6].

In UGE, the entities are very simple, consisting of an identifier, a archetype name and a collection of components. This approach is the same described by McShaffry and Graham in GCC4’s actor’s model [12] – for consistency with the code base, entities will be referred as actors from this point on.

UGE’s components are data-only. The components hold specialized data members, which are processed in the relevant subsystems (or by the GameLogic itself). For instance, the TransformableComponent stores the position, orientation and scale of an actor. These members are relevant, for instance, for the Physics and the Scene subsystems. The Physics subsystem can change its values, translating it around the world; the Scene subsystem places it in the world, allowing other subsystems to render them when it is appropriate.

A comprehensive description of actors and components is available in Section 4.2.

#### Game Commands and Events

As the components are specialized and self-constrict, the GameLogic does need to access any output component. However, it is also necessary to make it independent from user input. This is the goal of game commands, issued with the aid of the EventManager.

UGE does not uses the user input directly into the game logic. Instead, it uses game commands in the form of events: the input provided by the user is translated to a high-level game command, in the form of an event. For instance, in a race game, instead of accelerating when the user presses a button, the game view dispatches an event (let us say, CarAccelerated) which is later processed by the GameLogic.

This improves the engine flexibility, reduces coupling between classes, and offers the developers several benefits. For instance, as the GameLogic receive generic game commands, it does not distinguish between a player input and a NPC input. On one hand, this simplifies the creation of AI for the game. On the other hand, as the command does not depends on the input device, it is possible to use different input devices and mappings to issue commands to the game – which goes along with player profiles to provide a customizable and tailor-able game.

An overview of converting game input to game commands it available at Section 3.4.1.1. Events are further discussed in Section 4.3, while game commands are discussed in Section 4.4.

#### Game Physics

Either the GameLogic can use existing frameworks for physics or the game can define its own. UGE provides support for no physics and for the Bullet Physics[[8]](#footnote-8) engine out of the box. It also is possible to extend the IPhysics subsystem interface and add support for other physics engines or frameworks.

The Bullet Physics integration provides collision detection and hard body physics, easing the development of games requiring advanced physics simulation.

The Physics subsystem is discussed in Section 4.5.

#### Game Scene

UGE allows the creation of complex hierarchical game scenes with an IO-free Scene manager. The Scene is organized as a tree and manages the relationships between different SceneNodes attached either to the root or to another SceneNode.

The Scene has two purposes: easing the implementation of the GameLogic when it is easier or more convenient to work with relative coordinates and enabling the GameApplication to use different subsystems for presenting the game.

The first purpose explores the hierarchical structure of the game scene. When a SceneNode is transformed, all its children SceneNodes are also transformed. This is useful for defining complex actors with various parts (for instance, animated characters), for creating follow-up actors and cameras.

For an UA-Game, however, the second approach is more interesting. As the Scene contains the world transforms of all actors, output subsystems can fetch this data and combine it to an output component to render the scene. For instance, an IGraphics subsystem can use its correspondent IDrawableComponent to draw the actor’s model to the users screen. An IAudio subsystem can fetch this same data and, alongside with a IAudibleComponent, play a positional sound corresponding to the actor and its localization.

This provides a flexible mechanism to define and convey the game information to the user.

The Game Scene subsystem is further discussed in Section 4.8. Before this section, a few additional features are presented in Section 3.4.1.1.

#### Player Profiles

UA-Games require different customizations to deliver an accessible gaming experience for users. Although their importance and use is clearer in the GameView layer (and, thus, they are discussed in Section 3.4.1.4 below), they also enables users and developers to tweak the GameLogic gameplay.

UGE allows the player profile to override the components settings. This feature has two benefits: it makes it easier for defining different difficult levels for a game and, principally, it allows tweaking the gameplay for a specific profile.

The former allow the designers to balance and improve the gameplay for different interaction abilities. This is important due to difference in the human perception: the way the human senses perceive a stimuli is also different. The required time to a user hear and understand an aural stimuli is different usually greatly differs from seeing.

Thus, the player profiles permits overriding the default components settings to improve the gameplay for different senses. It is possible, for instance, to change the scale or the acceleration and speed of the actors to adapt the game for different disabilities.

#### Game States

It is possible to divide the GameLogic into different states. UGE approaches to game states employs a Factory pattern [3], which allows the developers to create as many game states as they wish for the game.

Each GameState in UGE is a subclass of the IGameState interface. **Figure 3** outlines this interface. It has five required methods: vInit(), vDestroy(), vOnUpdate(), vGetName() and vTailorToProfile(). The first two methods are used, respectively, for initialization and de-initialization of the game state. The method vUpdate() is called for the current state on every update of the GameLogic, allowing the GameState to update itself and its members attributes. The method vTailorToProfile() allows the game state to tailor its logic to the current player profile – as mentioned in Section 3.3.1.5. Finally, vGetName() is used for the GameStateFactory – it defines a key to the selection of the state. Thus, the only requirement for creating a new game state is giving it an unique name.

Any state changes should be defined in the vOnUpdate() method. This method should request the change to the IGameLogic object, supplying the new game state name to its vChangeGameState() method.



**Figure 3.** Game states.

UGE offers a default GameState implementation in the BaseGameState class. It offers basic functionality, with a pointer to the IGameLogic object and a lifetime counter. The three default game states derive from this class: Uninitialized, Initializing and Running. The states purpose matches their names; it is important to note, however, that Uninitialized is an internal state used only when the engine is loading. **Listing 4** presents a sample game state.

**Listing 4.** A sample game state.

|  |
| --- |
| class MyGameState : public uge::GameState::BaseGameState  {  public:  /// The name of the state.  static const char\* g\_Name;  MyGameState()  {  // Initialize data members.  }  virtual ~MyGameState()  {  }  virtual bool vInit(uge::BaseGameLogic\* pGameLogic) override  {  // Call the super class first for default init.  uge::GameState::BaseGameState::vInit(pGameLogic);  // Perform the initialization.  return true;  }  virtual bool vTailorToProfile(const std::string&  xmlResourceFilename) override  {  // Load the profile and override gameplay data,  // if needed.  return true;  }  virtual bool vDestroy() override  {  // De-initialize this class first.  // Call the super class last for default de-initialization.  uge::GameState::BaseGameState::vDestroy();  return true;  }  virtual bool vOnUpdate(unsigned long timeElapsed) override  {  // Update the game logic for this state.  // To change the game state, call  //m\_pGameLogic->vChangeGameState("DesiredStateName");  // The change will occurr in the next game tick.  return true;  }  virtual const std::string vGetName() const override  {  return g\_Name;  }  };  const char\* MyGameState::g\_Name = "MyGameState"; |

A GameStateFactory creates the GameStates for the game. Thus, every UGE game must define a subclass from the GameStateFactory and the desired IGameStates. **Figure 4** presents a class diagram for the GameStateFactory.



**Figure 4.** The Game State Factory.

As **Figure 4** suggests, the base GameStateFactory is also simple. All a subclass has to do is to override the vInitFactory() method, as exemplified in **Listing 5** for nine different states (Initializing, SplashScreen, Paused, HiScores, MainMenu, NewGame, Running, GameOver and Exiting). These states definition is similar to the one in **Listing 4**. For convenience, the names of these states were stored in a static g\_Name class attribute.

**Listing 5.** A sample implementation of the method vInitFactory().

|  |
| --- |
| void MyGameStateFactory::vInitFactory()  {  m\_StateFactory.Register<MyGameStates::Initializing>  (MyGameStates::Initializing::g\_Name);  m\_StateFactory.Register<MyGameStates::SplashScreen>  (MyGameStates::SplashScreen::g\_Name);  m\_StateFactory.Register<MyGameStates::Paused>  (MyGameStates::Paused::g\_Name);  m\_StateFactory.Register<MyGameStates::HiScores>  (MyGameStates::HiScores::g\_Name);  m\_StateFactory.Register<MyGameStates::MainMenu>  (MyGameStates::MainMenu::g\_Name);  m\_StateFactory.Register<MyGameStates::NewGame>  (MyGameStates::NewGame::g\_Name);  m\_StateFactory.Register<MyGameStates::Running>  (MyGameStates::Running::g\_Name);  m\_StateFactory.Register<MyGameStates::GameOver>  (MyGameStates::GameOver::g\_Name);  m\_StateFactory.Register<MyGameStates::Exiting>  (MyGameStates::Exiting::g\_Name);  } |

The method in **Listing 5** might seem a little convoluted, due to templates. However, creating the factory is a straightforward process: the factory only needs the class (template parameter) and its name (method parameter). When it is necessary to create a new state, the GameLogic queries for the state name and the game logic instantiates and returns the desired state.

### Game Logic Architecture

The architecture of the UGE Game Logic layer is illustrated in **Figure 5**. The layer provides an abstract interface (IGameLogic) and a default implementation (BaseGameLogic). As it was the case **Figure 2**, all methods preceded by the letter ‘v’ are also virtual.

The BaseGameLogic supports and integrates all the UGE core features: developers can use it out of the box to start creating their UA-Game. This class implements all the required IGameLogic’s methods and provides auxiliary higher-level methods to manipulate the game data. It also provides many methods to ease the subclassing process – such as the vCreateGameActorFactory(), vCreateGameStateFactory() and vCreatePhysics() methods.

A game only requires inheriting the BaseGameLogic, defining GameStateFactory and some GameStates class to implement the game logic. The BaseGameLogic default implementation manages actors, physics and the game scene; it also manages the current game state and player profiles, initializing and destructing the game states in a proper way.



**Figure 5.** The Game Logic Layer.

Thus, if desired, it is possible to start prototyping a game only by override the Running game state and implementing its game logic.

### Game Logic Run-Time

As the IGameLogic is an abstract interface, this section will outline the run-time of the BaseGameLogic. New implementation of the IGameLogic might want to use it as a reference implementation. Regardless of implementation, the IGameLogic is updated by the GameApplication during the game loop (refer to **Listing 2** in Section 3.2.3).

The BaseGameLogic lifecycle starts at the vInit() method. This method registers, in this order: game events and event delegates, the actor factory, the game physics, the task and scene managers and the initial game state (Initializing). This is outlined in **Listing 6**. This listing reveals the reason the Initializing and Running states are mandatory: their names are the default ones for the BaseGameLogic’s initialization.

**Listing 6.** The default initialization of the BaseGameLogic.

|  |
| --- |
| // Must be called by child.  bool BaseGameLogic::vInit()  {  RegisterEvents();  RegisterDelegates();  m\_LifeTime = 0u;  m\_LastActorID = Actor::NULL\_ACTOR\_ID;  m\_pActorFactory = vCreateActorFactory();  m\_pPhysics.reset(vCreatePhysics());  m\_pTaskManager = LIB\_NEW TaskManager;  if (!m\_SceneManager.Init())  {  return false;  }  m\_pGameStateFactory = vCreateGameStateFactory();  m\_pGameStateFactory->Init();  assert(m\_pGameStateFactory != nullptr &&  "Invalid game state factory!");    m\_pNextGameState = nullptr;  m\_pGameState = m\_pGameStateFactory->CreateGameState(  GameState::Initializing::g\_Name);  m\_pGameState->vInit(this);  m\_PlayerProfileFileName = "";  m\_pGameState->vTailorToProfile(m\_PlayerProfileFileName);  return true;  } |

The BaseGameLogic updates in the call to vOnUpdate(). This method updates all the game tasks, actors and the game scene logic and lets the current game state update itself. Thus, not changes happens to the game views – as stated in Section 2.2, the game logic in UGE is IO free. It handles state changes as well. **Listing 7** specifies the update process.

**Listing 7.** Updating the BaseGameLogic.

|  |
| --- |
| void BaseGameLogic::vOnUpdate(unsigned long currentTime,  unsigned long timeElapsed)  {  // Convert from nano to milliseconds.  unsigned long timeElapsedMs = timeElapsed \* 1000;  m\_LifeTime += timeElapsed;  m\_pGameState->vOnUpdate(timeElapsed);  if (m\_pNextGameState != nullptr)  {  vChangeState();  }  // Update the scene.  if (!m\_SceneManager.Update(timeElapsed))  {  assert(0 && "Error updating the game scene!");  }  // Update all actors.  for (auto& actorIt : m\_Actors)  {  actorIt.second->Update(timeElapsed);  }  } |

The vOnUpdate() method is still high-level, delegating the update process to its data members – each of them are detailed in Section 4. Due to UGE’s game state approach, most of the game logic’s complexity and functionality are hidden in and performed by the current game state.

It is important to note the physics and tasks are not updated directly in **Listing 7**. As some game states might not require physics (for instance, a Pause game state), each state may handle the physics update as needed. The same reasoning applies to game tasks. This way, it is up to the state implementation to decide whether it will update some game subsystems or not.

For instance, **Listing 8** describes how the Running game state updates these subsystems.

**Listing 8.** Updating the physics subsystem and task manager.

|  |
| --- |
| bool Running::vOnUpdate(unsigned long timeElapsed)  {  bool bSuccess = BaseGameState::vOnUpdate(timeElapsed);  unsigned long timeElapsedMs = timeElapsed \* 1000;  TaskManager\* pTaskManager = m\_pGameLogic->GetTaskManager();  pTaskManager->UpdateTasks(timeElapsedMs);  IPhysicsSharedPointer pPhysics = m\_pGameLogic->vGetPhysics();  if (pPhysics)  {  pPhysics->vUpdate(timeElapsed);  pPhysics->vSyncVisibleScene();  }  return bSuccess;  } |

The method vChangeState() (**Listing 9**) is internal and called when a game state change is requested via vChangeGameState(). This method destroy the current state and initializes the new one, making it the current state.

**Listing 9**. Changing game states.

|  |
| --- |
| void BaseGameLogic::vChangeState()  {  m\_pGameState->vDestroy();  m\_pGameState = m\_pNextGameState;  m\_pGameState->vInit(this);  m\_pGameState->vTailorToProfile(m\_PlayerProfileFileName);  m\_pNextGameState = nullptr;  } |

The BaseGameLogic lifecycle ends in vDestroy() method ().

**Listing 10.** Finishing the game: the default de-initialization of the BaseGameLogic.

|  |
| --- |
| // Must be called by child.  bool BaseGameLogic::vDestroy()  {  if (!m\_SceneManager.Destroy())  {  return false;  }  SAFE\_DELETE(m\_pActorFactory);  SAFE\_DELETE(m\_pGameStateFactory);  SAFE\_DELETE(m\_pTaskManager);  m\_pGameState->vDestroy();  m\_pGameState = nullptr;  m\_pNextGameState = nullptr;  for (auto& actorIt : m\_Actors)  {  actorIt.second->Destroy();  }  m\_Actors.clear();  // If debug drawer is enabled, physics must be deleted  // before the render systems.  if (!m\_pPhysics->vDestroy())  {  return false;  }  m\_pPhysics.reset();  UnregisterDelegates();  UnregisterEvents();  return true;  } |

This method de-initializes all the game logic data in an adequate order. It is called when the game is quitting as, in this moment, nothing else depends on it.

### Remarkable Methods

Section 3.3.3 described the run-time of the BaseGameLogic – with emphasis on vInit(), vDestroy() and vOnUpdate(). This section details the remaining methods of the IGameLogic interface not described previously.

The remaining methods regard game states, assets, physics and actors. As briefly stated in Section 3.3.3, vChangeGameState() requests the game logic to change its internal state. The requested state must had been defined in the game’s GameStateFactory. This method is public by default and can be called by any game state, whenever it is necessary to change the state. It is possible to query and obtain the current game state at any time using the method vGetGameState().

The method vLoadGame() allows the game logic to loads the assets for a game level. This is convenient, for instance, when one desires to load the data in one state and proceed to another when everything is loaded – this way, it is possible to share data between different game states.

It is also possible to create, update and remove game actors at any time (actors are detailed in Section 4.2). For this goal, the IGameLogic interface provides the methods vCreateActor(), vGetActor(), vDestroyActor(), vModifyActor() and vMoveActor(). The first three methods are self-explanatory and used to manage the actors. Implementations are free to store the actors in the most appropriate data structure for the game play. The BaseGameLogic uses a map; other games might find other data structures more suitable or explore patterns (such as Flyweight [3]) to improve the performance.

The method vModifyActor() is very interesting for UA-Games purposes. This method allows changing the actor’s components, attaching new components to it or changing the data of existing ones. Combined with player profiles, this allows adding different components to setup the game presentation (detailed in Sections 3.4, 4.2 and 5.3) or to tweak the gameplay from common components (for instance, allowing customizations of the speed of an actor in different profiles).

As the data from an actor originates from its components, vMoveActor() provides a convenient and easy way to change the transform stored in the TransformableComponent.

Finally, vGetPhysics() returns a pointer to the game’s physics subsystem. This is an alternative to using a global variable or a singleton and allows using all the physics subsystem functionalities.

## Game View Layer

As stated in Section 2.2, the GameLogic does not handle any input-output data for the game entities. This is the role of the GameView. The GameView abstracts the human interaction with the game, conveying the game data to output information and converting input to game commands. It defines how the user perceive and interact with the game – thus, controllers, cameras and output devices are created and chosen in the GameView.

This section presents the Game View layer, discussing its functionality, architecture and run-time lifecycle.

### Game View Functionality

As state previously, the GameView manages the IO for a UGE game. Thus, this layer is responsible for conveying the game information to the user and for sending the user’s commands to the game. As stated in Section 3.2.1.1, UGE’s provides different output subsystems, such as the IGraphics and IAudio subsystems. Supported input devices include keyboards and mice. If the developers want different IO subsystems, it is possible to extend the provided interfaces, as it will be discussed in Section 5.

Either way, the GameView uses the available IO subsystems to present the game to the user. A game can have as many GameViews as the developers wish. As this time, only one can be active at any given time.

As the GameLogic does not depends on the GameView, it is possible to choose and change the active GameView in run-time – both for input and for output. Thus, users can choose the available view that better suit their interaction needs. The next subsections describes how UGE eases presenting and interacting with the game in different way using GameViews – and with it is even possible for each of them to have totally different IO modalities, without modifying the GameLogic.

#### Game Controller and Game Commands

In a UA-Game, the required input devices may considerably depending on the users abilities. As the devices may range from joysticks and keyboards to eye-trackers and one-button devices, it is necessary to provide a flexible approach to translate the action performed in the input device to a game command. A GameController stores the InputDevices the game will use; each InputDevice reads the received data and converts it to game commands.

UGE abstracts the game commands in three levels of abstraction: a physical-level action performed (a raw action), a low-level game command (an enumeration value) and a high-level game command (an event). With this approach, it is possible to offer to the users a fully customizable input-mapper, allowing the users to map the buttons or keys they desired to perform the existing game commands.

For instance, let us consider a platform game with the following set of low-level game commands: jump, walk and run. If the user wants to jump, he/she presses the Space key. The GameController receives this key press and verifies it is mapped to the Jump command. Then it proceeds to send the ActorJumped event. This is illustrated in **Figure 6**.



**Figure 6.** Game command mapping.

**Figure 6** suggests that it is possible to swap the Space key to any other key – or even for a different input device. As long as the GameController sends the Jump command, the actor will eventually jump. UGE uses a data-driven approach for input mapping, allowing the users to customize the input devices and bindings they want to use.

Up to this point, the low-level command might seen enough – and it really is. However, converting it to a high-level command via an event has several benefits, including:

* Offering additional information for the GameLogic (for instance, the running direction or speed);
* Making it easier to provide input feedback when a command is performed, which is very useful for non-graphical GameViews;
* Allowing a common command interface to create human and AI game commands.

Game commands are further discussed in Section 4.4; input devices and input mapping are discussed in Section 5.2.

#### Game View Types

As hinted in Section 3.4.1.1, using events as a high-level game command allows the GameView to have different roles. A HumanGameView is available for interaction with human players; a RemoteView can be used for multiplayer games; an AIView can define the AI for NPC game actors, and so on.

To create a new type of GameView, developers need to derive the IGameView abstract interface. UGE provides a default HumanGameView implementation, which eases the definition of a game presentation for the user. Combined with the GameController, extending the HumanGameView provides an easy way to create the game’s user interface.

Both the IGameView and HumanGameView are discussed in Sections 3.4.2 and 3.4.3.

#### Game Scene Renderer

Section 3.3.1.4 stated that the GameLogic has its IO free GameScene. Without IO, there is no game – just a computer simulation. Thus, it is necessary to specialize the GameScene to present the game data and to collect user input – this is the role of the game controllers (presented in Section 3.4.1.1) and scene renderers.

The ISceneRenderer provides an abstract interface to present the scene data to the user. It is possible to define scene renderers for different modalities; UGE provides reference implementations for audio and graphics subsystems (IAudioSceneRenderer and IGraphicalSceneRenderer, respectively).

The scene renderer relies on cameras to position the observer in the game scene and scene nodes containing the game actors. Each SceneNode of the GameLogic has a SceneNodeRenderer correspondent for presenting its output data. A special SceneNodeRenderer is responsible for defining what shall be displayed to the user: this is the role of a camera, abstracted in the ICameraNode class.

This way, the active GameView presents the game to the user according to its scene renderers. Choosing different systems and scene renderers allows for completely different game presentations; thus, creating accessible user interfaces for different user’s abilities starts by defining the most adequate output stimuli and subsystems and implementing the ISceneRenderer interface.

UGE provides a high-level manager for rendering the scenes: the SceneRendererManager. This manager offers convenient methods for handling the run-time presentation. More information regarding it and scene rendering is available in Section 4.8.

#### Game Events

Due to differences in how the human senses perceive a stimuli, it is not always possible to define implement the ISceneRenderer abstract interface to create accessible output for a particular interaction ability. This might be enough, for instance, for graphical-only user interfaces; however, it usually is not for other types of senses.

In order to provide an accessible gaming experience, it is, thus, necessary to provide further information to the user. An interesting approach to achieve this is using game events to provide view feedback. With events, it is possible to offer additional information to the player during important moments in the game, with some interesting benefits:

* Decoupling the feedback from the game logic. It is possible to handle the events at any point of the implementation, allowing the presentation to convey more information without interfering with the game logic;
* Allowing the developers or users to enable or disable specific event handlers. This contributes to further tailoring the game to the user’s interaction needs. It is also possible to reuse existing events in different player profiles;
* Defining key points to conveying information to the player. This is especially useful when the designers want to specialize the game for different abilities: the events will provide references of what should be presented to the player. Furthermore, every new event can contribute to enhance future and even old specializations, contributing to continuous and iterative improvements;
* Making it easier to work with non-visual specialization, such as aural or tactile feedback. This is useful to providing information about game logic happenings, such as the sound of footsteps when an actor is walking or a vibration after a collision. Or even to provide input feedback for a successful game command.

Game events are further discussed in Section 4.3.

#### Player Profiles

Section 3.3.1.5 discussed the use of player profiles for the Game Logic layer. Their importance is, however, much clearer here in the Game View layer.

The player profile allows the developers (and even user themselves) to customize the game IO to suit specific interaction abilities. It is possible to define the input and output stimuli and desired subsystems; to provide a input mapping of the desired controller scheme; to choose which event specializations the game should use; and change other general settings (such as the desired language). For convenience, the game logic specialization are defined here as well.

**Listing 11** illustrates a player profile. The player profile is a XML file containing a resource for every setting. This file is parsed when the game is loading; this allows the game to tailor itself to the profile’s specifications.

**Listing 11.** A player profile.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfile name="Average User: Default"  resource="data/config/player\_profiles/average\_user/profile.xml">  <GeneralSettings>  <Preferences resource=  "data/config/player\_profiles/average\_user/preferences.xml"/>  </GeneralSettings>  <InputSettings>  <!--<Device type="keyboard"/>-->  <Mapping resource=  "data/config/player\_profiles/average\_user/input\_mapping.xml"/>  </InputSettings>  <OutputSettings>  <PrimaryOutput type="graphical"  resource=  "data/config/player\_profiles/average\_user/graphics.xml"  events=  "data/config/player\_profiles/average\_user/graphical\_events.xml"/>  <SecondaryOutput type="aural"  resource="data/config/player\_profiles/average\_user/audio.xml"  events=  "data/config/player\_profiles/average\_user/aural\_events.xml"/>  </OutputSettings>  <GameplaySettings>  <EntitySpecialization resource=  "data/config/player\_profiles/average\_user/entity/entities.xml"/>  </GameplaySettings>  </PlayerProfile> |

For instance, **Listing 12** offers an example for the PrimaryOutput XML file. It is possible to see it describes the desired settings for the graphical renderer, window settings and so on.

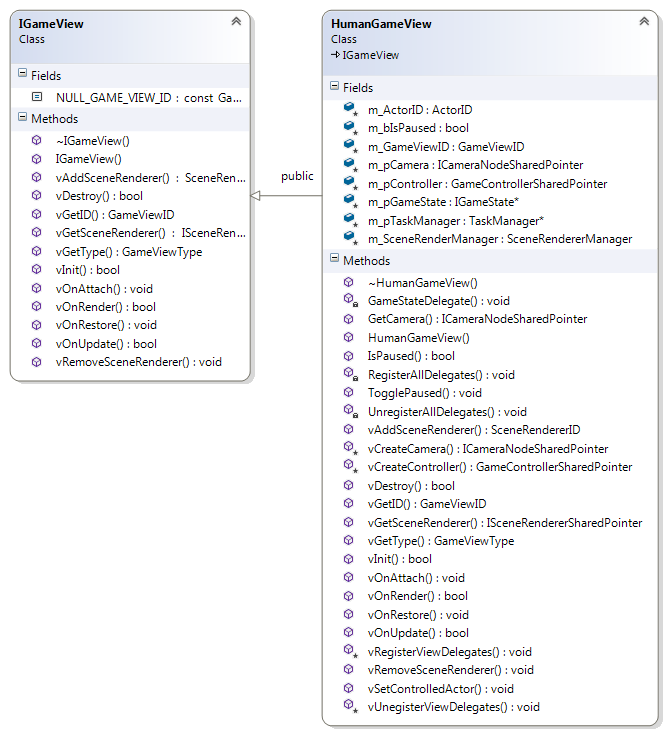
**Listing 12.** The specifications for the PrimaryOutput of **Listing 11**.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Graphics  resource="data/config/player\_profiles/average\_user/graphics.xml">  <Window width="1024" height="768"  pixel\_depth="32" vsync="false" fullscreen="false" />  <Renderer name="OpenGL Rendering Subsystem"/>  <Text>  <Font resource="font\_name" size="10" />  <Subtitles enabled="false" />  </Text>  </Graphics> |

The other files are similar and are discussed with more depth in Section 4.6.

### Game View Architecture

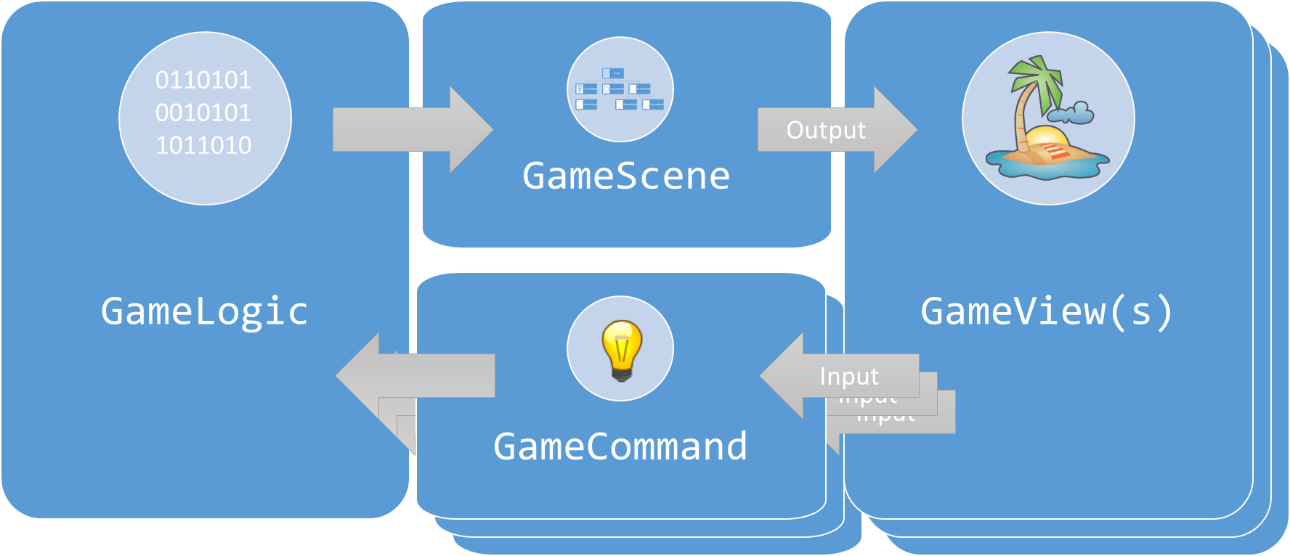
The architecture of the UGE Game View layer is illustrated in **Figure 7**. This figure presents the IGameView abstract interface, which provides an abstraction for any possible game view (as stated in Section 3.4.1.2, it is possible to create different types of game views, including views for non-human actors). A game can have as many IGameViews implementations as wanted; it is also possible to load a custom one from a player profile.



**Figure 7.** UGE Input and Output: the IGameView abstract interface and the HumanGameView.

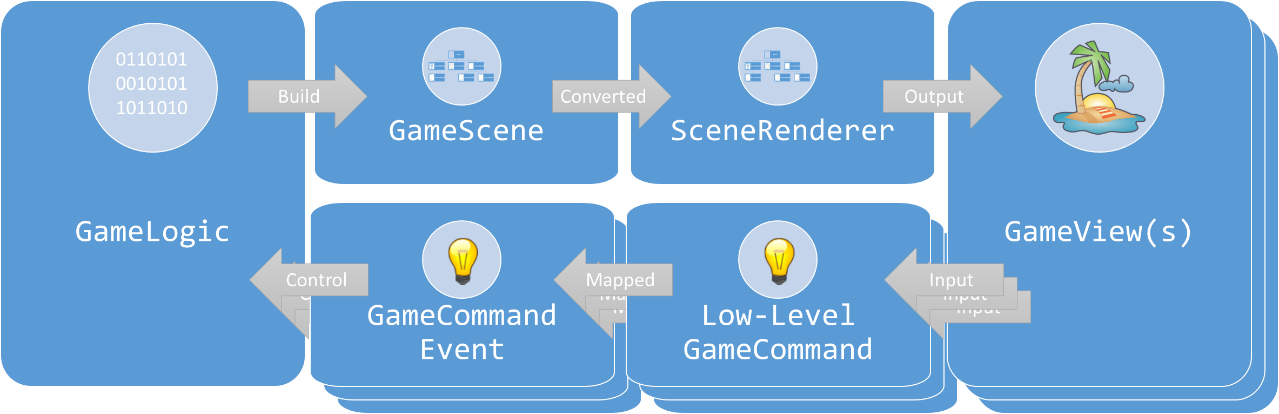
The IGameView defines the run-time lifecycle of a view, including its initialization and de-initialization, updating and rendering. The implementation of the IGameView are responsible for presenting the game to the user and fetching his/her game commands – or for controlling an NPC, presenting other human users in a network game, playing a replay... The way the information will be gathered and presented depends only on the implementation.

To bring everything together, the IGameView dataflow is illustrated in **Figure 8**. This figure sums UGE’s proposal: the GameLogic does not depends on specific IO and, thus, its IO can be specialized to suit the users’ abilities and interaction needs. The implementation of the IGameView abstract interface can capture the current GameScene and present its contents to the user using a SceneRenderer; it can also capture the input information and translate to GameCommands, which, in turn, will feed the GameLogic.



**Figure 8.** The IGameView dataflow.

A lower-level representation is illustrated in **Figure 9**. This figure subdivides the GameScene into in GameScene and SceneRenderer and the GameCommand into its low and high-level counterparts, as described in Sections 3.3.1 and 3.4.1.



**Figure 9.** A lower level IGameView dataflow.

UGE provides a default implementation of the IGameView interface for human users: the HumanGameView, also shown in **Figure 7**. The HumanGameView offers convenient methods to allow the user to interact with the game, including outputting the game data to the user using a Camera and SceneRenderers and gathering his/her commands using a GameController (as described in Section 3.4.1.1). It should be inherited by and can be overloaded by a game specific child class.

The following section describes how the IGameView works during run-time, taking the HumanGameView as a reference.

### Game View Run-Time

Similarly to the Section 3.3.3’s IGameLogic case, the IGameView is also an abstract interface; thus, this section will outline the run-time of the HumanGameView. New implementation of the IGameView might want to use it as a reference implementation. Regardless of implementation, the IGameView is updated after the IGameLogic by the GameApplication during the game loop (refer to **Listing 2** in Section 3.2.3).

The HumanGameView run-time lifecycle starts with a call to its vInit() method – outlined in **Listing 13**. This method initializes its data members, including the SceneRenderManager and the Camera (Sections 3.4.1.3 and 4.8) and the GameController (Section 3.4.1.1). The human view for a game should override this method and register the view’s event delegates, if any.

**Listing 13.** Initializing the HumanGameView.

|  |
| --- |
| bool HumanGameView::vInit(IScene\* const pScene)  {  m\_bIsPaused = false;  m\_pTaskManager = LIB\_NEW TaskManager;  RegisterAllDelegates();  if (!m\_SceneRenderManager.Init(pScene))  {  return false;  }  m\_pCamera = vCreateCamera();  assert(m\_pCamera != nullptr && "Invalid camera!");  m\_SceneRenderManager.AddCamera(m\_pCamera);  m\_pController = vCreateController();  assert(m\_pController != nullptr && "Invalid controller!");  m\_pController->vInit();  return true;  } |

It is interesting to note that the HumanGameView has its own TaskManager. This is convenient for handling UI specific tasks – for instance, updating health bars in a HUD (Head-Up Display).

The HumanGameView is supposed to be operated by a human player. Thus, it is necessary to know which actor the player controls. This is possible with the vOnAttach() method (**Listing 14**). With the ActorID, it is possible to query the actor’s data members, such as its position in the game world. On the other hand, the GameViewID allows using different IGameViews for different purposes.

**Listing 14.** Attaching a human controlled actor to the HumanGameView.

|  |
| --- |
| void HumanGameView::vOnAttach(GameViewID viewID, ActorID actorID)  {  m\_GameViewID = viewID;    vSetControlledActor(actorID, false);  } |

The vSetControlledActor() is specific to the HumanGameView and is discussed in Section 3.4.4.

With our without an attached actor, the vOnUpdate() method (**Listing 15**) is responsible for updating the HumanGameView. As the name suggests, this method does not render or presents the game yet; instead, it updates the IO subsystems. Thus, it allows the rendering subsystems to prepare for rendering and capture the user input. Any other state or logic operation regarding the game presentation should be updated by the subclass in this method.

**Listing 15.** Updating the HumanGameView.

|  |
| --- |
| bool HumanGameView::vOnUpdate(const unsigned long timeElapsed)  {  unsigned long timeElapsedMs = timeElapsed \* 1000u;  m\_pTaskManager->UpdateTasks(timeElapsed);  if (!m\_SceneRenderManager.Update(timeElapsed))  {  return false;  }  m\_pController->vUpdate(timeElapsed);  return true;  } |

After updating the internal members, the game is finally render by the vOnRender() method (**Listing 16**). The SceneRendererManager copes with rendering the game scene; the game subclass implementation can override this method to present further information, such as a HUD.

**Listing 16.** Rendering the HumanGameView.

|  |
| --- |
| bool HumanGameView::vOnRender(const unsigned long currentTime,  const unsigned timeElapsed)  {  if (!m\_SceneRenderManager.Render())  {  return false;  }  return true;  } |

When a HumanGameView should be destroyed, the vDestroy() method (**Listing 17**) is invoked. It performs the clean-up and frees the resources and memory used by the view.

**Listing 17.** De-initializing the HumanGameView.

|  |
| --- |
| bool HumanGameView::vDestroy()  {  UnregisterAllDelegates();  if (!m\_SceneRenderManager.Destroy())  {  return false;  }  SAFE\_DELETE(m\_pTaskManager);  m\_pController->vDestroy();  m\_pController.reset();  m\_pCamera.reset();  return true;  } |

It is important to note the Game Logic layer is not affect by creating or destroy a IGameView.

### Remarkable Methods

Section 3.4.3 outlined the most important methods of the IGameView abstract interface. However, it is possible to see in **Figure 7** there are more methods – notably the vAddSceneRenderer() and vRemoveSceneRenderer().

These methods register or unregister a SceneRenderer to present the game scene. Using the available SceneRenderManager, their implementation is trivial (as illustrated in **Listing 18**). Otherwise, the IGameView should register the SceneRenderers in an appropriate data structure.

**Listing 18.** Adding and removing SceneRenderers to the Scene.

|  |
| --- |
| SceneRendererID HumanGameView::vAddSceneRenderer(  ISceneRendererSharedPointer pRenderer)  {  return m\_SceneRenderManager.AddSceneRenderer(pRenderer);  }  void HumanGameView::vRemoveSceneRenderer(SceneRendererID rendererID)  {  m\_SceneRenderManager.RemoveSceneRenderer(rendererID);  } |

Finally, especially for game views without graphical output or first person cameras, it is convenient to set the camera position to the human controlled actor position after a call to vOnAttach(). **Listing 19** contains a sample implementation.

**Listing 19.** After attaching a human actor (**Listing 14**), it might be convenient to set the camera to its position.

|  |
| --- |
| void HumanGameView::vSetControlledActor(ActorID actorID,  bool bSetCameraTarget)  {  m\_ActorID = actorID;  if (m\_pController != nullptr)  {  m\_pController->SetControlledActorID(actorID);  }  if (bSetCameraTarget)  {  ISceneNodeSharedPointer pActorNode =  m\_SceneRenderManager.GetSceneNode(actorID);  m\_pCamera->vSetTarget(pActorNode);  }  } |

In this implementation, the camera assumed the position of the user’s controlled actor. This way, all output and information conveyed to the player will be relative to the actor’s position – which, in a first player camera, would be relative to the user.

For an audio-only HumanGameView, this offers a convenient and more intuitive way of conveying the information to the user [5].

# UGE Core

## Introduction

This section presents the core features of UGE. As stated in Section 2, the core of UGE provides features that abstract and manages the game behavior and specialization in constricted, defined ways. They help implementing the game in a higher level of abstraction and help keeping the Game Logic layer IO free.

The remaining of this section describe the UGE core features’ architecture, functionality and use in a game. It also discusses how the features aid in the development of a UA-Game.

It is recommended to read Sections 2 and/or 3 before reading this section, to have an overview of the engine organization and a general idea of how the features mix with each other.

## Entities (Actors) and Components

Game entities are characters, scenery or objects that interacts with each other and with game world. Thus, it is possible to consider a game entity as anything that has an exclusively non-aesthetical purpose in the game[[9]](#footnote-9).

In GCC4, McShaffry and Graham defines game entities as actors [12]. This definition is very accurate: in the same way actors play roles to compose, shape and set up a show, game actors shape the game world and create the game experience. Without actors, a play would be inanimate, dead. Without a game actor, a game would be empty, non-interactive.

The use of inheritance is a popular approach to define game actors. Albeit simpler and intuitive, for game accessibility purposes it constricts the flexibility for design, implementation and code-reuse – especially if an actor assumes a particular output representation [6]. Specialization is another drawback to inheritance: once a class for an actor type is defined, it cannot be changed without affecting all its subclasses. Alternatively, it could be even worse: different subclasses from unrelated classes might share a common functionality their parents do not – forcing a violation of the class hierarchy or duplicating code.

There is, however, a more flexible approach to designing entities: using entities and components. As stated in Section 3.3.1.1, an entity-component approach relies into smaller, self-constricted components to decouple the data and behaviors of an actor from its implementation [6].

Using this approach, components are attached to entities when needed. For instance, if an actor needs a world transform to store its position, orientation and scale, it needs to attach a TransformableComponent to its composition. If the actor should collide with the game world and/or other actors, it is a matter of attaching a CollidableComponent – for instance, a box shape component. And so on – as schematized in **Figure 10**. The components provide the necessary data members to the actor; an adequate subsystem can process the desired components to add its behavior to the actor during gameplay.



**Figure 10.** An actor with two components attached.

Thus, it is possible to attach and detach data members to actors when it is needed – and even during run-time. With an additional benefit for UA-Games: the actors do not depends on IO components to be processed by the GameLogic. There are not any representation assumed – to change how the actor will appear in an IGameView it is just necessary to change its output component.

For instance, let us consider the actor of **Figure 10** is an airplane. If the designers attach a DrawableComponent to the actor, a possible resulting IGameView could be the one drawn in **Figure 11**.



**Figure 11.** The actor from **Figure 10** with a DrawableComponent.

On the other hand, had the designers attached only an AudibleComponent, it would sound like in **Figure 12**. With an adequate IGameView and the aid of game events, the developers could make the game accessible to visually-impaired players, for instance.



**Figure 12.** The actor from **Figure 10** with an AudibleComponent.

What would happen if the actors added the two components to the actor? The IGameView would convey both graphical and aural information to the user, as shown in **Figure 13**.



**Figure 13.** The actor from **Figure 10** with an AudibleComponent and a DrawableComponent.

The previous examples show how flexible and useful an entity-component approach can be for a UA-Game. The chosen components alone can define which game world interactions are available to an actor – or how to present the actor to the user.

The following subsections describe UGE’s entity-component approach implementation more in-depth.

### Functionality

As stated in Section 3.3.1.1, UGE follows the actor’s model proposed by McShaffry and Graham for GCC4 [12] with some minor modifications. In UGE, the entities are very simple, consisting of an identifier, an archetype name and a collection of data-only components. The components hold specialized data members, which are processed in the relevant subsystems (or by the GameLogic itself).

For instance, in **Figure 13** the TransformableComponent stores the position, orientation and scale of an actor. These members are relevant, for instance, for the Physics and the Scene subsystems. The Physics subsystem can change its values, translating it around the world; the Scene subsystem places it in the world, allowing other subsystems to render them when it is appropriate.

On the other hand, CollidableComponent lets the Physics subsystem to detect whether the actor collided with another actor or not; and the DrawableComponent and AudibleComponent stores the art assets that the IGraphics and IAudio subsystems can use to render and present the actor to the user.

### Architecture

**Figure 14** illustrates the class diagram for the Actor class. It is possible to note the Actor is simple, having only a few data members – the most important being its ActorID, its archetype and a collection of components (a map, by default).



**Figure 14.** UGE’s entity: the Actor.

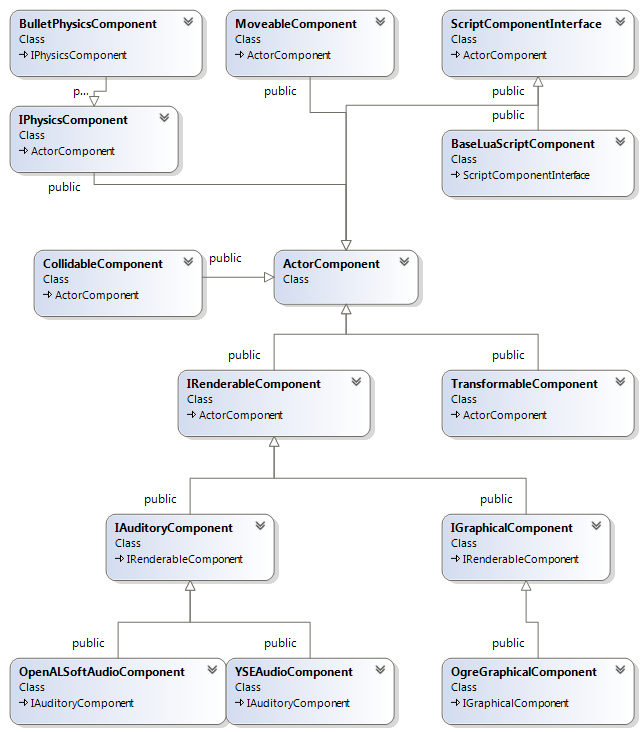
The complexity of the Actor derives from its attached components. It is possible to aggregate different components to an Actor to define its data members and behaviors. All the components derive from a base component[[10]](#footnote-10) (ActorComponent – illustrated in **Figure 15**). The ActorComponent class is also simple, adding only a pointer to the Actor that owns the component and a ComponentID (by default, the hashed component name).



**Figure 15.** The component base class.

When the developers wish to create a new component, they implement an ActorComponent subclass – many examples of out of the box UGE’s components are illustrated in **Figure 16**. The created subclass shall store all the data the component should have – for instance, some attributes to hold status information or the data or path to an art asset – and nothing else. Thus, creating a new component is a simple task: the heavy work will be performed by the subsystems that shall manipulate the data.

Another reason for the ease of component creation is to allow the components to be self-constricted. This way, a component – like a class – should have a single and well-defined responsibility. This contributes to code-reuse and enhance the flexibility of the code. If it is really needed (let us say for performance reasons), components could be merged with others.



**Figure 16.** Different types of components deriving from ActorComponent.

Another benefit of using data-only components is exploring data driven-architectures. This allows defining the Actor with an external data source (UGE uses a XML file) which is parsed during run-time to create the Actor with its ActorComponents. **Listing 20** presents an example of a XML file to a Ping-Pong paddle actor having various components, ranging from game-logic components, such as a TransformableComponent, to presentation components, such as the OgreGraphicsComponent.

**Listing 20.** A data-driven Actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Paddle-Player1" resource="paddle-player1.xml">  <TransformableComponent>  <Position x="-180.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) in radians -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="4.0f" y ="10.0f" z="20.0"/>  </TransformableComponent>  <MoveableComponent>  <Velocity vcs="100.0f" vdx="1.0f" vdy="-1.0f" vdz="0.0f"/>  <Acceleration ax="0.0f" ay="0.0f" az="0.0f"/>  </MoveableComponent>  <CollidableComponent>  <Shape type="Box">  <Dimension x="1.0f" y="1.0f" z="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Gold"/>  <Material type="Elastic"/>  </CollidableComponent>    <BulletPhysicsComponent>  <LinearFactor x="0.0f" y="0.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="15.0f"/>  </BulletPhysicsComponent>    <OgreGraphicalComponent>  <NodeName n="Paddle-Player1-Graphics"/>  <MeshFileName m="box.mesh"/>  <MaterialFileName n="box.material"/>  </OgreGraphicalComponent>    <OpenALSoftAudioComponent>  <NodeName n="Paddle-Player1-Audio"/>  <FileName n="data/audio/effects/blip.wav"/>  <Volume v="0.0f"/>  <InitialProgress p="0.0f"/>  <Loop l="true"/>  </OpenALSoftAudioComponent>    </Actor> |

It is possible to note the initial values for the variables are stated in the file. Thus, it is possible to update or tweak its value without recompiling the game. An ActorFactory (**Figure 17**) parses the XML file, calls the components registered initialization methods and returns an Actor ready for action.



**Figure 17.** The ActorFactory.

Using a data-driven architecture for actors have several other benefits, including easing the development of content creation tools and the definition of player profiles (c.f. Section 4.6).

### Run-Time

As stated in Section 4.2.2, to explore the data-driven potential data-only components, in UGE actors are created exclusively by an ActorFactory. This factory parses a resource file (similar to the one in **Listing 20**) passed as parameter to its CreateActor() method and returns a pointer to the newly created actor with all its components and initial data. The GameLogic provides the vCreateActor() method to ease the creation of game actors.

If the developers with to use the resource as an actor archetype, it is possible to pass an optional parameter containing the actor’s specific data to override the default data. For instance, it is possible to define a soldier archetype. Its resource can be a single actor – and created as such – or be used in a loop to create an army, in which every soldier has specific initial values. The overrides even allow attaching different components to the actors, which can have interesting game-logic effects. In the previous examples, soldiers could have different AI components, for instance.

To modify an actor, the ActorFactory provides the method ModifyActor(). This method serves the same purposes of the option override option in CreateActor(). However, it may be called at any time. By using ModifyActor() with Player Profiles, it is possible to customize the gameplay and to create tailored actors to suit different interaction abilities and capabilities (c.f. 4.6).

The ActorFactory is not a manager; therefore, it does not store the created actors. Thus, after the actor’s creation, it probably should be stored in game-logic variable – either its pointer for simple games or in a collection. The BaseGameLogic uses an ActorMap for this purpose, offering O(log(n)) access to the actor by an identifier – the ActorID. To fetch the Actor’s pointer, it is necessary to call the vGetActorMethod().

The Actor’s class method Destroy() is responsible for freeing the Actor’s data and components. The developers should use it whenever they want to delete an actor from the game. The GameLogic provides the method vDestroyActor() to ease the removal of actors.

### Example

This example shows how to create an Actor. First, it is necessary to define the XML resource for the Actor’s archetype. **Listing 21** shows the actor.xml file, describing an actor with two components: a TransformableComponent and a MoveableComponent.

**Listing 21.** A sample Actor resource.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Actor" resource="actor.xml">  <TransformableComponent>  <Position x="-180.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll), in radians -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="4.0f" y ="10.0f" z="20.0"/>  </TransformableComponent>  <MoveableComponent>  <Velocity vcs="0.0f" vdx="0.0f" vdy="0.0f" vdz="0.0f"/>  <Acceleration ax="0.0f" ay="0.0f" az="0.0f"/>  </MoveableComponent>  </Actor> |

To create this Actor in game, it is necessary to create an ActorFactory and use its CreateActor() method with the path to the XML resource – as illustrate in **Listing 22**. In this example, the new actor was inserted into an ActorMap. Afterwards, the program fetches and prints the position of the actor stored in its TransformableComponent.

**Listing 22.** Creating a game Actor from the resource of **Listing 21**.

|  |
| --- |
| #include <stdio.h>  #include <Core/EntityComponent/Entity/Actor.h>  #include <Core/EntityComponent/Entity/ActorFactory.h>  #include <Core/EntityComponent/Component/ActorComponent.h>  #include <Core/EntityComponent/Component/TransformableComponent.h>  int main()  {  // Actors  uge::ActorMap m\_ActorMap;  uge::ActorFactory m\_ActorFactory;  uge::ActorSharedPointer pActor =  m\_ActorFactory.CreateActor("actor.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pActor->GetActorID(), pActor));  uge::Component::TransformableComponentSharedPointer  pTransformComponent = pActor->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  uge::Vector3 position = pTransformComponent->GetPosition();  printf("Actor's position: (%f, %f, %f)\n",  position.x, position.y, position.z);  pActor.reset();  m\_ActorMap.clear();  return 0;  } |

It is important to note the TransformableComponent header was only include because it was fetched. The created Actor also has a MoveableComponent attached to it.

Due to the data-driven architecture, if the developers wish to change the Actor’s transform or speed, they just want to modify the values in **Listing 21** and re-run the program.

To use the Actor as an archetype, the developers can either use the ModifyActor() method or provide the XML resource. **Listing 23** provides an overriding resource, which changes the Actor’s position and adds a new component (a CollidableComponent).

**Listing 23.** A XML resource modifying for modifying an Actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Actor" resource="actor.xml">  <TransformableComponent>  <Position x="10.0f" y="20.0f" z="30.0f"/>  <!-- YXZ order (yaw, pitch, roll), in radians -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="1.0f" y ="1.0f" z="1.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Sphere">  <Radius r="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  </Actor> |

**Listing 24** updates the source code from **Listing 22** and adds a call to the ModifyActor() method.

**Listing 24.** Using the Actor as an archetype and overriding its data.

|  |
| --- |
| #include <stdio.h>  #include <Core/EntityComponent/Entity/Actor.h>  #include <Core/EntityComponent/Entity/ActorFactory.h>  #include <Core/EntityComponent/Component/ActorComponent.h>  #include <Core/EntityComponent/Component/TransformableComponent.h>  int main()  {  // Actors  uge::ActorMap m\_ActorMap;  uge::ActorFactory m\_ActorFactory;  uge::ActorSharedPointer pActor = m\_ActorFactory.CreateActor(  "actor.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pActor->GetActorID(), pActor));  uge::Component::TransformableComponentSharedPointer  pTransformComponent = pActor->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  uge::Vector3 position = pTransformComponent->GetPosition();  printf("Actor's original position: (%f, %f, %f)\n",  position.x, position.y, position.z);  uge::XMLFile actorOverride;  actorOverride.OpenFile("actor-override.xml", uge::File::FileMode::FileReadOnly);  m\_ActorFactory.ModifyActor(pActor,  &actorOverride.GetRootElement());  position = pTransformComponent->GetPosition();  printf("Actor's new position: (%f, %f, %f)\n",  position.x, position.y, position.z);  pActor.reset();  m\_ActorMap.clear();  return 0;  } |

After running the code, it is possible to see the old and new position. The actor also has its new component: it now has a sphere collision shape.

## Events

Sections 3.3.1.2, 3.4.1.1 and 3.4.1.4 provided an overview on importance and usage of game events for UGE; in fact, event-driven programming can be of great aid for UA-Games implementation [4]. Events are useful and flexible for the both the Game Logic and the Game View layers – it is even possible to use events to tailor and customize the game user interface presentation.

DEFINITION. In simple terms, events are useful to define game actions, activities and happenings. As it is possible to handle events anywhere in the codebase, they are useful for decoupling the game logic, which, in turn, enhance the game flexibility – and, possibility, increases mod potential. Events also contributes to the use of scripting languages in game-development (refer to 4.9).

To illustrate the uses of game events, let us assume a platform game. Jumping is a traditional mechanic in this genre. In this example, let us assume jumping has an animation, a height and a sound.

The jump mechanic could be hard-coded into the implementation (for instance, whenever a button is pressed, the actor jumps). When the jump happens, a flag is set with the information and the implementation handles this change.

Instead of using a flag, it is possible to dispatch an event (ActorJumped, for instance, as in **Figure 18**). This time, instead, any function or method listening to the event can handle it (event handler or callback function) – decoupling the game mechanic implementation from the request. Moreover, it is possible to subscribe multiple handlers to listen to an event.



**Figure 18.** A high-level event.

This lets any subscribed function or method to handle the event as needed – something that either the Game Logic and/or the Game View layers can perform. For instance, an appropriate subsystem of the Game Logic layer can receive the event and change its TransformableComponent height; another may use its CollidableComponent to detect if the actor collided with another actor after the jump; and so on. This is illustrated in **Figure 19** (a).



**Figure 19.** Different layers handling the high-level event of **Figure 18**.

**Figure 19** (b) also suggests how the Game View layer may benefit from the event: to provide feedback to the user. Any interested Game View can subscribe to and event and convey relevant information to the user right after its dispatch. This way, it is possible to provide critical information or to inform the user something happened or exists near the player – which is useful for audio only presentations [5].

This has also a more subtle benefit: it allows conveying implicit feedback. For instance, in a Ping-Pong game, it is usually clear to see when the ball hits a wall or a paddle. However, for a visually impaired user, this is not obvious – it is necessary to convey this information to the user.

Events provide a flexible mechanism to do so. When defining new polymorphic specializations to the game, if some information is missing, it is always possible to define and dispatch a new event in the Game Logic layer to allow providing the required information. This has the added benefit of allowing to improve other specializations: if this new data could improve the gaming experience, the existing specializations can handle this new event and convey the information to the user.

Combined with output components (Section 4.2) and player profiles (Section 4.6), output only event handlers enables the designers to change the game presentation completely without interfering in the Game Logic. Besides, as it is possible to subscribe or unsubscribe handlers during run-time, it also possible to customize the game presentation during run-time. This is explained with more detail in Section 4.6.

The following subsections describe UGE’s event subsystem implementation.

### Functionality

UGE uses the event subsystem created by McShaffry and Graham for GCC4 [12], as their subsystem is flexible enough for the engine needs and integrates with game scripting easily.

Events are classes with immutable, read-only data. As it is possible to attach and detach components during run-time, the class attributes must not be pointers or references –they should be raw data or data-only classes. This is also necessary to ease handling events in game scripting – the communication between application code and script code is usually expensive, so it is important to minimize it when possible.

After creating the event and initializing its data, it is necessary to dispatch the event. The engine provides an event manager for this purpose. When it updates, it sends the event to all subscribed listeners, which are free to read and/or copy the data for their purposes.

### Architecture

**Figure 20** illustrates the IEventData abstract interface, the interface used to implement the data for an event. The figure also shown its reference implementation: the BaseEventData. The difference between this class and the interface is only a time stamp attribute, which is convenient for time-based events. Regardless of the implementation, it is important to note all methods are read-only.



**Figure 20.** The event abstract interface and base implementation.

To create a new event, the developers should implement either the IEventData interface or subclass the BaseEventData. **Figure 21** shows an example for some of the UGE’s default events.



**Figure 21.** Some default events.

In **Figure 21**, it is possible to observer that all events have data-only attributes, such as a matrix or a number – and that all methods are read-only. It is also important to note the EventType attribute: this a Global Unique Identifier (GUID) used as the key to the event; it is necessary to define one for every event class.

To manage sending, updating and removing events, UGE provides the IEventManager abstract interface (**Figure 22**). This interface provides the required methods to handle event operations. Its implementations should iterate on all the registered listeners and events to dispatch the events. It is important to note its Get() method is static; this method should be used to get the only global event manager in the game. Thus, it is possible to define many IEventManagers and use them with the remaining methods (such as vQueueEvent() to add an event to the manager); however, it is only possible to have one global event manager using the interface.



**Figure 22.** The event manager.

The default IEventManager implementation is the EventManager (also in **Figure 22**). It used queues to store events and maps events to listeners using a map data-structure. The next section describes how it works.

### Run-Time

As the EventManager is the reference implementation for the IEventManager, this section examines its run-time. It also describes how to use the manager alongside with the IEventData abstract interface for game events.

The IEventData class has only a few required methods:

* vGetEventType(): used to return the event’s identifier (its GUID);
* vGetTimeStamp(): used to return the time when the event was created;
* vSerialize() and vDeserialize(): used to serialize and de-serialize the event data from an output and to an input stream, respectively;
* vCopy(): used to copy the event data to another event;
* vGetName(): returns the event name. In general, it should be the same name as the class.

The real event data should be provided by the subclass. For instance, in **Figure 21**, the Evt\_Data\_Move\_Actor stores and provides read-only access to the Actor’s ActorID and its world transform.

After defining the event, it is necessary to dispatch it. There are two ways to send an event: using vQueueEvent() and using vTriggerEvent(). The first method (vQueueEvent()) is the preferred way of sending an event: it adds the event to the end of the queue. This way, the event waits its turn to be sent to its listeners in the vUpdate() method. The vUpdate() method iterates on all un-triggered events until they end or until its time limit (in milliseconds).

If, however, it is necessary to trigger the event at the exact moment it occurs, it might be necessary to use the vTriggerEvent() method. This method bypasses the EventManager queue and dispatches the event at the moment of the call.

Regardless of the method, at some moment all registered event handles (also known as listeners or callbacks) receive the event data. The delegate should have a parameter receiving an IEventDataSharedPointer, containing the event’s data. To access the subclass implementation, it is necessary to cast it.

To subscribe to an event, it is necessary to provide the handler delegate (a function or a method) and the event’s EventType to the EventManager’s vAddListener() method. In turn, to unregister, it is necessary to call the vRemoveListener() method.

It is also possible to remove an event from the queue, provided it was not triggered already – for instance, the actor referred in the example does not exist anymore. This is the goal of the vAbortEvent() method. It is important to note event are one-time actions; thus, after triggered, they are removed from the queue.

Therefore, if it is necessary to indicate the beginning and end of an activity, it is necessary to define, at least, two events – for instance, Evt\_Data\_SomethingHappened and Evt\_Data\_SomethingEnded.

### Example

**Listing 25** illustrates how to create an event and use the event system. This listening starts by defining the event. It is important to define the EventType and assign a GUID to it.

Afterwards, the code shows two different handlers: one using a class (EventHandlerMethod in EventClass) and another using a function (EventHandlerFunction). It is necessary to create a delegate to the functions and register the delegate to the desired event.

**Listing 25.** Event subsystem example.

|  |
| --- |
| #include <stdio.h>  #include <Core/Events/Event.h>  #include <Core/Events/EventManager.h>  #include <Utilities/System/Clock.h>  #include <Utilities/System/Time.h>  class Event : public uge::BaseEventData  {  public:  static const uge::EventType sk\_EventType;  explicit Event(unsigned int value)  : m\_Value(value)  {  }  virtual const uge::EventType& vGetEventType() const override  {  return sk\_EventType;  }  virtual uge::IEventDataSharedPointer vCopy() const override  {  return uge::IEventDataSharedPointer(LIB\_NEW Event(m\_Value));  }  virtual void vSerialize(std::ostrstream& out) const override  {  out << m\_Value;  }  virtual void vDeserialize(std::istrstream& in) override  {  in >> m\_Value;  }  virtual const char\* vGetName() const override  {  return "Event";  }  unsigned int GetValue() const  {  return m\_Value;  }  private:  unsigned int m\_Value;  };  const uge::EventType Event::sk\_EventType(0xbd131e2d);  class EventClass  {  public:  EventClass()  {  }  ~EventClass()  {  }  void EventHandlerMethod(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<Event> pData =  std::static\_pointer\_cast<Event>(pEventData);  printf("[EventClass:EventHandlerMethod()] Event received!"  " Value = %u.\n", pData->GetValue());  }  };  void EventHandlerFunction(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<Event> pData =  std::static\_pointer\_cast<Event>(pEventData);  printf("[EventHandlerFunction()] Event received! Value = %u.\n",  pData->GetValue());  }  void RandomlyCreateEvent()  {  const unsigned int kChance = 10;  const unsigned int kJackPot = 100;  unsigned int value = rand() % 100 + 1;  if (value < kChance)  {  std::shared\_ptr<Event> pNewActorEvent(LIB\_NEW Event(value));  // Event will be triggered during the vUpdate() call.  uge::IEventManager::Get()->vQueueEvent(pNewActorEvent);  }  else if (value == kJackPot)  {  std::shared\_ptr<Event> pNewActorEvent(LIB\_NEW Event(value));  // Event will be triggered now, before the vUpdate() call.  uge::IEventManager::Get()->vTriggerEvent(pNewActorEvent);  }  }  int main(int argc, char\* argv[])  {  srand(time(NULL));  // Create the event manager.  uge::IEventManager\* pEventManager =  new uge::EventManager("Global Event Manager", true);  // Creating and registering the event handlers.  uge::EventListenerDelegate functionDelegate =  fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(EventHandlerFunction);  uge::IEventManager::Get()->vAddListener(  functionDelegate, Event::sk\_EventType);  EventClass eventClass;  functionDelegate = fastdelegate::MakeDelegate(  &eventClass, &EventClass::EventHandlerMethod);  uge::IEventManager::Get()->vAddListener(  functionDelegate, Event::sk\_EventType);  // Running.  const unsigned long kMaxTimeToUpdate = 10u;  const unsigned long kTimeToRun = 10000u; // In milliseconds.  unsigned long timeElapsed = 0u;  uge::Time::TimePoint lastTime = uge::Time::GetTime();  while (timeElapsed < kTimeToRun)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  unsigned long timeDelta =  uge::Time::GetDeltaAsMilliseconds(currentTime, lastTime);  RandomlyCreateEvent();  pEventManager->vUpdate(kMaxTimeToUpdate);  lastTime = currentTime;  timeElapsed += timeDelta;  }  printf("\nTime elapsed: %u.\n", timeElapsed);  // Unregistering the event handlers.  functionDelegate = fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(EventHandlerFunction);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, Event::sk\_EventType);  functionDelegate = fastdelegate::MakeDelegate(  &eventClass, &EventClass::EventHandlerMethod);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, Event::sk\_EventType);  // Delete the event manager.  SAFE\_DELETE(pEventManager);  return 0;  } |

The EventManager is created at the main() function. After its creation, it is possible to register the listeners (with a call to vAddListener()). Before ending the program execution (or whenever they are not needed anymore), the event listeners should be removed (with a call to vRemoveListener()).

After creating the EventManager, the program draws a random number and dispatch an event depending on its value. It might be interesting to step the code with a debugger to see the difference between the calls to vQueueEvent() and vTriggerEvent(). (discussed in Section 4.3.3).

Afterwards, the vUpdate() method updates the task manager and triggers the events (until it finishes or the timer reaches the kMaxTimeToUpdate). If no value was given to the function, the method triggers all events in the queue.

When the EventManager is not needed anymore (or before finishing the application), it should be destroyed.

## Game Commands

Section 4.3 discussed game events. As mentioned in 3.3.1.2 and 3.4.1.1, game events are one of the key strategies used for defining game commands in an UGE game. A game command provides a high-level mechanism to abstract and separate human (or even AI) interaction from the game logic. It offers a high-level way of determining the run-time behavior of game entities.

This way, the game logic does not necessarily know or even care about the origin of the game command. It only uses the command’s data to determine what action a game entity should perform. Thus, it is possible to say the game logic is command-driven, as it reacts to game commands to change its flow.

For instance, the ActorJumped event defined in Section 4.3 is actually a game command (refer to **Figure 18** and **Figure 19**). In Section 4.3, its purpose was to allow an actor to jump – let us say, allow the player avatar to jump. In this case, it is a highly specialized command, as it only applies to a single actor.

However, it is possible to make ActorJumped more generic. For instance, adding an ActorID attribute to the event’s definition. This allows using the event with different actors (including with non-human controlled actors), offering a single, and unified action interface: a GameCommand.

GameCommands provides six main benefits – many of each overlap with the benefits of events. They:

1. Decouple the game-logic from the command issuing;
2. Provide a unified, generic command which can be used by different actors;
3. Allow different parts of the codebase to handle the command, if needed;
4. Make it easier to provide input feedback to the user;
5. Ease the creation of different input schemes to the game – especially when combined with a data-driven approach;
6. Allow different ways to interact with the game – makes it possible to explore a same game mechanic in different ways.

Items 1, 2 and 3 are useful for most games and allows for interesting mechanics. For instance, it is possible to make the AI react to an ActorMoved event as the NPC heard the actor’s steps.

Items 4, 5 and 6 are particularly useful for UA-Games. In most graphical games, it is usually easy to perceive the result of an input: the game might provide implicit feedback (such as with the Ping-Pong example in Section 4.3). For instance, it should be possible to see the result of the ActorJumped command. However, for other senses it might not be that clear: a visually impaired user cannot see the result. In this case, an auditory cue might be a better solution.

Item 6 is particularly interesting. As the game logic does not care about the origin of the GameCommand, anything can issue the command. Although apparently trivial, this has a less obvious result: it is possible to change the way the player interacts with the game.

Let us consider players with motor impairments. It might be difficult to issue several game commands at the same time; however, it is possible to automatize certain game commands to make the game easier. Maybe it is hard to move and jump; however, it may be easier to only move or only jump.

The same applies for cognitive impairments, as it allows simplifying the game. Using a series of GameCommands as the input of the game might allow for an auto-play section, similar to demonstration modes. This might ease the difficult and help players whose progress is stuck.

### Functionality

In UGE, there are two types of game commands: low-level and high-level commands (**Figure 23**). High-level game commands are events described so far in this section, such as the ActorJumped. Section 4.3 discussed events in-depth.



**Figure 23.** Low and high-level game commands.

The low-level game command, on the other hand, is much simpler: it is just an enumeration value used to represent a game command. Usually, for each low-level game command there will be one or more high-level game commands – more are required for commands which require states (such as continuous walking when the player holds down a button).

As **Figure 23** suggests, the low-level command is translated into a high-level command. Thus, its goal is not to act as a real command; rather, it is to act as a command stub. The low-level command is a stub for data-driven input mapping (discussed in Section 5.2). It allows using different buttons, keys or even devices to issue the same high-level command without changing the game implementation.

### Architecture

The high-level GameCommand is implemented as an IEventData class. Section 4.4.3 describes its architecture.

The low-level command is much simpler. It is only an integer value, represented as one of three different enumerations: Action, State and Range (**Figure 24**). The developers should choose the most appropriate one for their needs according to the input device used.



**Figure 24.** The low-level game command is a simple value.

Actions are single time game commands that do not need to store state information. As they are performed only once, regardless of whether the player kept holding the button or not, they are suitable for single, slow-paced game commands. To perform an Action again, it is necessary to release the button and press it again. This kind of low-level command needs only a single high-level game command.

States are game commands that requires state information. Although they can be more complex, States are usually similar to on/off switches: they are either enabled or disabled. Thus, it is usually necessary to have two or more high-level game commands for each State: for instance, a CommandEnabled and a CommandDisabled event. States are useful for continuous game commands performed with non-analog devices, such as running using a keyboard as input device.

Ranges are for devices that map its input values in ranges (for instance, analog devices which bases the position from two axes values ranging from -1.0 to 1.0). Ranges are useful to continuous game commands performed with analog devices.

Action, State and Range are detailed in Section 5.2, as they depend on input devices.

### Run-Time

The high-level game command run-time is similar to the event one (Section 4.3.3). The low-level runtime is easier to understand with an example, as it mostly maps a low-level action performed to a high-level game command. Refer to Section 5.2 for more details.

### Example

Let us suppose a human controller to a Ping-Pong game. It may use a state for a move down low-level command and two high-level events for moving the paddle (MovePaddle and StopPaddle). A possible implementation for this example is outlined in **Listing 26**.

**Listing 26.** An example of using game commands.

|  |
| --- |
| #include <Engine/GameController/GameController.h>  #include <Core/PlayerProfile/GraphicalPreferences.h>  #include <IO/Input/InputDevice/Implementation/OIS/OISMouse.h>  #include <IO/Input/InputDevice/Implementation/OIS/OISKeyboard.h>  enum class uge::InputMapping::State : unsigned int  {  Player1MoveUp,  Player1MoveDown,  Player2MoveUp,  Player2MoveDown  };  class Controller  {  public:  void PongGameInputCallback(uge::InputMapping::MappedInput& inputs)  {  bool bState = inputs.IsStateEnabled(  uge::InputMapping::State::Player1MoveUp);  if (bState != m\_LastPlayer1MoveUp)  {  if (bState)  {  std::shared\_ptr<MovePaddle> pEvent(  LIB\_NEW MovePaddle(m\_ActorID, "Up"));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  else  {  std::shared\_ptr<StopPaddle> pEvent(  LIB\_NEW StopPaddle(m\_ActorID));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  m\_LastPlayer1MoveUp = bState;  }  }  private:  bool m\_LastPlayer1MoveUp;  uge::ActorID m\_ActorID;  uge::InputDevice::OISKeyboard m\_Keyboard;  uge::InputDevice::OISMouse m\_Mouse;  }; |

The same idea allows moving the paddle down. If, instead of using human input, the developers sent the high-level commands directly, it would be possible to create an AI player, as it would be able to move the paddle automatically[[11]](#footnote-11).

## Physics

As stated in Sections 2.2 and 3.3, the Game Logic layer simulates the game world. The game world has all game rules and behaviors, defining and managing the actors’ relationships and interaction.

Physics is generally an important part of games. Kinematic and dynamics features are present in most games, even in the simplest ones. For instance, collision detection is a feature present in almost every game.

Due to the importance of Physics, UGE provides a Physics subsystem to manage physical interaction between actors in the game world. It is possible to disable the subsystem, although it might be useful for most games, as it eases the implementation and speed up prototyping.

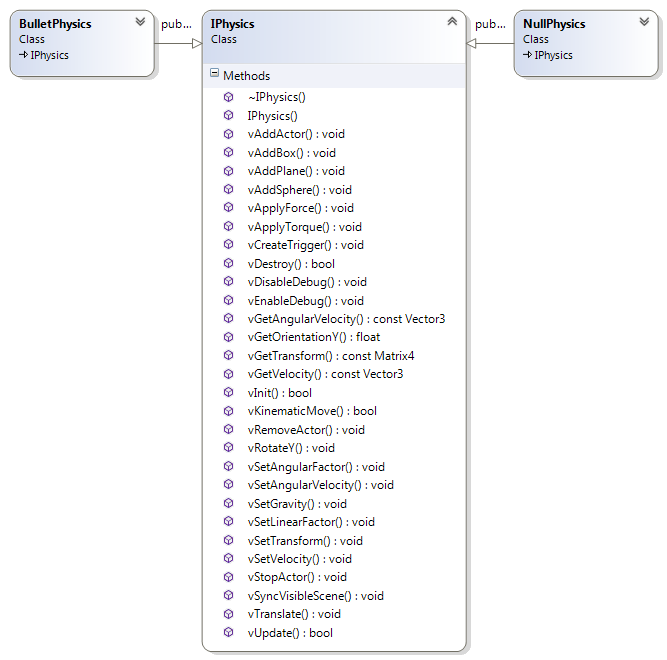
### Functionality

UGE has a Physics subsystem abstract interface (IPhysics). This interface offers a set of rigid body physics and collision detection operations. These operations range from kinematic do dynamics operations, making it easier to operate on the required components (the TransformableComponent, the CollidableComponent and the PhysicsComponent). For instance, it is possible to move, apply forces and torques and to handle collisions without having to do so directly in the components: the Physics subsystem manages it all.

There are two default implementations for the Physics subsystem. The first is NullPhysics, which, as the name suggests, provides no Physics operations at all – it disables physics. The second, as stated in Section 3.3.1.3, uses the Bullet Physics library to provide a full implementation of the IPhysics interface.

### Architecture

The IPhysics abstract interface is illustrated in **Figure 25**, alongside with the two default implementations. It is important to note the available operations: once implemented in a subclass, they ease the game implementation by offering common procedures, such as moving, rotating and check collisions between actors.



**Figure 25.** The Physics subsystem interface.

The NullPhysics implementation overrides all methods with either a blank implementation or a default return value. In other words, it disables game Physics. This is useful in two situations: if the developers do not want to use a Physics library or if they wish to debug the game with a still world.

The BulletPhysics, on the other hand, offers a complete rigid body Physics implementation and collision detection (currently, the available shapes are box, plane and sphere). This is the reference implementation of the IPhysics interface, shall developers wish to create a new one.

### Run-Time

In the same way as many of UGE’s subsystems, a call to the vInit() method of a Physics subsystem is responsible for initializing its data. After this call, the subsystem must be ready for use. For every tick of the game, it is necessary to update the Physics simulation with a call to vUpdate(). Finally, the vSyncVisibleScene() synchronizes the internal data to the actor’s component data.

The Physics simulation requires registered actors to work. To add an actor to the simulation, it is necessary to call the vAddActor() method. After the insertion, the subsystem initializes its required data and, from the next update on, the actor participates on the simulation. Conversely, to remove an actor from the simulation, it is necessary to call vRemoveActor() method.

To ease the use of the subsystem, there are four main Physics events available:

1. EvtData\_PhysTrigger\_Enter: dispatched when an actor enter a trigger;
2. EvtData\_PhysTrigger\_Leave: dispatched when an actor leaves a trigger;
3. EvtData\_PhysCollision: dispatched when a collision between two actors happens;
4. EvtData\_PhysSeparation: dispatched when a collision between two actors finish.

As the events are dispatched when they start and they end, it is correct to consider that, in the meantime, the two actors are still colliding with each other.

As stated in Section 4.5.1, the Physics subsystem operations acts on three different components: the TransformableComponent, the CollidableComponent and the PhysicsComponent. The only exception is the NullPhysics, which does not operate on any component, as it is an empty implementation.

The TransformableComponent stores the world transform of an actor in the game world, defining its position, orientation and scale. The CollidableComponent stores the collision shape, the entities center of mass, density (mass and volume) and material. Finally, the PhysicsComponent stores implementation specific data (mostly regarding dynamics). In the case of BulletPhysics, it holds the available axis for the simulation and max velocity values.

The implementation of the Physics subsystem is also data-driven. It uses XML configuration files to set the components values and for general settings, making it easier to change the data or tweaking the parameters. Next section provides an example of the Physics subsystem in action.

### Example

**Listing 27** presents the Physics configuration file. Currently, its name and path must be "data/config/physics.xml". This file describes general Physics settings, such as the available materials and densities for the CollidableComponent.

**Listing 27.** A sample general settings file for the Physics subsystem.

|  |
| --- |
| <Physics>  <PhysicsMaterials>  <PlayDough restitution="0.05f" friction="0.9f"/>  <Normal restitution="0.25f" friction="0.5f"/>  <Bouncy restitution = "0.95f" friction = "0.5f"/>  <Slippery restitution = "0.25f" friction = "0.0f"/>  <Elastic restitution = "1.00f" friction = "0.0f"/>  </PhysicsMaterials>  <DensityTable>  <Air>0.0013</Air>  <Water>1.000</Water>  <Infinite>0.000</Infinite>  <Kinematic>0.000</Kinematic>  </DensityTable>  </Physics> |

**Listing 28** presents an actor with a sphere CollidableComponent (in other words, a ball). As it is a text file, it is possible to change the values without having to change the game implementation.

**Listing 28.** An actor with Physics related components.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Ball" resource="ball.xml">  <TransformableComponent>  <Position x="0.0f" y="50.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="1.0f" y ="1.0f" z="1.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Sphere">  <Radius r="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Water"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="1.0f" y="1.0f" z="1.0f"/>  <MaxVelocity v="15.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>    </Actor> |

Currently, there is nothing else to the ball actor to interact with – it is the only actor in the game world. **Listing 29** describes a box acting as a plane at the position (0, 0, 0). Provided the ball is falling (which assumes an adequate gravity) and considering its initial height is higher than the plane’s, the two objects should collide provided the simulation runs for enough time.

**Listing 29.** Another actor for the simulation: a simple plane.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Plane" resource="plane.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="1.0f" z="10.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Box">  <Dimension x="1.0f" y="1.0f" z="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Infinite"/>  <Material type="Normal"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="0.0f" y="0.0f" z="0.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="0.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>    </Actor> |

Finally, **Listing 30** presents a bouncing ball example. It describes how to create, initialize and update the Physics subsystem and its effects on the ball’s actor. It is important to note how the actor’s ‘y’ axis value decreases while falling – until it collides with the plane. Afterwards, it then increases after bouncing before start falling again – and repeating the process.

**Listing 30.** A simple physics simulation: a bouncing ball.

|  |
| --- |
| #include <stdio.h>  #include <Core/EntityComponent/Entity/Actor.h>  #include <Core/EntityComponent/Entity/ActorFactory.h>  #include <Core/EntityComponent/Component/TransformableComponent.h>  #include <Core/Events/Event.h>  #include <Core/Events/EventManager.h>  #include <Core/Physics/IPhysics.h>  #include <Core/Physics/Implementation/BulletPhysics/BulletPhysics.h>  #include <Utilities/System/Clock.h>  #include <Utilities/System/Time.h>  int main(int argc, char\* argv[])  {  // Create the event manager.  uge::IEventManager\* pEventManager = new uge::EventManager(  "Global Event Manager", true);  // Create the physics subsystem.  uge::IPhysicsSharedPointer pPhysics(LIB\_NEW uge::BulletPhysics);  if (!pPhysics->vInit())  {  fprintf(stderr,  "Error during the Physics subsystem initialization!\n");  return -1;  }  pPhysics->vSetGravity(uge::Vector3(0.0f, -9.8f, 0.0f));  // Create an actor.  uge::ActorMap m\_ActorMap;  uge::ActorFactory m\_ActorFactory;  uge::ActorSharedPointer pActor = m\_ActorFactory.CreateActor(  "ball.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pActor->GetActorID(), pActor));  // Add the actor to the physics simulation.  pPhysics->vAddActor(pActor);  // Create a plane.  uge::ActorSharedPointer pPlane = m\_ActorFactory.CreateActor(  "plane.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pPlane->GetActorID(), pPlane));  // Add the actor to the physics simulation.  pPhysics->vAddActor(pPlane);  // Simulate the game world for kTimeToRun milliseconds.  const unsigned long kFPS = 1000u / 60u; // In milliseconds.  const unsigned long kTimeToRun = 1000u \* 1000u;// In milliseconds.  unsigned long timeElapsed = 0u;  uge::Time::TimePoint lastTime = uge::Time::GetTime();  while (timeElapsed < kTimeToRun)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  unsigned long timeDelta =  uge::Time::GetDeltaAsNanoseconds(currentTime, lastTime);  // Step the simulation.  pPhysics->vUpdate(timeDelta);  pPhysics->vSyncVisibleScene();  pEventManager->vUpdate();  uge::Component::TransformableComponentSharedPointer  pTransformComponent = pActor->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  uge::Vector3 position = pTransformComponent->GetPosition();  printf("Actor's current position: (%f, %f, %f)\n",  position.x, position.y, position.z);  std::this\_thread::sleep\_for(  std::chrono::milliseconds(kFPS));  lastTime = currentTime;  timeElapsed += (timeDelta / 1000u);  }  // Remove the actor.  pActor.reset();  m\_ActorMap.clear();  // Delete the physics subsystem.  if (!pPhysics->vDestroy())  {  fprintf(stderr,  "Error during the Physics subsystem de-initialization!\n");  return -1;  }  pPhysics.reset();  // Delete the event manager.  SAFE\_DELETE(pEventManager);  return 0;  } |

Should the developers remove the Physics components (CollidableComponent and BulltPhysicsComponent), the Physics subsystem would stop updating the actor.

In **Listing 30**, it is possible to use the position to verify the collision. However, it is much more practical to use the events, as shown in **Listing 31**. This listing add two event handlers to **Listing 30**: CollisionStarted() and CollisionEnded() for, respectively, the EvtData\_PhysCollision and the EvtData\_PhysSeparation events.

**Listing 31.** Adding events to the simulation of **Listing 30**.

|  |
| --- |
| #include <stdio.h>  #include <Core/EntityComponent/Entity/Actor.h>  #include <Core/EntityComponent/Entity/ActorFactory.h>  #include <Core/EntityComponent/Component/TransformableComponent.h>  #include <Core/Events/Event.h>  #include <Core/Events/EventManager.h>  #include <Core/Physics/IPhysics.h>  #include <Core/Physics/PhysicsEvents.h>  #include <Core/Physics/Implementation/BulletPhysics/BulletPhysics.h>  #include <Utilities/System/Clock.h>  #include <Utilities/System/Time.h>  void CollisionStarted(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<uge::EvtData\_PhysCollision> pData =  std::static\_pointer\_cast<  uge::EvtData\_PhysCollision>(pEventData);  printf("[CollisionStarted()] Actors %u and %u collided!\n",  pData->GetActorA(), pData->GetActorB());  }  void CollisionEnded(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<uge::EvtData\_PhysSeparation> pData =  std::static\_pointer\_cast<  uge::EvtData\_PhysSeparation>(pEventData);  printf("[CollisionEnded()] Actors %u and %u"  "stopped colliding!\n",  pData->GetActorA(), pData->GetActorB());  }  int main(int argc, char\* argv[])  {  // Create the event manager.  uge::IEventManager\* pEventManager =  new uge::EventManager("Global Event Manager", true);  // Creating and registering the event handlers.  uge::EventListenerDelegate functionDelegate =  fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(CollisionStarted);  uge::IEventManager::Get()->vAddListener(  functionDelegate, uge::EvtData\_PhysCollision::sk\_EventType);  functionDelegate = fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(CollisionEnded);  uge::IEventManager::Get()->vAddListener(functionDelegate,  uge::EvtData\_PhysSeparation::sk\_EventType);  // Create the physics subsystem.  uge::IPhysicsSharedPointer pPhysics(LIB\_NEW uge::BulletPhysics);  if (!pPhysics->vInit())  {  fprintf(stderr,  "Error during the Physics subsystem initialization!\n");  return -1;  }  pPhysics->vSetGravity(uge::Vector3(0.0f, -9.8f, 0.0f));  // Create an actor.  uge::ActorMap m\_ActorMap;  uge::ActorFactory m\_ActorFactory;  uge::ActorSharedPointer pActor = m\_ActorFactory.CreateActor(  "ball.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pActor->GetActorID(), pActor));  // Add the actor to the physics simulation.  pPhysics->vAddActor(pActor);  // Create a plane.  uge::ActorSharedPointer pPlane = m\_ActorFactory.CreateActor(  "plane.xml", nullptr);  m\_ActorMap.insert(std::make\_pair(pPlane->GetActorID(), pPlane));  // Add the actor to the physics simulation.  pPhysics->vAddActor(pPlane);  // Simulate the game world for kTimeToRun milliseconds.  const unsigned long kFPS = 1000u / 60u; // In milliseconds.  const unsigned long kTimeToRun = 1000u \* 1000u;// In milliseconds.  unsigned long timeElapsed = 0u;  uge::Time::TimePoint lastTime = uge::Time::GetTime();  while (timeElapsed < kTimeToRun)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  unsigned long timeDelta =  uge::Time::GetDeltaAsNanoseconds(currentTime, lastTime);  // Step the simulation.  pPhysics->vUpdate(timeDelta);  pPhysics->vSyncVisibleScene();  pEventManager->vUpdate();  uge::Component::TransformableComponentSharedPointer  pTransformComponent =  pActor->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  uge::Vector3 position = pTransformComponent->GetPosition();  printf("Actor's current position: (%f, %f, %f)\n",  position.x, position.y, position.z);  std::this\_thread::sleep\_for(  std::chrono::milliseconds(kFPS));  lastTime = currentTime;  timeElapsed += (timeDelta / 1000u);  }  // Remove the actor.  pActor.reset();  m\_ActorMap.clear();  // Removing the event handlers.  functionDelegate = fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(CollisionStarted);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, uge::EvtData\_PhysCollision::sk\_EventType);  functionDelegate = fastdelegate::FastDelegate1<  uge::IEventDataSharedPointer>(CollisionEnded);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, uge::EvtData\_PhysSeparation::sk\_EventType);  // Delete the physics subsystem.  if (!pPhysics->vDestroy())  {  fprintf(stderr,  "Error during the Physics subsystem de-initialization!\n");  return -1;  }  pPhysics.reset();  // Delete the event manager.  SAFE\_DELETE(pEventManager);  return 0;  } |

This way, whenever a collision starts or ends, the event is triggered and the desired code is run.

## Player Profiles

Due to the diverse demographic of possible players for a UA-Game, alongside with all the different interaction needs and capabilities, it is necessary to allow the game to adapt itself to the player’s particular needs.

Sections 3, 4.2 and 4.3 discussed different strategies to increase the run-time flexibility of the game implementation. It is, however, necessary to offer a structured way to provide the customization information to the game.

UGE achieves this using different player profiles. A player profile is a structured document containing different parameters to modify the game during run-time. Combined with the data-driven architecture of the engine, a player profile allows changing various different configurations from the game, ranging from an actor to the entire game presentation to the input device and command scheme.

It is possible to define player profiles for a specific user or for a group of user. This allows to tailor the game to user needs or to provide accessible presets for different user interaction abilities. It is also possible to use a preset as the basis for creating a user specific profile.

### Functionality

In UGE, there are two parts of a player profile: several XML resource files containing the configuration data and a PlayerProfile class, which provides easy access for the data parsed from the XML files. A game many have many player profile; in fact, the PlayerProfiles class acts as a manager for all the game profiles the developers wish to load.

A player profile stores data regarding IO, gameplay and general game preferences. The data stored also varies: some might be general, whilst other can be more specific. For instance, general settings (such as language) are usually applicable to the entire game application. As such, they are usually useful during the game initialization.

IO and gameplay settings are usually more specific, as they allow customizing the way the player interacts with the game and receiving the game’s stimuli. These settings might vary even between game states, as it is not uncommon to change the interaction in different states (for instance, a game menu is usually different from the game itself).

Thus, whenever a new game state is loaded, the active PlayerProfile is offered to the new state. This allows the game to customize itself, adding components to actors, changing existing component data, adding or removing event handlers, changing the game presentation or game interaction, among others possibilities.

Besides, as the game is data-driven, with an adequate implementation, it is even possible to change the active profile whilst the game is running. Scripting languages (Section 4.9) offer many interesting benefits to help achieving this.

### Architecture

To make it simpler and avoid duplication, the XML files are described in Section 4.6.4, whilst this section describes the PlayerProfile class.

The PlayerProfile is illustrated in **Listing 32**. As its attributes and methods suggests, it is a container for the XML resource files’ data. It is important to note this class is read-only: all accessory methods return constant references.

**Listing 32.** The PlayerProfile class.



To make the interface more extensible, several specialized classes split the PlayerProfile data. The classes are:

1. AuralPreferences: stores data for different game audio types, such as music, sound effects and speech settings;
2. GameplayPreferences: keeps gameplay related data. This data is useful for customizing game actors and event specialization during the game play. This data is available for each game state during its initialization;
3. GraphicalPreferences: holds data graphical renderer, text rendering and window settings;
4. InputPreferences: stores the data used for game input, such as the desired input device and input and command mapping;
5. PlayerPreferences: stores general game settings, such as language. These settings are usually valid for the entire game application.

Each of the classes reflect their XML resource file correspondent (described in Section 4.6.4).

As a UA-Game usually has several different game profiles, UGE provides a PlayerProfile manager: the PlayerProfiles class. This class is a high-level container for all the game profiles.

**Listing 33.** The PlayerProfile manager.



It also stores the active profile, which allows the game to tailor itself for the current player needs.

### Run-Time

The player profile run-time is simple. From a PlayerProfile perspective, everything is ready after calling the Init() method. This method loads a PlayerProfile from the resource file passed as parameter, parsing its contents and loading each of the preferences classes (AuralPreferences, GameplayPreferences, GraphicalPreferences, InputPreferences and PlayerPreferences).

Loading many profiles at once is similar: it only requires passing the profile list resource to the Init() method of the PlayerProfiles class. As this class also manages the active profile, it is possible to set and retrieve it (respectively, with the SetCurrentProfile() and GetCurrentProfile() methods).

The complexity and usefulness of the profiles come from the game’s implementation. The GameApplication can retrieve the current profile at any moment, allowing to change the game.

On the other hand, all game states receive the active profile during their initialization – the IGameState abstract interface provides the method vTailorToProfile() for this purpose. This way, it is possible to tweak and add relevant data to the actors and events to suit the gameplay to the user interaction abilities.

### Example

This section has two objectives: to show and describe the player profile’s XML resource file and to show how to load its data into the PlayerProfile class.

UGE allows using one more player profile. A list of player profiles is shown in **Listing 34**.

**Listing 34.** A list of player profiles.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfiles resource="player\_profiles/player\_profiles.xml">  <PlayerProfile name="Average User: Default"  resource="player\_profiles/average\_user/profile.xml"/>  <PlayerProfile name="Visually Impaired: Blind"  resource="player\_profiles/visually\_impaired/profile.xml"/>  </PlayerProfiles> |

This example will examine the first profile in the list (the "Average User: Default"). All other profiles should be similar, only changing the settings, except case stated otherwise.

The name of a profile is its key in the PlayerProfiles manager. A player profile follows the structure of **Listing 35**, containing four different elements: the GeneralSettings, the InputSettings, the OutputSettings and the GameplaySettings. Each of the elements contains one or more sub-elements, which refer to another XML resource file.

**Listing 35**. A sample profile.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfile name="Average User: Default"  resource="player\_profiles/average\_user/profile.xml">  <GeneralSettings>  <Preferences  resource="player\_profiles/average\_user/preferences.xml"/>  </GeneralSettings>  <InputSettings>  <Mapping  resource="player\_profiles/average\_user/input\_mapping.xml"/>  </InputSettings>  <OutputSettings>  <PrimaryOutput type="graphical"  resource="player\_profiles/average\_user/graphics.xml"  events="player\_profiles/average\_user/graphical\_events.xml"/>  <SecondaryOutput type="aural"  resource="player\_profiles/average\_user/audio.xml"  events="player\_profiles/average\_user/aural\_events.xml"/>  </OutputSettings>  <GameplaySettings>  <EntitySpecialization  resource="player\_profiles/average\_user/entity/entities.xml"/>  <EventSpecializations  resource="player\_profiles/average\_user/events/events.xml"/>  </GameplaySettings>  </PlayerProfile> |

Although all the profile content could be described into one single file, separating them into multiple resources contributes to reusing and makes it easier to create custom user profiles.

The GeneralSettings elements describes a XML resource containing the PlayerPreferences data (mentioned in Section 4.6.2). Currently, it is very simple; **Listing 36** illustrates one.

**Listing 36.** The general settings resource.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerPreferences player\_name="user"  resource="player\_profiles/average\_user/preferences.xml">  <Language name="English" />  </PlayerPreferences> |

Next in **Listing 35** comes the InputSettings. This element describes the input mapping for the game (as briefly stated in Section 4.4). An example is available at **Listing 37**.

**Listing 37.** Input mapping resource.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource="player\_profiles/average\_user/input\_mapping.xml">  <Actions>  <!-- Actions go here. -->  </Actions>  <States>  <State game\_command\_name="Player1MoveUp"  device="keyboard" input\_value="KC\_W"/>  <State game\_command\_name="Player1MoveDown"  device="keyboard" input\_value="KC\_S"/>  <State game\_command\_name="Player2MoveUp"  device="keyboard" input\_value="KC\_UP"/>  <State game\_command\_name="Player2MoveDown"  device="keyboard" input\_value="KC\_DOWN"/>  </States>  <Ranges>  <!-- Actions go here. -->  </Ranges>  </Commands> |

As this is an external configuration file, it is possible to change the device or mapping without modifying the game code. This provides greater flexibility for tailoring the game.

Next in **Listing 35** comes the OutputSettings. It provides the data for the AuralPreferences and GraphicalPreferences (mentioned in Section 4.6.2). These settings define the way the game should convey its information to the player – it defines the user presentation. The elements define the desired output subsystems, which are divided in two: one for the PrimaryOutput and another for the SecondaryOutput. Which subsystem will be primary or secondary depends on the user abilities.

Both output system requires three attributes: its type, a configuration resource and a list of all the available events for this output. **Listing 38** presents the configuration for the graphics subsystem.

**Listing 38.** The graphics subsystem configuration.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Graphics resource="player\_profiles/average\_user/graphics.xml">  <Window width="1024" height="768"  pixel\_depth="32" vsync="false" fullscreen="false" />  <Renderer name="OpenGL Rendering Subsystem"/>  <!-- Other options:  Direct3D9 Rendering Subsystem  Direct3D10 Rendering Subsystem  Direct3D11 Rendering Subsystem  OpenGL Rendering Subsystem  OpenGL ES 1.x Rendering Subsystem  -->  <Text>  <Font resource="font\_name" size="10" />  <Subtitles enabled="false" />  </Text>  </Graphics> |

Currently, it also stores the window and text rendering settings.

**Listing 39** presents the audio subsystem settings. It has three sound groups: Music, sound effects (SFX) and Speech. It is possible to enable or disable each of these groups and to set a default volume for the group.

**Listing 39.** The audio subsystem configuration.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Audio resource="player\_profiles/average\_user/audio.xml">  <Settings type="stereo" />  <Music enabled="true" volume="1.0f" />  <SFX enabled="true" volume="1.0f" />  <Speech enabled="true" volume="1.0f" />  </Audio> |

Finally, the developers should define the available events for output specialization (**Listing 40**).

**Listing 40.** A list of game events.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <EventSpecialization  resource="player\_profiles/average\_user/events/ events.xml">  <Event name="OnRestartGame"/>  <Event name="OnPlayerScored"/>  <Event name="OnBallWallCollision"/>  <Event name="OnBallPaddleCollision"/>  <Event name="OnPaddleWallCollision"/>  <Event name="OnMovePaddle"/>  <Event name="OnStopPaddle"/>  </Events>    </EventSpecialization> |

The final element is **Listing 35** is the GameplaySettings, which defines the data for the GameplayPreferences (mentioned in Section 4.6.2). The GameplaySettings have two sub-elements: EntitySpecialization and EventSpecialization.

The EntitySpecialization allows to override component settings and to add output components to define the game output. An entity list definition is similar to the one in **Listing 41**.

**Listing 41.** A list of entities to specialize in the profile.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actors resource="player\_profiles/average\_user/entity/entities.xml">  <Actor name="Ball"  resource="player\_profiles/average\_user/entity/ball.xml"/>  <Actor name="Paddle-Player1"  resource="player\_profiles/average\_user/entity/paddle-player1.xml"/>  <Actor name="Paddle-Player2"  resource="player\_profiles/average\_user/entity/paddle-player2.xml"/>  <Actor name="Plane"  resource="player\_profiles/average\_user/entity/plane.xml"/>  <Actor name="Wall-Bottom"  resource="player\_profiles/average\_user/entity/wall-bottom.xml"/>  <Actor name="Wall-Left" resource="player\_profiles/average\_user/entity/wall-left.xml"/>  <Actor name="Wall-Right"  resource="player\_profiles/average\_user/entity/wall-right.xml"/>  <Actor name="Wall-Top"  resource="player\_profiles/average\_user/entity/wall-top.xml"/>    </Actors> |

**Listing 42** illustrated a specialization for the actor of **Listing 28**. The specialization keep all the original components the actor already has and adds the new ones (in this case, the OgreGraphicalComponent and the OpenALSoftAudioComponent). If there are redefinition for existing components, its data is overridden. This is the case of the TransformableComponent, which had its scale increased (from 1.0 to 3.0).

**Listing 42.** A specialized actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Ball"  resource="player\_profiles/average\_user/entity/ball.xml">  <TransformableComponent>  <Position x="0.0f" y="50.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="3.0f" y ="3.0f" z="3.0"/>  </TransformableComponent>  <OgreGraphicalComponent>  <NodeName n="Ball-Graphics" />  <MeshFileName m="sphere.mesh" />  <MaterialFileName n="" />  </OgreGraphicalComponent>    <OpenALSoftAudioComponent>  <NodeName n="Ball-Audio" />  <FileName n="data/audio/effects/Pickup\_Coin.wav" />  <Volume v="1.0f" />  <InitialProgress p="0.0f" />  <Loop l="true" />  </OpenALSoftAudioComponent>    </Actor> |

Thus, the resulting actor defined in **Listing 28** and specialized in **Listing 42** has a TransformableComponent, a CollidableComponent, a BulletPhysicsComponent, an OgreGraphicalComponent and an OpenALSoftAudioComponent. As it have both graphical and aural components, the graphical subsystem will render it into the window while the audio subsystem will play its representation sound.

To use the profiles into the game, it is necessary to load them. **Listing 43** shows a simple example the required code.

**Listing 43.** Loading the profile data into an application.

|  |
| --- |
| #include <Core/PlayerProfile/PlayerProfiles.h>  int main()  {  // Loading a single profile.  uge::PlayerProfile profile;  profile.Init("player\_profiles/average\_user/profile.xml");  // Loading a list of profiles.  uge::PlayerProfiles profiles;  profiles.Init("player\_profiles/player\_profiles.xml");  return 0;  } |

After loaded, they are ready for use.

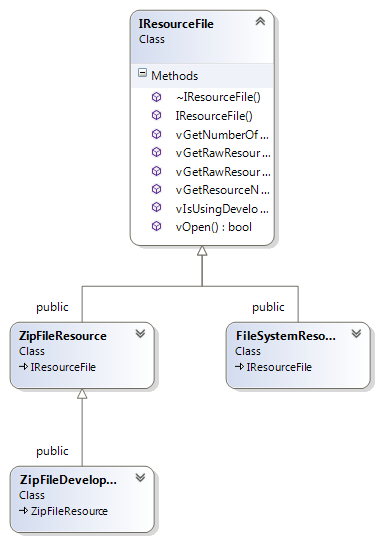
## Resources

### Functionality

### Architecture







### Run-Time

### Example

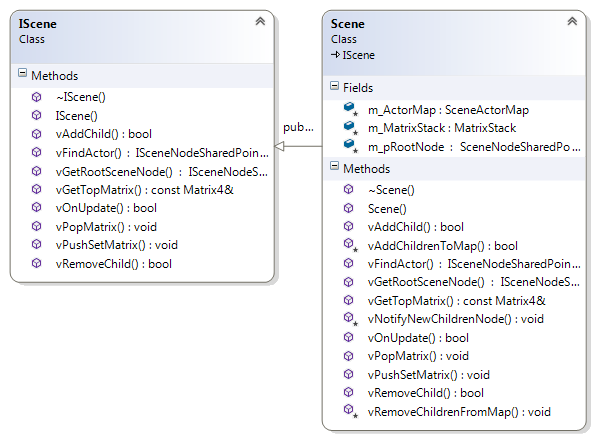
|  |
| --- |
| Hello, world! |

|  |
| --- |
| #include <Core/Resource/ResourceCache.h>  #include <Core/Resource/ZipFileResource.h>  #include <Utilities/Debug/Logger.h>  #include <Utilities/File/ZipFile.h>  #include <iostream>  int main()  {  uge::IResourceFile\* pResourceFile =  new uge::ZipFileResource("ResourceCache.zip");  uge::ResourceCache resourceCache;  if (!resourceCache.Init(10, pResourceFile)) // 10MB  {  std::cerr << "Could not create the resource cache!"  << std::endl;  return -1;  }  uge::Resource resource("text.txt");  uge::ResourceHandleSharedPointer pHandle =  resourceCache.GetHandle(&resource);  assert((pHandle != nullptr) && "Error loading the file!");  int textLength = pHandle->GetSize();  assert((textLength > 0) && "Error loading the file!");  char\* pText = new char[textLength + 1];  memcpy(pText, (char\*) pHandle->GetData(), textLength);  pText[textLength] = '\0';  std::cout << pText << std::endl;    // The resource file is deleted by the resource cache.  return 0;  } |

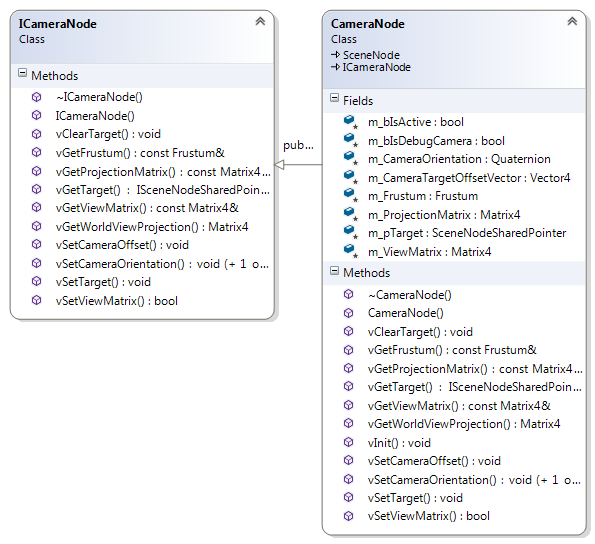
## Scenes

### Functionality

### Architecture

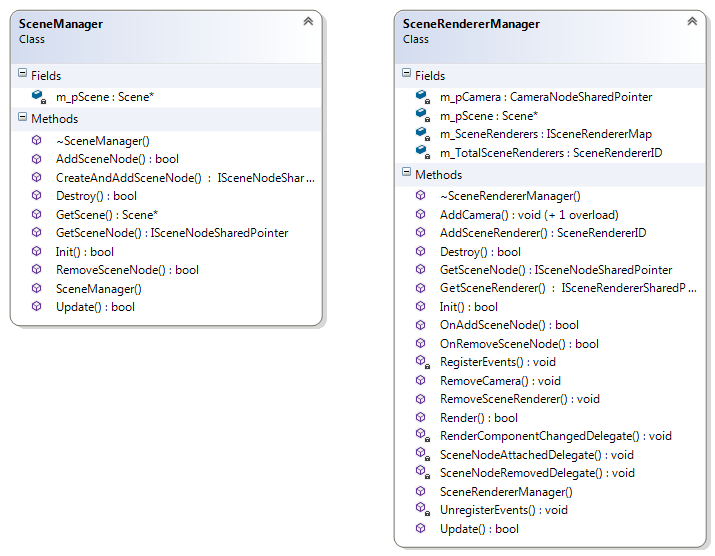












### Run-Time

### Example

## Scripting

### Functionality

### Architecture



### Run-Time

### Example

|  |
| --- |
| function print\_prompt(times)  local write = io.write  for i = 1, times, 1 do  write(">")  end    write(" ")  end  function read\_string()  return read\_string("\*l")  end  write = io.write  read\_string = io.read  write("Lua simple shell\n")  write("Enter 'quit' to exit.\n")  -- Very simple REPL in Lua.  level = 1  print\_prompt(level)  command\_string = read\_string()  while (command\_string ~= "quit") do  -- Run the given command. The output is written on the screen.  -- assert(loadstring(command\_string))()  out\_string, err\_string = loadstring(command\_string)  if (out\_string ~= nil) then  -- Success!  out\_string()  else  -- Something went wrong!  write(err\_string, "\n")  end  level = 1  print\_prompt(level)  command\_string = read\_string()  end |

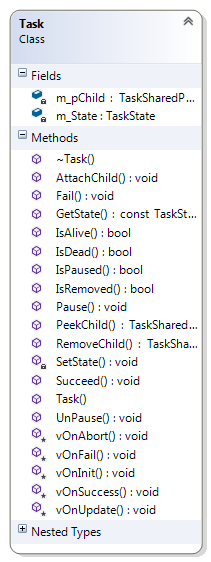
|  |
| --- |
| #include <Core/Script/Lua/LuaStateManager.h>  #include <Core/Script/Lua/ScriptTask.h>  #include <Core/Script/Lua/ScriptExports.h>  #include <Utilities/Debug/Logger.h>  int main(int argc, char\* argv[])  {  uge::debug::log::Init("data/debug/LogConfig.xml");  if (!uge::LuaStateManager::Create())  {  std::cerr << "Error creating the state manager!" << std::endl;  return -1;  }  uge::IScriptManager\* pScriptManager =  uge::LuaStateManager::Get();  pScriptManager->vExecuteFile("PreInit.lua");  uge::ScriptExports::Register();    pScriptManager->vExecuteString("x = 1");  pScriptManager->vExecuteString("x = x + 10");  pScriptManager->vExecuteString("print(x)");  pScriptManager->vExecuteString("= x");  pScriptManager->vExecuteString("fileName = \"LuaTask.lua\"");  // A very simple Lua shell.  LuaPlus::LuaState\* pLuaState =  uge::LuaStateManager::Get()->GetLuaState();  pLuaState->DoFile("Shell.lua");  pScriptManager->vExecuteString("print(\"bye!\")");  pScriptManager->vExecuteString("io.write(x)");  pScriptManager = nullptr;  uge::LuaStateManager::Destroy();  uge::ScriptExports::Unregister();  uge::debug::log::Destroy();  return 0;  } |

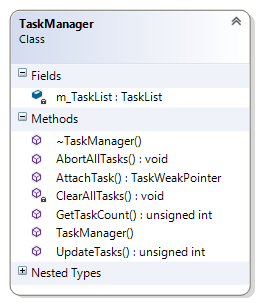
## Tasks

Processing takes multiple cycles/frames of the game – difference of events.

### Functionality

### Architecture





### Run-Time

### Example



|  |
| --- |
| #include <Core/Task/Task.h>  #include <Core/Task/TaskManager.h>  #include <Core/Task/DefaultTasks/TimeDelayTask.h>  #include <Utilities/System/Time.h>  class PrintTask : public uge::Task  {  public:  explicit PrintTask(  std::string message, const unsigned long times)  {  m\_Message = message;  m\_Times = times;  m\_Counter = 0;  }  protected:  virtual void vOnUpdate(const unsigned long dt)  {  if (m\_Counter < m\_Times)  {  std::cout << m\_Message << std::endl;  ++m\_Counter;  }  else  {  Succeed();  }  }  private:  unsigned long m\_Times;  unsigned long m\_Counter;  std::string m\_Message;  };  int main()  {  // Tasks  uge::TaskManager m\_TaskManager;  uge::TaskSharedPointer pDelayTask(  LIB\_NEW uge::TimeDelayTask(5000));  m\_TaskManager.AttachTask(pDelayTask);  uge::TaskSharedPointer pAnotherDelayTask(  LIB\_NEW uge::TimeDelayTask(1000));  pDelayTask->AttachChild(pAnotherDelayTask);  uge::TaskSharedPointer pTask(LIB\_NEW PrintTask("foo", 100));  m\_TaskManager.AttachTask(pTask);  pTask.reset(LIB\_NEW PrintTask("bar", 150));  m\_TaskManager.AttachTask(pTask);  uge::Time::TimePoint startTime = uge::Time::GetTime();  while (m\_TaskManager.GetTaskCount() > 0)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  #if \_DEBUG  unsigned long deltaMilliseconds =  uge::Time::GetDeltaAsMilliseconds(currentTime, startTime);  std::cout << "Time elapsed: " << deltaMilliseconds  << " ms." << std::endl;  #else  // Milliseconds times is too fast for those tasks in release.  unsigned long deltaMilliseconds =  Time::GetDeltaAsNanoseconds(currentTime, startTime);  deltaMilliseconds /= 100;  std::cout << "Time elapsed: " << deltaMilliseconds  << " ns." << std::endl;  #endif  m\_TaskManager.UpdateTasks(deltaMilliseconds);  startTime = currentTime;  }  return 0;  } |

# UGE IO

## Introduction

## Input



### Input Mapping

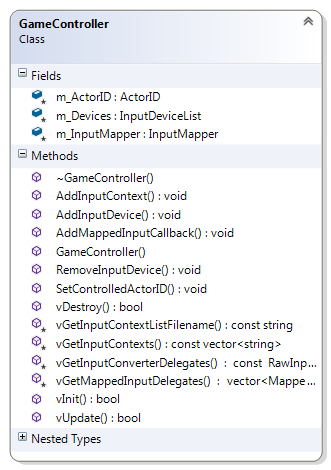












## Output





### Audio





|  |
| --- |
| #include <Core/Resource/Resource.h>  #include <Core/Resource/ResourceCache.h>  #include <Core/Resource/XMLResource.h>  #include <Core/Resource/ZipFileResource.h>  #include <Core/Script/Lua/ScriptResource.h>  #include <Core/Task/TaskManager.h>  #include <IO/Output/Audio/Audio.h>  #include <IO/Output/Audio/AudioTask.h>  #include <IO/Output/Audio/Implementation/OpenALSoft/OpenALSoftAudio.h>  #include <IO/Output/Audio/Implementation/OpenALSoft/OpenALSoftAudioResource.h>  #include <Utilities/Debug/Logger.h>  #include <Utilities/System/Time.h>  int main(int argc, char\* argv[])  {  const int TOTAL\_BUFFERS = 4u;  std::shared\_ptr<uge::Audio> pAudio =  std::shared\_ptr<uge::OpenALSoftAudio>(LIB\_NEW  uge::OpenALSoftAudio(TOTAL\_BUFFERS));  pAudio->vInit();  uge::TaskManager taskManager;  uge::IResourceFile\* pResourceFile =  LIB\_NEW uge::ZipFileResource("data.zip");  uge::ResourceCache resourceCache;  if (!resourceCache.Init(100, pResourceFile)) // 100MB  {  std::cerr << "Error creating the resource cache!"  << std::endl;  return -1;  }  uge::IResourceLoaderSharedPointer pCreateXMLLoader =  uge::XMLResourceLoader::CreateLoader();  resourceCache.RegisterLoader(pCreateXMLLoader);  uge::IResourceLoaderSharedPointer pCreateLuaScriptLoader =  uge::LuaScriptResourceLoader::CreateLoader();  resourceCache.RegisterLoader(pCreateLuaScriptLoader);  // OGG loader.  resourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(LIB\_NEW  uge::OpenALSoftOggResourceLoader));  // WAVE loader.  resourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(LIB\_NEW  uge::OpenALSoftWaveResourceLoader));  // Load a sound resource and start playing it.  uge::Resource oggAudioResourceFile("sound.ogg");  uge::ResourceHandleSharedPointer pOggAudioResource =  resourceCache.GetHandle(&oggAudioResourceFile);  uge::IAudioBuffer\* pAudioBuffer =  pAudio->vInitAudioBuffer(pOggAudioResource);  pAudioBuffer->vPlay(0.5f, true);  // Load a sound resource and attach it to a task.  uge::Resource wavAudioResourceFile("sound.wav");  uge::ResourceHandleSharedPointer pWavAudioResource  = resourceCache.GetHandle(&wavAudioResourceFile);  std::shared\_ptr<uge::SoundTask>soundTask(LIB\_NEW  uge::SoundTask(pAudio, pWavAudioResource, 1.0f, true));  taskManager.AttachTask(soundTask);  uge::Time::TimePoint startTime = uge::Time::GetTime();  while (true)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  unsigned long deltaNanoseconds =  uge::Time::GetDeltaAsNanoseconds(currentTime, startTime);  taskManager.UpdateTasks(deltaNanoseconds);  pAudio->vUpdate(deltaNanoseconds);  startTime = currentTime;  }  pAudio->vDestroy();  SAFE\_DELETE(pResourceFile);  return 0;  } |

|  |
| --- |
| #include <Core/Resource/Resource.h>  #include <Core/Resource/ResourceCache.h>  #include <Core/Resource/XMLResource.h>  #include <Core/Resource/ZipFileResource.h>  #include <Core/Script/Lua/ScriptResource.h>  #include <Core/Task/TaskManager.h>  #include <IO/Output/Audio/Audio.h>  #include <IO/Output/Audio/AudioTask.h>  #include <IO/Output/Audio/Implementation/YSE/YSEAudio.h>  #include <IO/Output/Audio/Implementation/YSE/YSEAudioResource.h>  #include <Utilities/Debug/Logger.h>  #include <Utilities/System/Time.h>  int main(int argc, char\* argv[])  {  std::shared\_ptr<uge::Audio> pAudio =  std::shared\_ptr<uge::YSEAudio>(LIB\_NEW uge::YSEAudio);  pAudio->vInit();  uge::TaskManager taskManager;  uge::IResourceFile\* pResourceFile =  LIB\_NEW uge::ZipFileResource("data.zip");  uge::ResourceCache resourceCache;  if (!resourceCache.Init(100, pResourceFile)) // 100MB  {  std::cerr << "Error creating the resource cache!"  << std::endl;  return -1;  }  uge::IResourceLoaderSharedPointer pCreateXMLLoader =  uge::XMLResourceLoader::CreateLoader();  resourceCache.RegisterLoader(pCreateXMLLoader);  uge::IResourceLoaderSharedPointer pCreateLuaScriptLoader =  uge::LuaScriptResourceLoader::CreateLoader();  resourceCache.RegisterLoader(pCreateLuaScriptLoader);  // OGG loader.  resourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(LIB\_NEW  uge::YSEOggResourceLoader));  // WAVE loader.  resourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(LIB\_NEW  uge::YSEWaveResourceLoader));  // Load a sound resource and start playing it.  uge::Resource oggAudioResourceFile("sound.ogg");  uge::ResourceHandleSharedPointer pOggAudioResource =  resourceCache.GetHandle(&oggAudioResourceFile);  uge::IAudioBuffer\* pAudioBuffer =  pAudio->vInitAudioBuffer(pOggAudioResource);  pAudioBuffer->vPlay(1.0f, true);  // Load a sound resource and attach it to a task.  uge::Resource wavAudioResourceFile("sound.wav");  uge::ResourceHandleSharedPointer pWavAudioResource =  resourceCache.GetHandle(&wavAudioResourceFile);  std::shared\_ptr<uge::SoundTask>soundTask(  LIB\_NEW uge::SoundTask(pAudio, pWavAudioResource, 0.2f, true));  taskManager.AttachTask(soundTask);  uge::Time::TimePoint startTime = uge::Time::GetTime();  while (true)  {  uge::Time::TimePoint currentTime = uge::Time::GetTime();  unsigned long deltaNanoseconds = uge::Time::GetDeltaAsNanoseconds(currentTime, startTime);  taskManager.UpdateTasks(deltaNanoseconds);  pAudio->vUpdate(deltaNanoseconds);  startTime = currentTime;  }  pAudio->vDestroy();  SAFE\_DELETE(pResourceFile);  return 0;  } |

### Graphics



# UGE Utilities

## Introduction

## Debug

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <LogConfig>  <Log tag="TAG" debugger="1" file="1"/>  <Log tag="Debugger only" debugger="1" file="0"/>  <Log tag="Log file only" debugger="0" file="1"/>  </LogConfig> |

|  |
| --- |
| #include <Utilities/Debug/Logger.h>  int main()  {  // Create the log file using the configuration file.  // The log file is created in "data/debug/log{\_d}.log".  uge::debug::log::Init("data/debug/LogConfig.xml");  // The logger can be used with the following function:  uge::debug::log::Log("TAG",  "Error message", \_\_FUNCTION\_\_, \_\_FILE\_\_, \_\_LINE\_\_);  // The TAG must exists - it can be one of the default ones or  // defined in the log configuration file.    // Or with these macros:  LOG\_INFO("Information: "); // Uses INFO tag.  LOG\_WARNING("Warning: "); // Uses WARNING tag.    LOG\_ASSERT(true && "Message"); // Logs the assertion case  // it evaluates to false.  LOG\_FATAL("Fatal Error!"); // The game cannot continue to run  // if this happens.  LOG\_ERROR("Error: "); // Serious, but non-fatal errors.    // Destroy the log file.  uge::debug::log::Destroy();  return 0;  } |

## File

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <rootElement>  <foo string="value">  <bar>  <foobar>Some text value</foobar>  </bar>  </foo>  <quux string="abc" wstring="Hello!" int="-1"  uint="100" float="1.23" double="-1.23" boolean="true"/>  </rootElement> |

|  |
| --- |
| #include <Utilities/File/XMLFile.h>  int main()  {  uge::XMLFile xmlFile;  if (!xmlFile.OpenFile("XMLFile.xml",  uge::File::FileMode::FileReadOnly))  {  std::cerr << "Could not load the file." << std::endl;  exit(EXIT\_FAILURE);  }  uge::XMLElement xmlRootElement(xmlFile.GetRootElement());  if (!xmlRootElement.IsGood())  {  std::cerr << "Could not read the root element from the file."  << std::endl;  exit(EXIT\_FAILURE);  }  std::cout << "Root Element: " << xmlRootElement.GetElementName()  << std::endl;  uge::XMLElement xmlElement(  xmlRootElement.GetFirstChildElement("foo"));  if (!xmlElement.IsGood())  {  std::cerr << "Could not read the element from the file."  << std::endl;  exit(EXIT\_FAILURE);  }  std::cout << "Element name: " << xmlElement.GetElementName()  << std::endl;  std::string textValue("");  if (!xmlElement.GetAttribute("string", &textValue))  {  std::cerr << "Could not fetch an attribute from the file."  << std::endl;  exit(EXIT\_FAILURE);  }  std::cout << "std::string (text in between tags): "  << textValue << std::endl;  uge::XMLElement foobarElement(xmlRootElement.GetFirstChildElement("foo"));  foobarElement = foobarElement.GetFirstChildElement(  "bar").GetFirstChildElement("foobar");  std::string value("");  foobarElement.GetElementAsText(value);  std::cout << "Element name: " << foobarElement.GetElementName()  << " - Element value: " << value << std::endl;  uge::XMLElement quux(  xmlRootElement.GetLastChildElement("quux"));  std::cout << "Element name: "  << quux.GetElementName() << std::endl;  std::string stringAttribute("");  quux.GetAttribute("string", &stringAttribute);  std::cout << "std::string: " << stringAttribute << std::endl;  std::wstring wstringAttribute(L"");  quux.GetAttribute(L"wstring", &wstringAttribute);  std::wcout << "std::wstring: " << wstringAttribute << std::endl;  int intAttribute = 0;  quux.GetIntAttribute("int", &intAttribute);  std::wcout << "int: " << intAttribute << std::endl;  unsigned int uintAttribute = 0;  quux.GetUnsignedIntAttribute("uint", &uintAttribute);  std::wcout << "uint: " << uintAttribute << std::endl;  float fFloatAttribute = 0.0f;  quux.GetFloatAttribute("float", &fFloatAttribute);  std::wcout << "float: " << fFloatAttribute << std::endl;  double fDoubleAttribute = 0.0;  quux.GetDoubleAttribute("double", &fDoubleAttribute);  std::wcout << "double: " << fDoubleAttribute << std::endl;  bool boolAttribute = false;  quux.GetBoolAttribute("boolean", &boolAttribute);  std::wcout << "bool: " << boolAttribute << std::endl;  xmlFile.CloseFile();  return 0;  } |

|  |
| --- |
| #include <Utilities/Debug/Logger.h>  #include <Utilities/File/ZipFile.h>  #include <iostream>  int main()  {  uge::ZipFile zipFile;  std::string fileName("ZipFile.zip");  if (!zipFile.Init(fileName))  {  std::cerr << "Error loading the zip file." << std::endl;  return -1;  }  std::vector<std::string> allFiles(zipFile.GetAllFileNames());  std::cout << "Files in " << fileName << ":";  for (const std::string& filename : allFiles)  {  std::cout << " " << filename;  }  std::cout << std::endl;  int totalFiles = allFiles.size();  for (int filePosition = 0;  filePosition < totalFiles; ++filePosition)  {  const std::string& filename = allFiles[filePosition];  std::cout << " " << filename << std::endl;  const int fileLength = zipFile.GetFileLength(filePosition);  if (fileLength > 0)  {  // Assuming it is a text file (ASCII).  char\* pBuffer = new char[fileLength + 1]; // +1: '\0'  if (zipFile.ReadFile(filePosition, pBuffer))  {  pBuffer[fileLength] = '\0';  std::cout << pBuffer;  }  SAFE\_DELETE\_ARRAY(pBuffer);  }  else  {  std::cout << "Directory or blank file.";  }  std::cout << std::endl;  }  zipFile.Destroy();  return 0;  } |

## Macros

## Math

## String

## System Information

|  |
| --- |
| #include <Utilities/System/SystemInformation.h>  int main()  {  // System information gathering  uge::SystemInformation systemInfo;  systemInfo.Init();  unsigned long long cpuSpeed = systemInfo.GetCPUSpeedInMHz();  unsigned int cpuCores = systemInfo.GetTotalCPUCores();  long long totalRAM = systemInfo.GetSystemRAMInMB();  long long freeRAM = systemInfo.GetAvailableSystemRAMInMB();  std::vector<std::string> diskNames = systemInfo.GetDiskNames();  std::vector<unsigned long long> diskFreeSpace =  systemInfo.GetFreeDiskSpaceInBytes();  systemInfo.Destroy();  return 0;  } |

|  |
| --- |
| #include <Utilities/System/Clock.h>  #include <Utilities/System/Time.h>  #include <iostream>  int main()  {  uge::Time::Clock clock;  clock.Init();  uge::Time::TimePoint startTime = uge::Time::GetTime();  std::this\_thread::sleep\_for(std::chrono::seconds(1));  uge::Time::TimePoint finalTime = uge::Time::GetTime();  std::cout << "Time difference:" << std::endl;  std::cout << "In seconds: " << uge::Time::GetDeltaAsSeconds(  finalTime, startTime) << " s." << std::endl;  std::cout << "In milliseconds: "  << uge::Time::GetDeltaAsMilliseconds(finalTime, startTime)  << " ms." << std::endl;  std::cout << "In microseconds: "  << uge::Time::GetDeltaAsMicroseconds(finalTime, startTime)  << " us." << std::endl;  std::cout << "In nanoseconds: "  << uge::Time::GetDeltaAsNanoseconds(finalTime, startTime)  << " ns." << std::endl;  std::cout << "Time elapsed since the clock creation: "  << clock.GetTimeElapsed() << "s." << std::endl;  clock.Init(); // Reset the clock.  std::cout << "Time elapsed since the clock was reinitialized: "  << clock.GetTimeElapsed() << "s." << std::endl;  return 0;  } |

## Templates

# UGE Tutorials

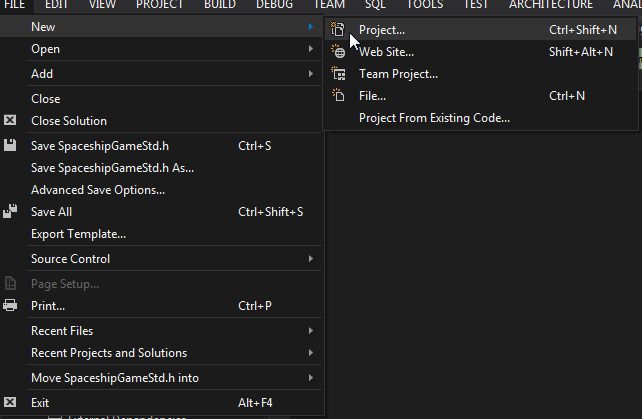
## Setting Up the Development Environment

### Microsoft Visual Studio 2012

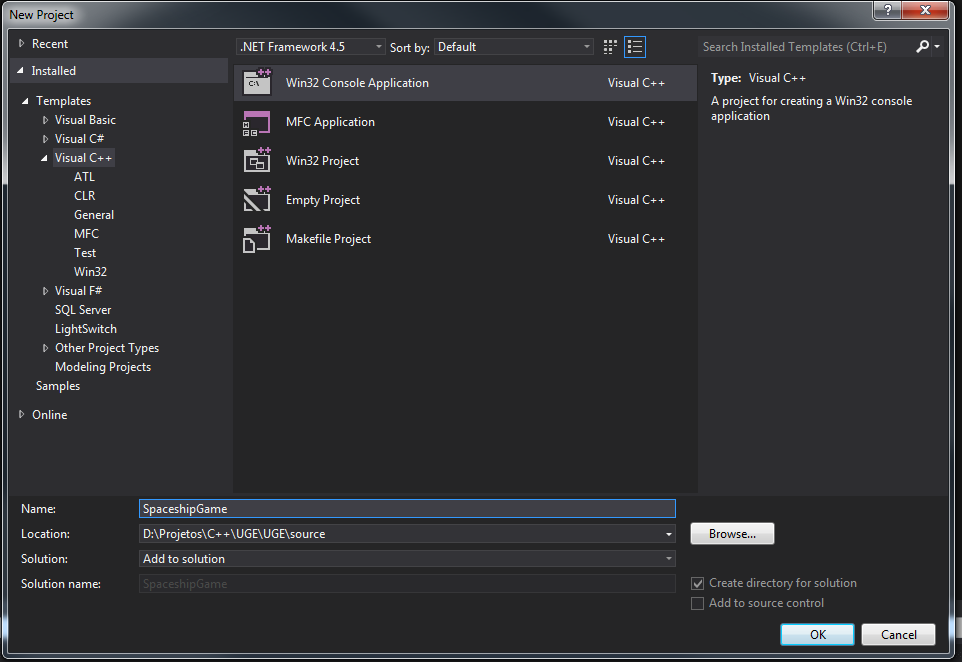
The following subsections describe how to set the development environment for Microsoft Visual Studio 2012. It provides an illustrated walkthrough to configure the settings and run a simple application using the Vector3 class. After configuring the environment, check Section 7.2 for a brief game tutorial.

#### Creating the Project

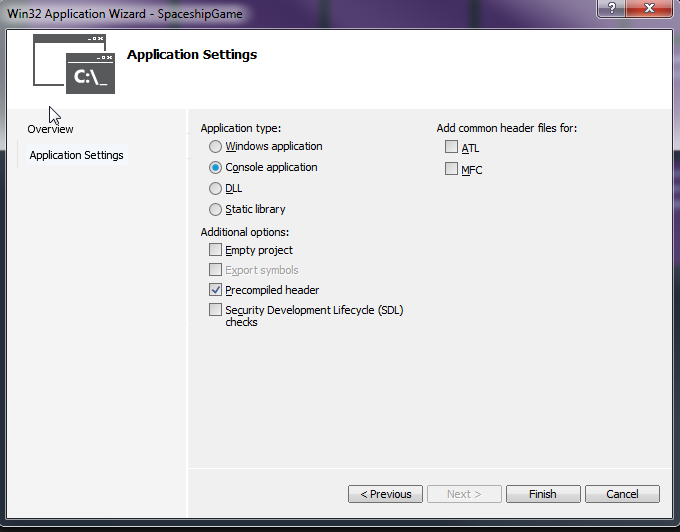
1. File / New / Project:



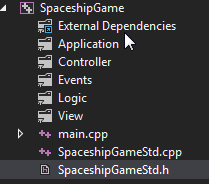
1. Visual C++ / Win32 Console Application:



1. Next / Finish:

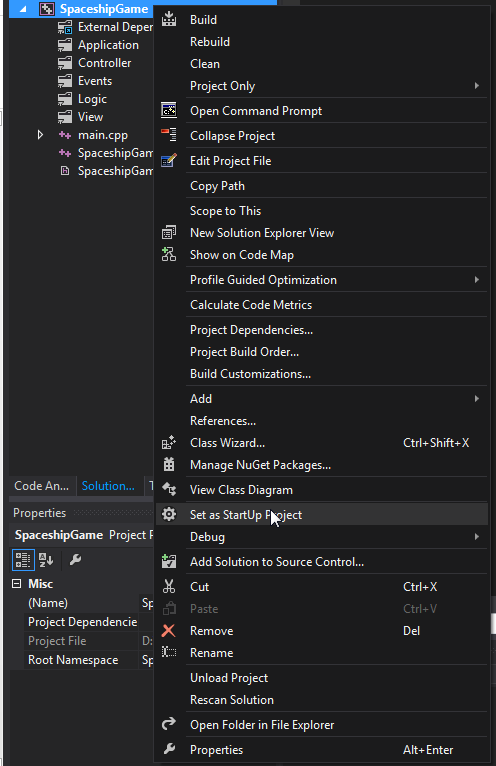


1. Project with filters (suggested – the filters match the directory structure):

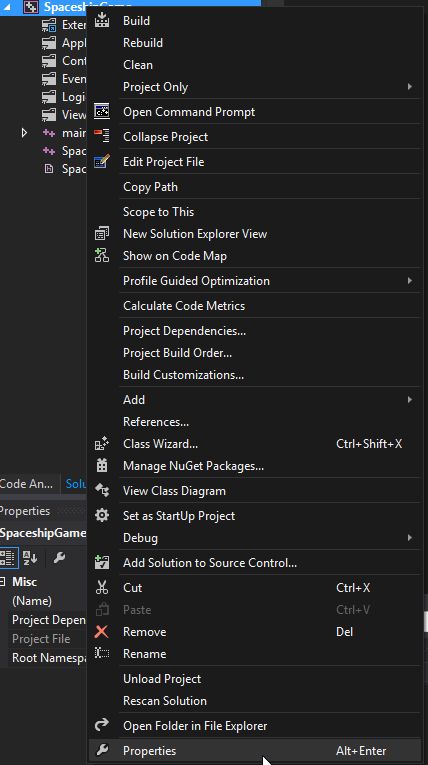


#### Setting the Project Configuration

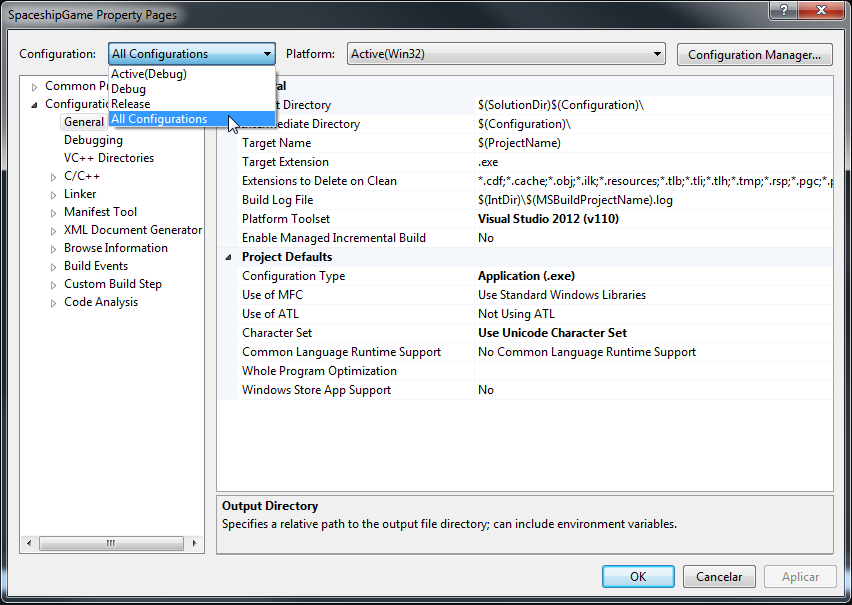
1. Project Name / Set as Startup Project:



1. Project Name / Properties:



1. Configuration: All Configurations:



1. Configuration Properties / General:
   1. Intermediate Directory:

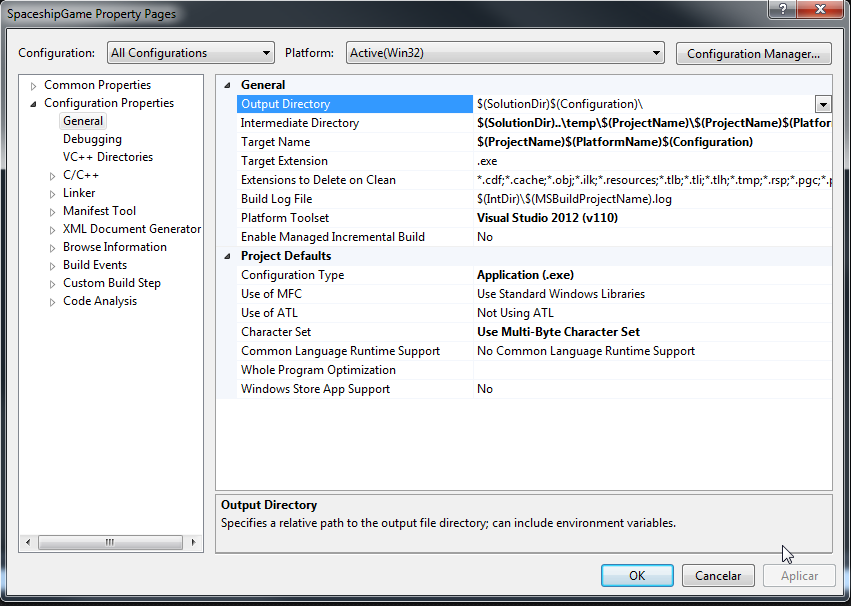
|  |
| --- |
| $(SolutionDir)..\temp\$(ProjectName)\$(ProjectName)$(PlatformName)$(Configuration) |

* 1. Target Name:

|  |
| --- |
| $(ProjectName)$(PlatformName)$(Configuration) |

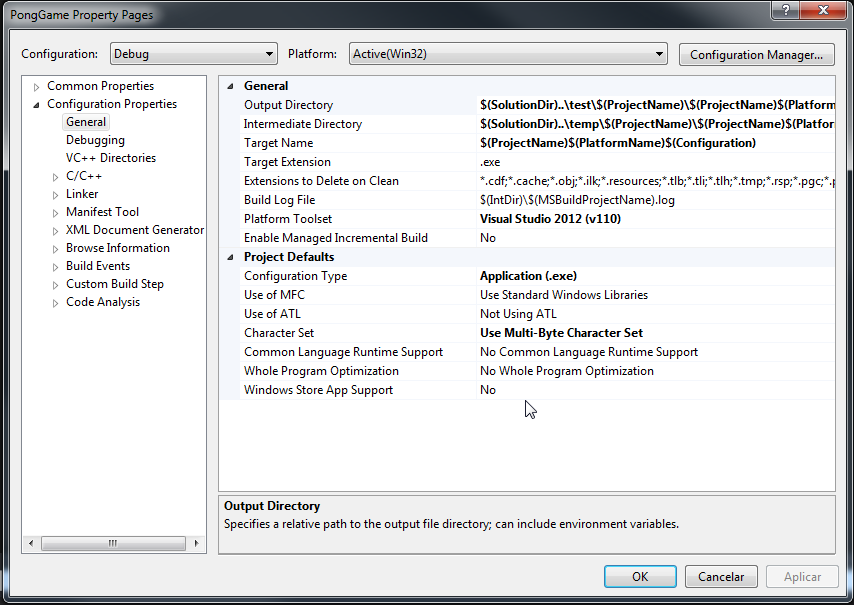
* 1. Character Set:

|  |
| --- |
| Use Multi-Byte Character Set |



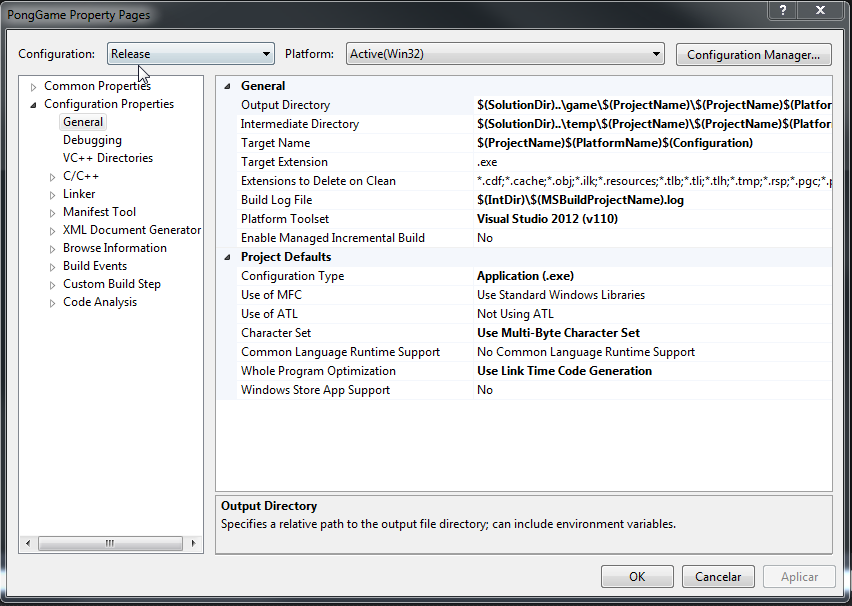
1. Configuration: Debug:
   1. Output Directory:

|  |
| --- |
| $(SolutionDir)..\test\$(ProjectName)\$(ProjectName)$(PlatformName)$(Configuration)\ |



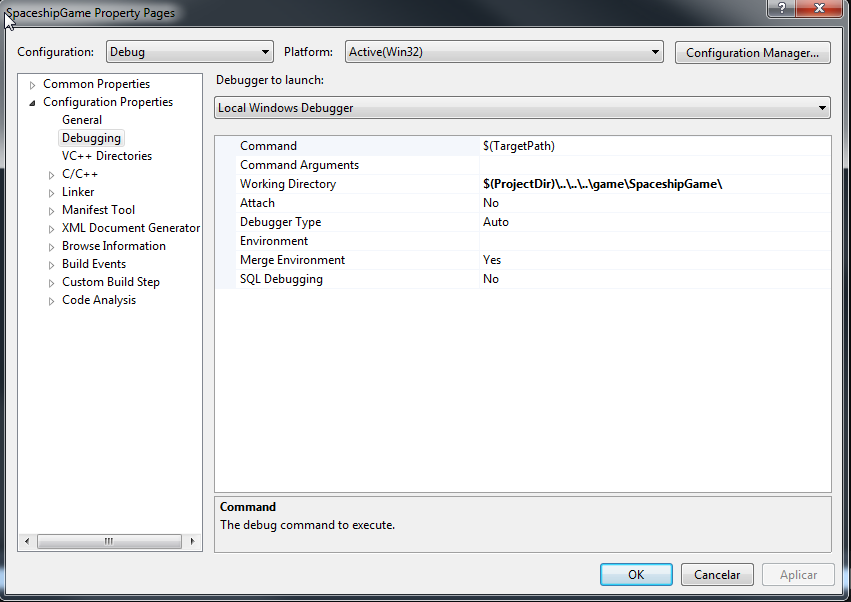
1. Configuration: Release:
   1. Output Directory:

|  |
| --- |
| $(SolutionDir)..\game\$(ProjectName)\$(ProjectName)$(PlatformName)$(Configuration)\ |



1. Configuration Properties / Debugging:
   1. Configuration: Debug:
      * Working Directory:

|  |
| --- |
| $(ProjectDir)\..\..\..\game\$(ProjectName)\ |



* 1. Configuration: Release:
     + Working Directory:

|  |
| --- |
| $(OutDir)\..\..\ |

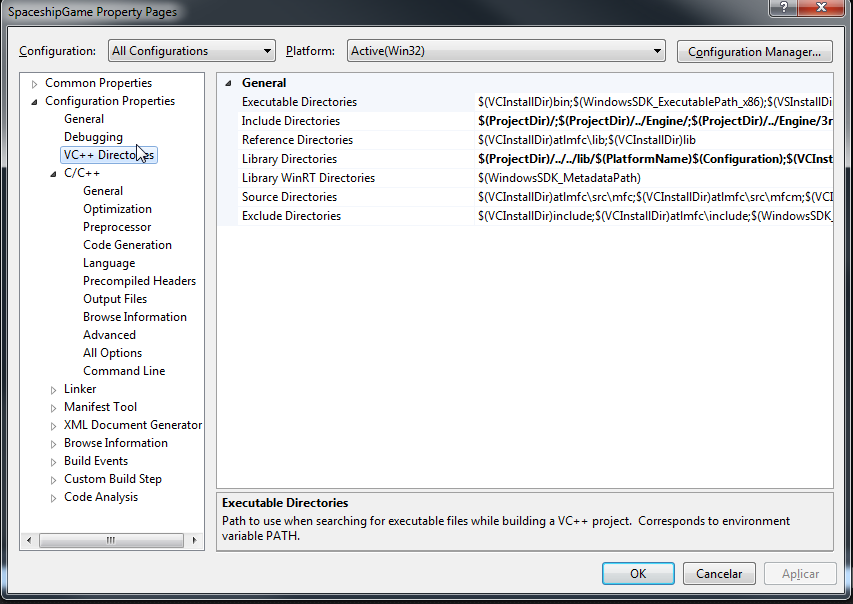


1. VC++ Directories:
   1. Configuration: All Configurations:
      * Include Directories:

|  |
| --- |
| $(ProjectDir)/;$(ProjectDir)/../Engine/;$(ProjectDir)/../Engine/3rd Party/boost/;$(ProjectDir)/../Engine/3rd Party/miniz/;$(ProjectDir)/../Engine/3rd Party/physfs/;$(ProjectDir)/../Engine/3rd Party/FastDelegate/;$(ProjectDir)/../Engine/3rd Party/bullet/;$(ProjectDir)/../Engine/3rd Party/ogre/include/;$(ProjectDir)/../Engine/3rd Party/ogre/include/OGRE/;$(ProjectDir)/../Engine/3rd Party/ogre/include/OGRE/Overlay/;$(ProjectDir)/../Engine/3rd Party/ogre/include/OIS/;$(ProjectDir)/../Engine/3rd Party/ogreoggsound/include/;$(ProjectDir)/../Engine/3rd Party/openal-soft/include/AL/;$(ProjectDir)/../Engine/3rd Party/openal-soft/include/;$(ProjectDir)/../Engine/3rd Party/TinyOAL/tinyoal-read-only/include/;$(ProjectDir)/../Engine/3rd Party/glm/;$(ProjectDir)/../Engine/3rd Party/hyperic-sigar/sigar-bin/include/;$(ProjectDir)/../Engine/3rd Party/OpenGL/glfw-2.7.6/include/;$(ProjectDir)/../Engine/3rd Party/OpenGL/glew-1.9.0/include/;$(ProjectDir)/../Engine/3rd Party/tinyxml/;$(ProjectDir)/../Engine/3rd Party/LuaPlus/Src/;$(ProjectDir)/../Engine/3rd Party/yse/include/;$(IncludePath) |

* + - Library Directories:

|  |
| --- |
| $(ProjectDir)/../../lib/$(PlatformName)$(Configuration);$(VCInstallDir)lib;$(VCInstallDir)atlmfc\lib;$(WindowsSDK\_LibraryPath\_x86); |

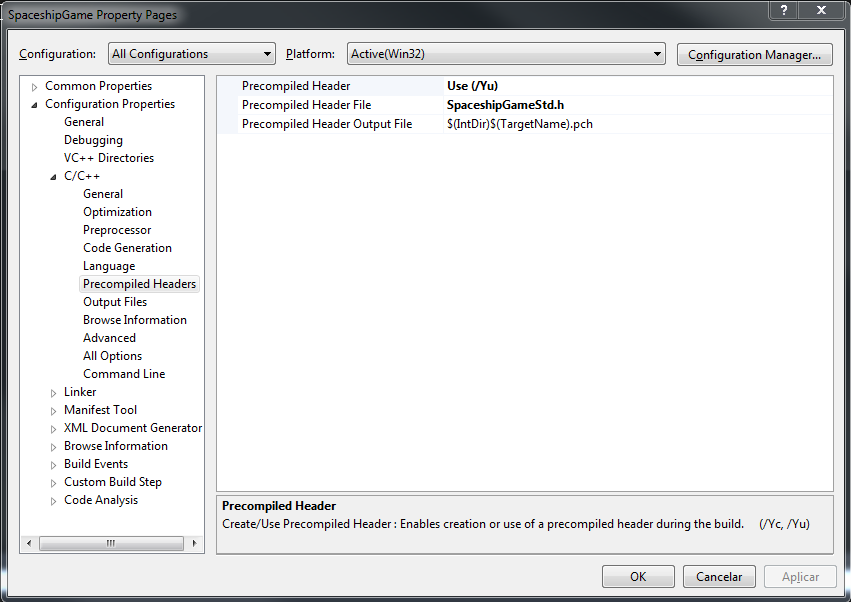


1. Configuration Properties / C/C++ / Precompiled Headers:
   1. Configuration: All Configurations:
      * Precompiled Header:

|  |
| --- |
| Use (/Yu) |

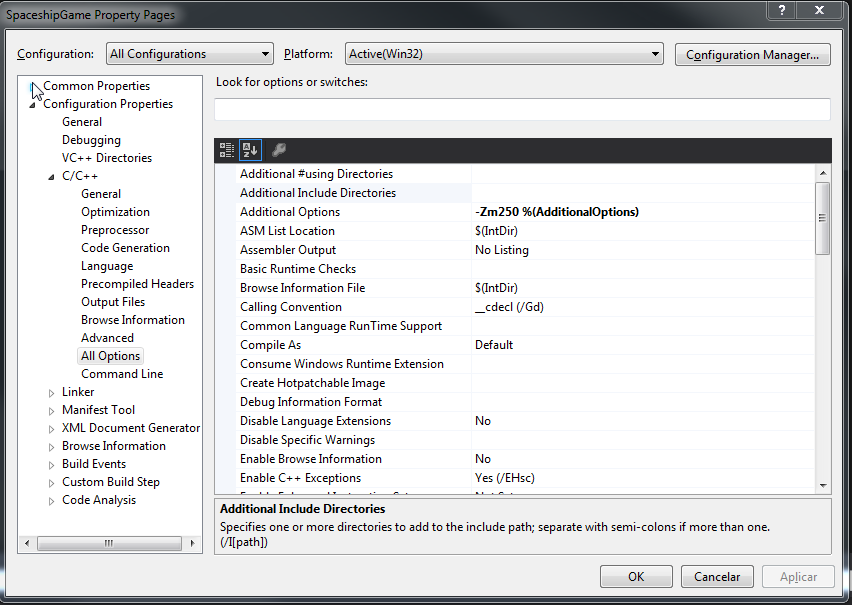
* + - Precompiled Header File (only needed if changing the name):

|  |
| --- |
| DesiredName.h |



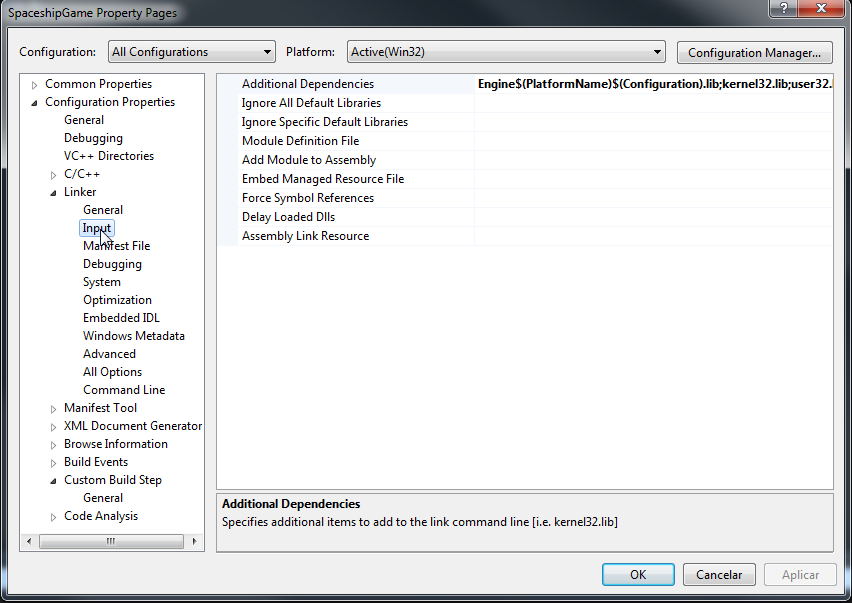
1. Configuration Properties / C/C++ / All Options:
   1. Configuration: All Configurations:
      * Additional Options:

|  |
| --- |
| -Zm250 %(AdditionalOptions) |



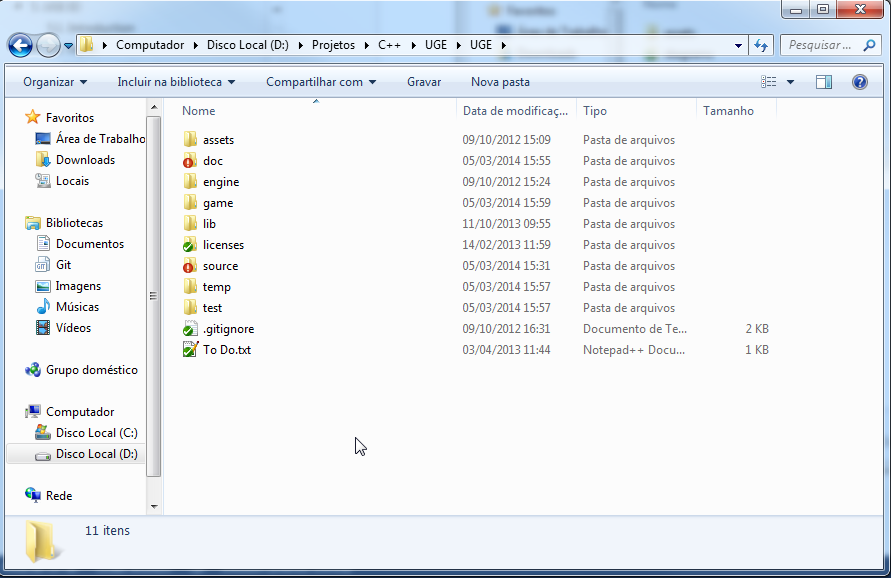
1. Configuration Properties / Linker / Input:
   1. Configuration: All Configurations
      * Additional Dependencies:

|  |
| --- |
| Engine$(PlatformName)$(Configuration).lib;kernel32.lib;user32.lib;gdi32.lib;winspool.lib;comdlg32.lib;advapi32.lib;shell32.lib;ole32.lib;oleaut32.lib;uuid.lib;odbc32.lib;odbccp32.lib;%(AdditionalDependencies) |

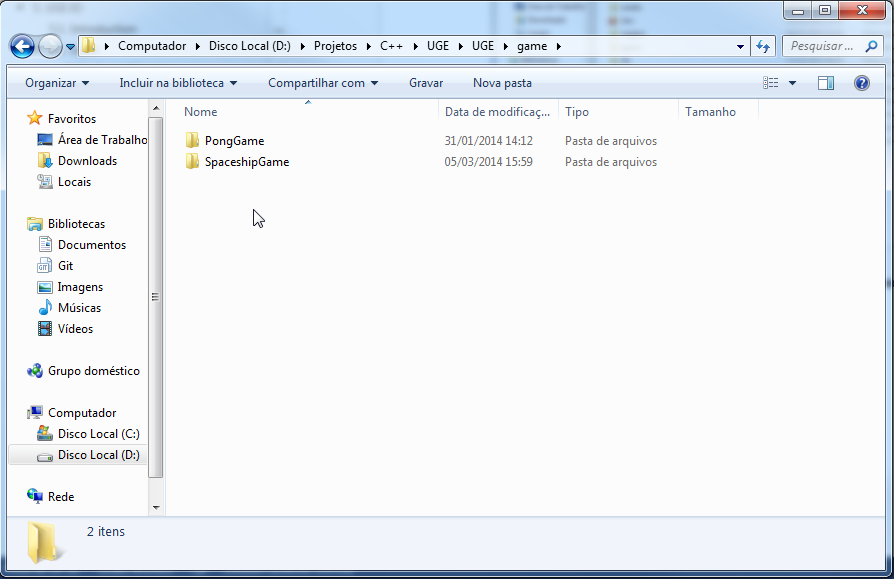


#### Setting the Game Directories

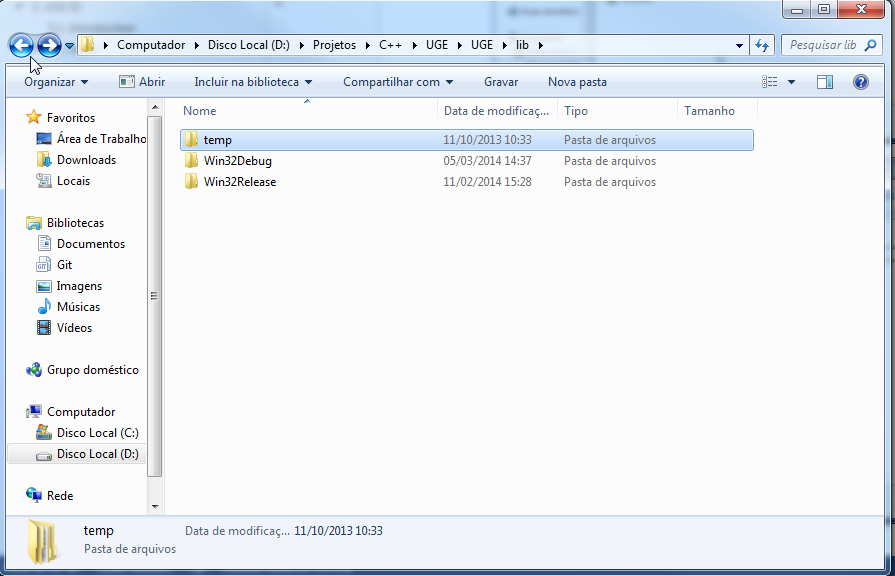
1. Root: the main directory of the project, also containing UGE code.



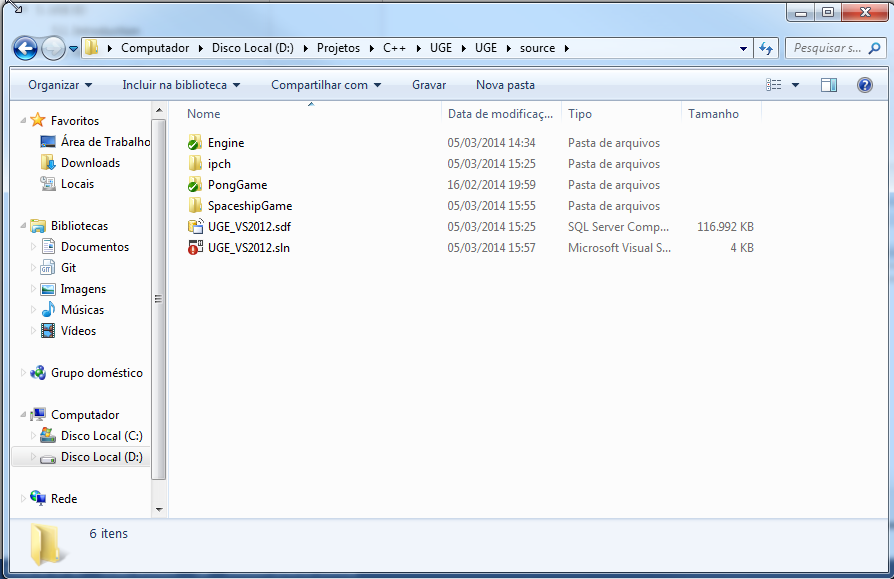
1. Game: the directory for the compiled game. Game assets should be stored here.



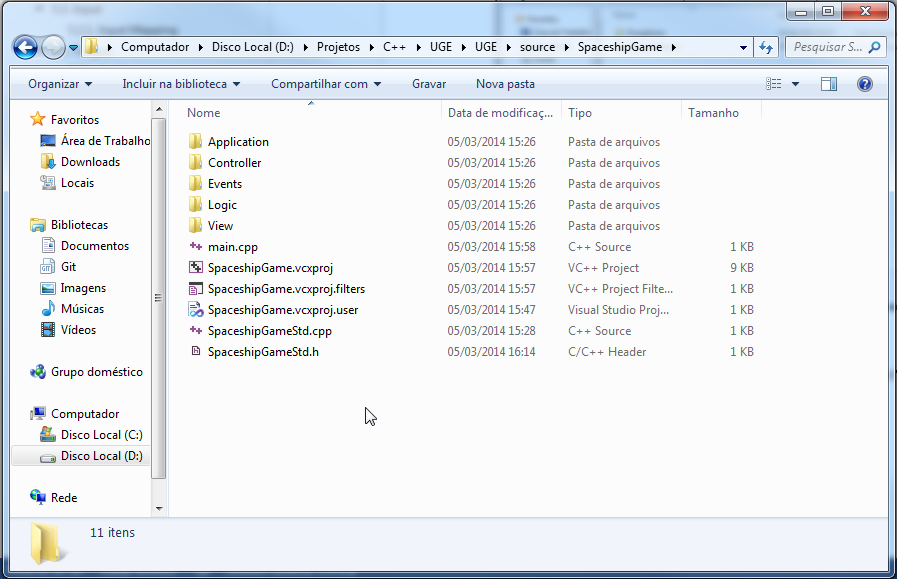
1. Lib: the libraries (dependencies) which will be linked to the project.



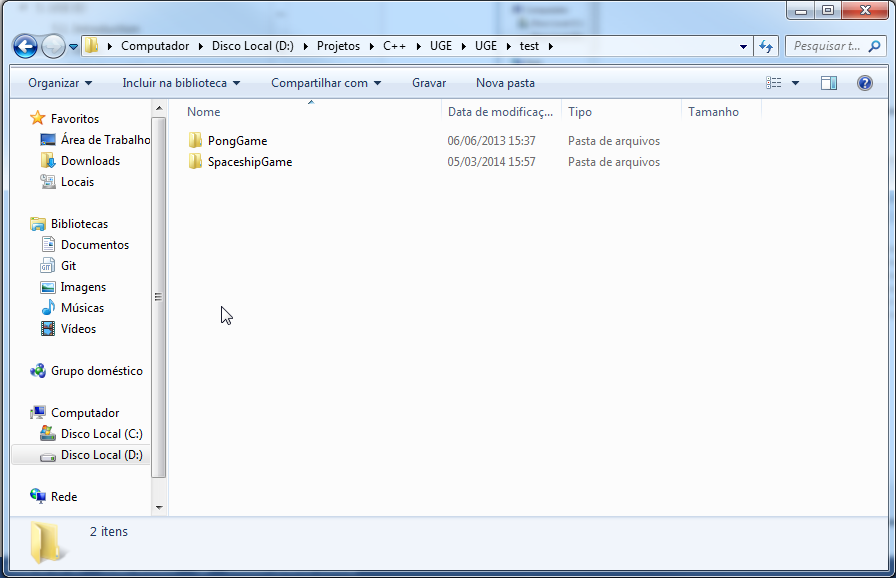
1. Source: the source code of the game.



1. Game Project Directory (suggest):



1. Test: the debug build of the game will be saved here.



#### Setting the Includes

1. Precompiled header (useful for dependencies):

|  |
| --- |
| #include "../Engine/GameEngineStd.h" |

1. Game layers, controller and events:
   1. Game Application layer:

|  |
| --- |
| #include <Engine/GameApplication/BaseGameApplication.h> |

* 1. Game Logic layer:

|  |
| --- |
| #include <Engine/GameLogic/BaseGameLogic.h> |

* 1. Game View layer:

|  |
| --- |
| #include <Engine/GameView/GameView.h> |

* 1. Controller:

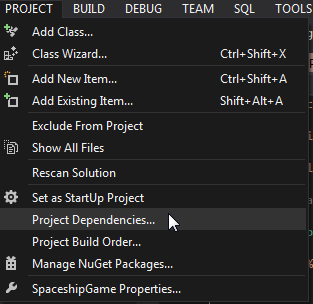
|  |
| --- |
| #include <Engine/GameController/GameController.h> |

* 1. Events:

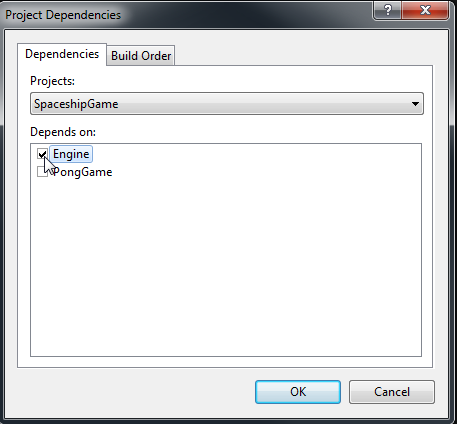
|  |
| --- |
| #include <Core/Events/IEventManager.h> |

#### Setting the Project Dependencies

1. Project /



1. Select the project name and mark Engine as its dependence.

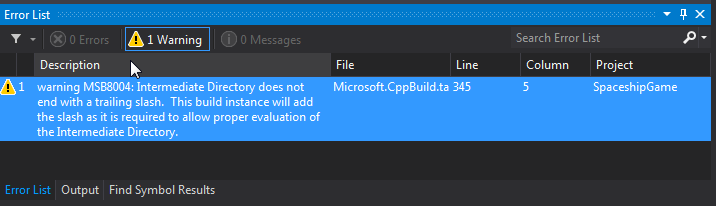


#### Testing the Development Environment

1. In main.cpp, add the following code:

|  |
| --- |
| #include "SpaceshipGameStd.h"  #include <stdio.h>  #include <Utilities/Math/Vector.h>  int main(int argc, char\* argv[])  {  uge::Vector3 vec(1.0f, 2.0f, 3.0f);  printf("(%f, %f, %f)\n", vec.x, vec.y, vec.z);  return 0;  } |

1. Compile the code:
   1. This should be the build output:



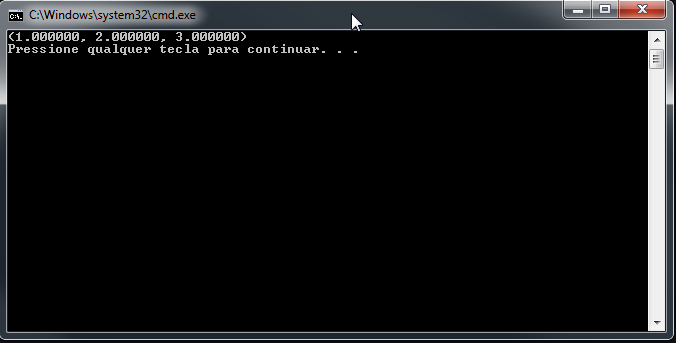
* 1. If there is the error:

|  |
| --- |
| error LNK1104: cannot open file 'EngineWin32Debug.lib' |

This usually happens when the Project Dependencies were not set (c.f. Section 7.1.1.5) and Visual Studio does not find the library as it was not compiled before. Fix the settings and compile again;

* 1. If there are hundreds or thousands of warning regarding linking, it is safe to disregard them. This will be fixed in the future (they seem to be linked twice, so the second is ignored).

1. Run the project (Ctrl + F5 to pause after running). This should be the output:



## Creating a Game

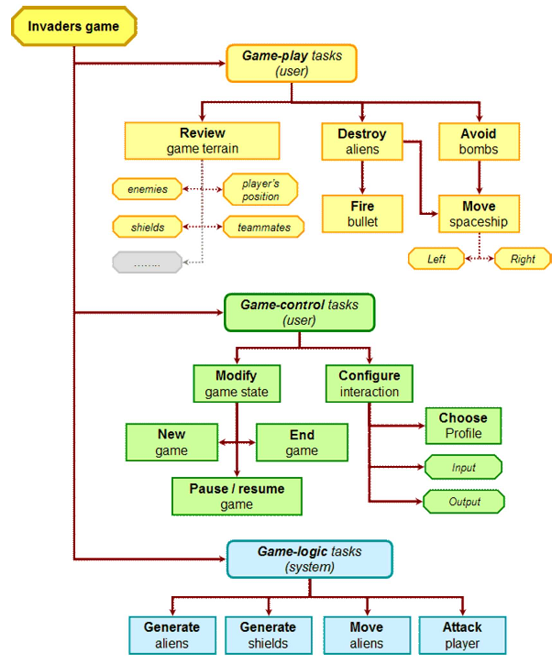
This section describes the implementation of a simple Spaceship game prototype to illustrate UGE features. This game is similar to Grammenos et al. Access Invaders game [7] and use his Unified Design tasks to illustrate how UGE helps the development of UA-Games.

It is important to note the goal of this section is not to create a fully accessible game. Rather, the objective is to outline how to use UGE’s features to prototype and implement the game. In order to achieve this, this section shows how the engine’s features contribute to improve the flexibility of the implementation and how it is possible to reuse existing game logic code to define a new player profile for a different interaction ability.

At the end of this section, it is possible to tweak the resulting prototype and make an accessible game for the desired public.

### Unified Design

**Figure 26** illustrates the Unified Design for a Space Invaders clone. It was extracted from Grammenos et al. paper “Unified design of universally accessible games” [8][[12]](#footnote-12). This game was chosen due to its simple design to illustrate how to use UGE’s layers and core features to implement an accessible game design (in this case, resulting of the Unified Design framework).



**Figure 26.** The Unified Design for a Space Invaders clone (extracted from [8]).

Like in the original game, the goal of the player is to eliminate the invading aliens’ spaceships before they destroy the world. The player can move in predefined directions (left and right, in this design) and should fire to destroy the aliens enemies. The remaining subsections will focus on how to implement the game using UGE.

### Implementation: Initial Header

The initial code is available to download at <>.

#### Game Application Layer

This is the initial source code for the tutorial game’s GameApplication layer (**Listing 44**).

**Listing 44.** The Spaceship game GameApplication.

|  |
| --- |
| #pragma once  #include <Engine/GameApplication/BaseGameApplication.h>  #include <IO/Output/Audio/Implementation/OpenALSoft/OpenALSoftAudio.h>  #include <IO/Output/Graphics/Implementation/Ogre3D/OgreGraphics.h>  #include "../Logic/GameLogic.h"  #include "../View/HumanView.h"  namespace sg  {  class Application : public uge::BaseGameApplication  {  public:  Application();  ~Application();  virtual bool vInit() override;  virtual uge::BaseGameLogic\* vCreateGameLogic() override;  virtual std::wstring vGetGameTitle() const override;  virtual bool vInitOutputSystems();  uge::IGameViewSharedPointer CreateGameView();  private:  uge::PlayerProfile m\_CurrentPlayerProfile;  };  } |

Section 7.2.4 describes its implementation.

#### Game Logic

This is the initial source code for the tutorial game’s GameLogic layer (**Listing 45**).

**Listing 45.** The Spaceship game GameLogic.

|  |
| --- |
| #pragma once  #include <Core/EntityComponent/Component/Implementation/BulletPhysicsComponent.h>  #include <Core/Events/DefaultEvents.h>  #include <Core/Physics/Implementation/BulletPhysics/BulletPhysics.h>  #include <Core/Physics/Implementation/NullPhysics/NullPhysics.h>  #include <Engine/GameLogic/BaseGameLogic.h>  #include "GameStateFactory.h"  namespace sg  {  class GameLogic : public uge::BaseGameLogic  {  public:  GameLogic();  ~GameLogic();  virtual uge::GameStateFactory\*  vCreateGameStateFactory() override;  virtual uge::IPhysics\* vCreatePhysics() override;  void vRegisterGameDelegates() override;  void vUnregisterGameDelegates() override;  protected:  void ActorCreatedDelegate(  uge::IEventDataSharedPointer pEventData);  private:  };  } |

Section 7.2.5 describes its implementation.

#### Game View

This is the initial source code for the tutorial game’s GameView layer (**Listing 46**).

**Listing 46.** The Spaceship game GameView.

|  |
| --- |
| #pragma once  #include <Core/Events/DefaultEvents.h>  #include <Core/Events/IEventManager.h>  #include <Core/PlayerProfile/PlayerProfiles.h>  #include <Core/Scene/Implementation/Ogre3D/OgreSceneNodeRenderer.h>  #include <Core/Scene/Implementation/Ogre3D/OgreSceneRenderer.h>  #include <Core/Scene/Implementation/OpenALSoft/OpenALSoftSceneNodeRenderer.h>  #include <Core/Scene/Implementation/OpenALSoft/OpenALSoftSceneRenderer.h>  #include <Engine/GameView/GameView.h>  #include "../Controller/GameController.h"  namespace sg  {  class HumanView : public uge::HumanGameView  {  public:  HumanView(uge::IGraphicsSharedPointer pGraphics,  uge::IAudioSharedPointer pAudio,  uge::ResourceCache& resourceCache,  const uge::PlayerProfile& playerProfile);  ~HumanView();  uge::ISceneRendererSharedPointer GetPhysicsDebugRenderer();  protected:  virtual bool vInit(uge::IScene\* pScene) override;  virtual bool vDestroy() override;  virtual uge::ICameraNodeSharedPointer vCreateCamera() override;  virtual uge::GameControllerSharedPointer  vCreateController() override;  virtual void vSetControlledActor(uge::ActorID actorID,  bool bSetCameraTarget = false) override;  private:  void RegisterEventDelegates();  void UnregisterEventDelegates();  void ControlledActorDelegate(  uge::IEventDataSharedPointer pEventData);  private:  uge::IGraphicsSharedPointer m\_pGraphics;  uge::IAudioSharedPointer m\_pAudio;  uge::ResourceCache& m\_ResourceCache;  uge::SceneRendererID m\_AuralRendererID;  uge::SceneRendererID m\_GraphicalRendererID;  uge::PlayerProfile m\_PlayerProfile;  };  } |

Section 7.2.6 describes its implementation.

### Main

The main function of the program is simple (**Listing 47**): it just creates and run the GameApplication. The GameApplication, in turn, is responsible for managing the run-time of the GameLogic and the GameView(s).

**Listing 47.** The entry point of the tutorial game.

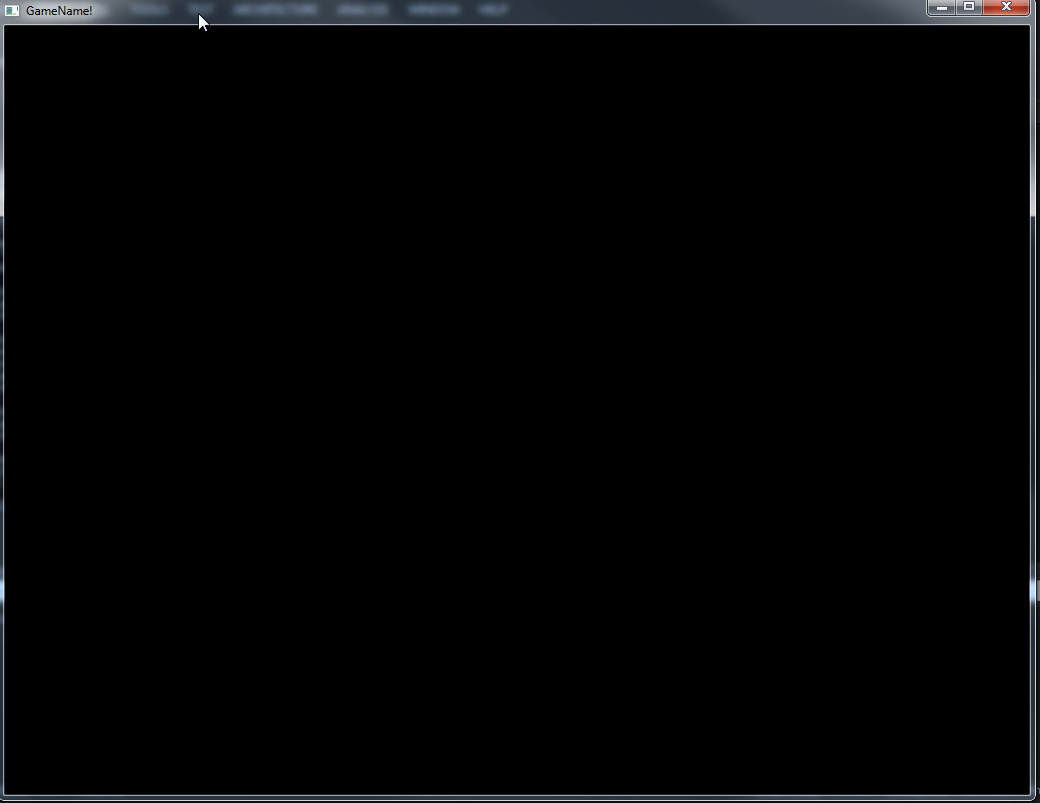
|  |
| --- |
| #include "SpaceshipGameStd.h"  #define UGE\_ENABLE\_PHYSICS 1  #define UGE\_DEBUG\_PHYSICS 1  #define SG\_USE\_DEVELOPMENT\_RESOURCE\_FILE 1  // Game specific headers  #include "Application/Application.h"  // UGE headers.  #include <Utilities/Debug/Logger.h>  int main()  {  uge::debug::log::Init("data/debug/LogConfig.xml");  sg::Application game;  game.vInit();  game.vRun();  game.vDestroy();  uge::debug::log::Destroy();  return 0;  } |

The first two #define directives are flags for enabling or disabling the game physics and physics debug. This is useful to check the game world simulation – in fact, the entire game logic runs without output data, thus it is possible to create the game using only the physics debugger world representation.

The third #define will be used to set a custom ResourceCache in Section 7.2.4.6. The ResourceCache provides a useful interface for managing game resources and assets – such as XML configuration files, scripting files and game art.

To disable the #defines, it is necessary to either comment or remove them. It is important to note they should be before the #include directive – otherwise they will fail (and the compiler will assume they are disabled), as they are pre-compilation directives.

Running the game so far will present a blank window – the game world is empty (**Figure 27**).



**Figure 27.** Checking the game Application: the initial window.

The following sections describe the implementation of the three game layers, starting with the GameApplication (Application, in this project), going through the GameLogic and finishing with the GameView (HumanView). All the layers use the default layer implementations (BaseGameApplication, BaseGameLogic and HumanGameView, respectively).

### Game Application Layer

Using the available subsystems and default BaseGameApplication, it is simple to implement the Game Application layer. In this layer, it is necessary to setup and initialize the game IO subsystems, load the initial player profile and create the Game Logic and View layers. More information about the Game Application Layer is available at Section 3.2. The header for the Application class can be located in **Listing 44**.

#### Creating an Initial Player Profile

As it is necessary to know how to convey the game information to user, the PlayerProfile is among the very first structures to be loaded (*c.f.* **Listing 1**). During the initialization, the BaseGameApplication’s method vInitPlayerProfiles() loads a list of pre-defined profiles. The default path for this file is "data/config/player\_profiles/player\_profiles.xml". It is possible to change this path by overriding the method.

Let us create the first item in the player profile list – for instance, a default profile for average users (**Listing 48**).

**Listing 48.** An initial list of player profiles.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfiles resource="data/config/player\_profiles/player\_profiles.xml">  <PlayerProfile name="Average User: Default"  resource="data/config/player\_profiles/average\_user/profile.xml"/>  </PlayerProfiles> |

This item refers to a resource containing the profile description (resource) and an identifier (name). The name is the key used to load the profile during run-time. Thus, if the developers wish to add a new profile (which this tutorial will cover later), it is just necessary to add a new item list and to create its profile. It is possible to define profiles for a single user or for a group of users (**Listing 49**).

**Listing 49.** Several player profiles in the list.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfiles resource="data/config/player\_profiles/player\_profiles.xml">  <PlayerProfile name="Average User: Default"  resource="data/config/player\_profiles/average\_user/profile.xml"/>  <PlayerProfile name="Visually Impaired: Blind"  resource="data/config/player\_profiles/blind/profile.xml"/>  <PlayerProfile name="Mary’s Profile"  resource="data/config/player\_profiles/mary/profile.xml"/>  </PlayerProfiles> |

After the definition of the list, it is necessary to create the profile itself (**Listing 50**).

**Listing 50.** The player profile for average users.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerProfile name="Average User: Default"  resource="data/config/player\_profiles/average\_user/profile.xml">  <GeneralSettings>  <Preferences resource=  "data/config/player\_profiles/average\_user/preferences.xml"/>  </GeneralSettings>  <InputSettings>  <!--<Device type="keyboard"/>-->  <Mapping resource=  "data/config/player\_profiles/average\_user/input\_mapping.xml"/>  </InputSettings>  <OutputSettings>  <PrimaryOutput type="graphical"  resource=  "data/config/player\_profiles/average\_user/graphics.xml"  events=  "data/config/player\_profiles/average\_user/graphical\_events.xml"/>  <SecondaryOutput type="aural"  resource=  "data/config/player\_profiles/average\_user/audio.xml"  events=  "data/config/player\_profiles/average\_user/aural\_events.xml"/>  </OutputSettings>  <GameplaySettings>  <EntitySpecialization resource=  "data/config/player\_profiles/average\_user/entity/entities.xml"/>  <EventSpecializations resource=  "data/config/player\_profiles/average\_user/events/events.xml"/>  </GameplaySettings>  </PlayerProfile> |

Each element of the profile defines a way to tailor the profile – ranging from IO devices to game preferences to game logic customizations. The definition of each element is available at the suitable section (OutputSettings in Section 7.2.4.2, GameplaySettings in Sections 7.2.5.8 and 7.2.6.8, InputSettings in Section 7.2.6.6 and, finally, GeneralSettings in Section 7.2.4.2).

#### Creating and Initializing the Output Subsystems

After knowing the desired player profile, it is time to create and initialize the output subsystems. The OutputSettings of the PlayerProfile offers the following information for the IGraphics (**Listing 51**) and IAudio (**Listing 52**) subsystems:

**Listing 51.** The settings for the graphical renderer.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Graphics resource="data/config/player\_profiles/average\_user/graphics.xml">  <Window width="1024" height="768" pixel\_depth="32" vsync="false" fullscreen="false" />  <Renderer name="OpenGL Rendering Subsystem"/>  <!-- Other options:  Direct3D9 Rendering Subsystem  Direct3D10 Rendering Subsystem  Direct3D11 Rendering Subsystem  OpenGL Rendering Subsystem  OpenGL ES 1.x Rendering Subsystem  -->  <Text>  <Font resource="font\_name" size="10" />  <Subtitles enabled="false" />  </Text>  </Graphics> |

**Listing 52.** The settings for audio playback.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Audio resource="data/config/player\_profiles/average\_user/audio.xml">  <Settings type="stereo" />  <Music enabled="true" volume="1.0f" />  <SFX enabled="true" volume="1.0f" />  <Speech enabled="true" volume="1.0f" />  </Audio> |

With this information, it is possible to create the output subsystems. This is the goal of the vInitOutputSystems() method (**Listing 53**).

**Listing 53.** Creating the output subsystems for the GameApplication.

|  |
| --- |
| virtual bool vInitOutputSystems()  {  m\_PlayerProfiles.SetCurrentProfile("Average User: Default");  m\_CurrentPlayerProfile = m\_PlayerProfiles.GetCurrentProfile();  // Graphics  uge::IGraphicsSharedPointer pGraphics(  LIB\_NEW uge::OgreGraphics(vGetGameTitle(),  m\_CurrentPlayerProfile.GetGraphicalPreferences()));  // Audio  const int TOTAL\_BUFFERS = 32;  uge::IAudioSharedPointer pAudio(  LIB\_NEW uge::OpenALSoftAudio(TOTAL\_BUFFERS));  return m\_Output.Init(pGraphics,  pAudio);  } |

In this method, the developers should initialize the subsystems according to the profile specifications (for now, the tutorial sets the "Average User: Default" as the active profile). Currently, a graphical output subsystem is mandatory as it handles the creation of the window.

**Listing 53** uses the OGRE and OpenALSoft subsystems for outputting data. Those are good default options to start implementing the game.

#### Creating the Game Logic Layer

It is time to allocate the game logic that will control the fate of the game world. This is straightforward with the GameLogic’s method vCreateGameLogic (**Listing 54**):

**Listing 54.** Creating the game logic.

|  |
| --- |
| virtual uge::BaseGameLogic\* vCreateGameLogic() override  {  uge::BaseGameLogic\* pGameLogic;  pGameLogic = LIB\_NEW sg::GameLogic;  return pGameLogic;  } |

It is important not to initialize the GameLogic yet – it is just constructed. The BaseGameLogic will perform the initialization automatically after the call to the method.

#### Creating the Game View Layer

So far, the GameApplication has defined its player profiles, IO subsystems and the Game Logic layer. There is still no game presentation, though – thus, it is necessary to create a game view.

Depending on the developers goals, creating the Game View Layer can be as simple as creating the Game Logic layer or much more complex. This depends on the implementation of HumanView – for instance, will there be many game views (one for each profile) or a single one, which tailor itself? Both approaches are valid and have its advantages. For simplicity, this tutorial assumes the usage of many game views – it will start implementing one then add others to specialize the game implementation.

As this is the first view, the CreateGameView() method (**Listing 55**) just instantiates the HumanView class.

**Listing 55.** Creating the game view.

|  |
| --- |
| uge::IGameViewSharedPointer CreateGameView()  {  uge::IGameViewSharedPointer pGameView(  LIB\_NEW sg::HumanView(m\_Output.GetGraphics(),  m\_Output.GetAudio(),  m\_Resources.GetResourceCache(),  m\_PlayerProfiles.GetCurrentProfile()));  return pGameView;  } |

If there were additional game views, it would be necessary to use a conditional statement here to select the most appropriate IGameView implementation for the current player profile. This is the first non-virtual method of the Application – another possibility to initialize the game view is in the GameApplication method’s vInit().

#### Setting the Game Title

Finally, it is possible to change the window title with the game title with the vGetGameTitle() method (**Listing 56**).

**Listing 56.** Setting the game title.

|  |
| --- |
| virtual std::wstring vGetGameTitle() const override  {  return L"Spaceship!";  } |

This method returns a wide string (hence the 'L') – which should be encoded in UTF-16 on Windows and the native Unicode implementation on Linux platforms.

#### Initializing the Resource Cache

By default, the BaseGameApplication uses a ZIPFileResource to store all the game resources and assets. This reduces the total file size and helps to decrease loading times by exploring cache coherence and reducing file seek times in slow media drives (such as the hard disk drive).

Although this is useful for distribution, it might slow down the development, as it requires re-creating the ZIP file each time a resource is created or modified. To speed up the development process, there is another default IResourceFile implementation: the ZipFileDevelopmentResource. This implementation fakes out a ZIP file from the file system. This makes it much more convenient to develop the game – after the game is ready for distribution, the developers just need to create a ZIP file of the data directory and change the implementation.

**Listing 57** contains the code of the custom resource cache for the game.

**Listing 57.** Creating and initializing a custom resource cache.

|  |
| --- |
| bool vInitResourceCache() override  {  const std::string resourceFileName = "data/data.zip";  int resourceCacheSizeMB = 100; // 100MB  #if SG\_USE\_DEVELOPMENT\_RESOURCE\_FILE  uge::IResourceFile\* pResourceFile =  LIB\_NEW uge::ZipFileDevelopmentResource(resourceFileName,  "./", uge::ZipFileDevelopmentResource::Mode::Editor);  #else  uge::IResourceFile\* pResourceFile = LIB\_NEW  uge::ZipFileResource(resourceFileName);  #endif  return m\_Resources.Init(resourceCacheSizeMB, pResourceFile);  } |

It uses the #define created in **Listing 47**: when this preprocessor setting is defined, the resource cache uses a ZipFileDevelopmentResource; otherwise, it uses the default ZIPFileResource.

#### Initializing the Game

It is possible to further override the virtual methods and customize the game application. However, for a simple project, the previous sections are enough.

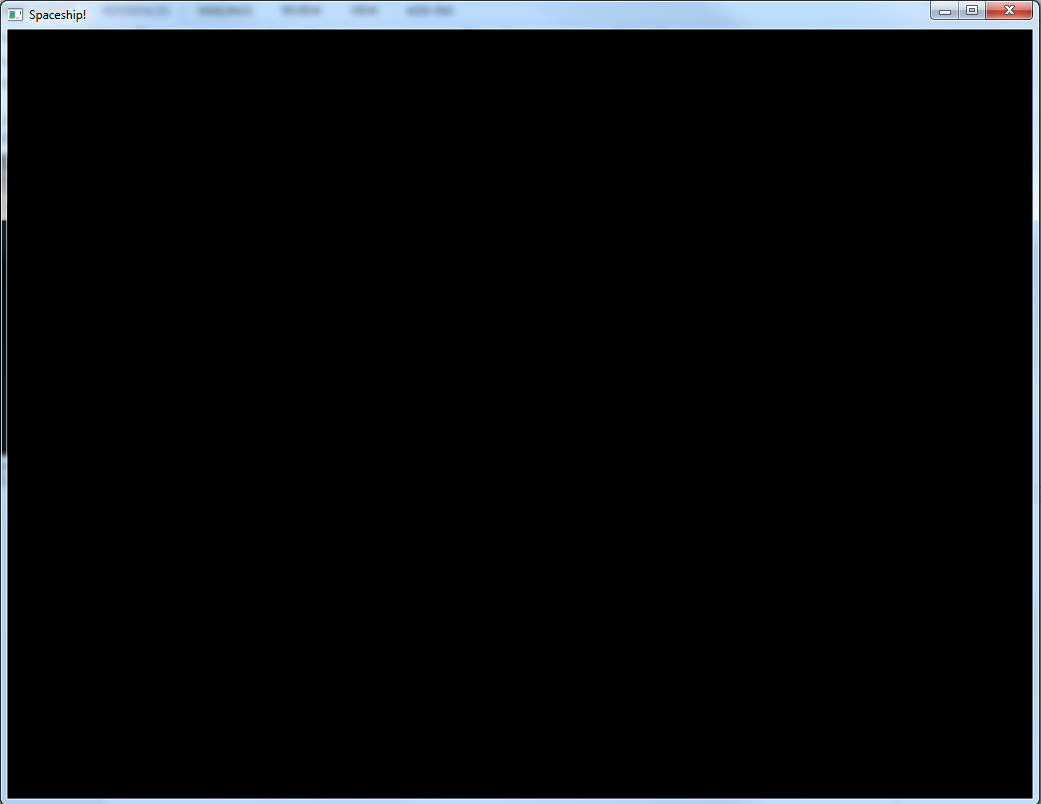
It is time to initialize the game application overriding the method vInit() (**Listing 58**). By default, this method initializes all the chosen game subsystems and the game logic.

**Listing 58.** Initializing the Application.

|  |
| --- |
| virtual bool vInit() override  {  if (!uge::BaseGameApplication::vInit())  {  return false;  }  uge::IGameViewSharedPointer pGameView = CreateGameView();  vAddGameView(pGameView);  #ifdef UGE\_DEBUG\_PHYSICS  std::shared\_ptr<sg::HumanView> pCastGameView =  std::dynamic\_pointer\_cast<sg::HumanView>(pGameView);  m\_pGameLogic->vEnablePhysicsDebug(  pCastGameView->GetPhysicsDebugRenderer());  #endif  m\_Output.PostInit();  const uge::GameplayPreferences::GameplaySettings&  gameplaySettings = m\_CurrentPlayerProfile.  GetGameplayPreferences().GetGameplaySettings();  m\_pGameLogic->vSetPlayerProfileFileName(  gameplaySettings.entitySpecializationFileName);  return true;  } |

As stated in Section 7.2.4.4, the GameView should be created by the Application – hence the call to CreateGameView(). If the flag USE\_DEBUG\_PHYSICS was set before including the Application’s header in **Listing 47**, the Application enables the debugger for the Physics subsystem. This is useful for visualizing the game world without the output specialization, as discussed in next section.

So far, the only apparent change from the end of Section 7.2.3 is the game title (**Figure 28**).



**Figure 28.** The tutorial game window after setting the Game Application layer.

It is time to implement the Game Logic layer to start defining the game world.

### Game Logic Layer

As **Figure 28** suggests, the game world is still empty. This section will start implementing some features of the game designed in Section 7.2.1 step by step, showing the results of each step in the game world. This section operates in two different parts of the Game Logic layer: the implementation of the BaseGameLogic itself and on several IGameStates (although it will focus on the Running state).

#### Creating the Game Physics

Using the default Physics subsystem implementations, it is simple to define the game Physics (**Listing 59**).

**Listing 59.** Creating the game Physics.

|  |
| --- |
| virtual uge::IPhysics\* vCreatePhysics() override  {  // GAME PHYSICS  #ifdef UGE\_ENABLE\_PHYSICS  uge::IPhysics\* pPhysics = LIB\_NEW uge::BulletPhysics;  #else  uge::IPhysics\* pPhysics = LIB\_NEW uge::NullPhysics;  #endif  assert(pPhysics != nullptr);  bool bSuccess = pPhysics->vInit();  assert(bSuccess);  return pPhysics;  } |

As the game view will be defined in Section 7.2.6, this current section uses the Physics debugger to display the entities in the game world (Section 7.2.4.7). This is valid because the Game Logic layer defines the game world by itself, without requiring a view. In fact, the Game View layer’s implementation acts like an observer to the Game Logic simulation, showing, saying or describing to the player what is currently happening the game scene.

#### Creating the Game States and Game State Factory

Before populating the game world, it is useful to define at least the required game states (Initializing and Running). This tutorial game will define nine states (illustrated in **Figure 29**: Initializing, SplashScreen, MainMenu, Running, Paused, NewGame, GameOver and Exiting), although the remaining of this section will focus only on the Running state.



**Figure 29.** Finite state machine for the game states.

##### Creating a Few Game States

**Listing 60** presents the header for the game states of **Figure 29**.

**Listing 60.** The game states header.

|  |
| --- |
| #pragma once  #include <Core/EntityComponent/Component/TransformableComponent.h>  #include <Core/EntityComponent/Component/Implementation/BulletPhysicsComponent.h>  #include <Core/Events/IEventManager.h>  #include <Core/Events/DefaultEvents.h>  #include <Core/Physics/Implementation/BulletPhysics/BulletPhysics.h>  #include <Core/Physics/Implementation/NullPhysics/NullPhysics.h>  #include <Engine/GameLogic/BaseGameLogic.h>  #include <Engine/GameLogic/GameState/BaseGameState.h>  #include "../Events/GameEvents.h"  namespace sg  {  namespace GameState  {  class Initializing : public uge::GameState::Initializing  {  friend class uge::BaseGameLogic;  public:  Initializing();  virtual ~Initializing();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  };  class SplashScreen : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  SplashScreen();  virtual ~SplashScreen();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  class MainMenu : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  MainMenu();  virtual ~MainMenu();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  class NewGame : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  NewGame();  virtual ~NewGame();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  class Running : public uge::GameState::Running  {  friend class uge::BaseGameLogic;  public:  Running();  virtual ~Running();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  private:  void CollisionStarted(  uge::IEventDataSharedPointer pEventData);  void CollisionEnded(  uge::IEventDataSharedPointer pEventData);  };  class Paused : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  Paused();  virtual ~Paused();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  class GameOver : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  GameOver();  virtual ~GameOver();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  class Exiting : public uge::GameState::BaseGameState  {  friend class uge::BaseGameLogic;  public:  /// The name of the state.  static const char\* g\_Name;  Exiting();  virtual ~Exiting();  virtual bool vInit(  uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(  unsigned long timeElapsed) override;  virtual bool vOnRender(  unsigned long timeElapsed) override;  virtual const std::string vGetName() const override;  };  }  } |

A simple game state is illustrated in **Listing 61**.

**Listing 61.** The initialization game state.

|  |
| --- |
| namespace sg  {  namespace GameState  {  Initializing::Initializing()  {  }  Initializing::~Initializing()  {  }  bool Initializing::vInit(uge::BaseGameLogic\* pGameLogic)  {  uge::GameState::BaseGameState::vInit(pGameLogic);  return true;  }  bool Initializing::vTailorToProfile(  const std::string& xmlResourceFilename)  {  return true;  }  bool Initializing::vDestroy()  {  uge::GameState::BaseGameState::vDestroy();  return true;  }  bool Initializing::vOnUpdate(unsigned long timeElapsed)  {  bool bSuccess =  uge::GameState::BaseGameState::vOnUpdate(timeElapsed);  m\_pGameLogic->vChangeGameState(SplashScreen::g\_Name);  return bSuccess;  }  bool Initializing::vOnRender(unsigned long timeElapsed)  {  return true;  }  }  } |

All this sample state does is request a change to next game state (SplashScreen) in its vOnUpdate() method. The next states keep requesting state changes following **Figure 29** until reaching Running. The remaining of this tutorial will focus only on the Running states.

##### The Game State Factory

After defining some game states, it is necessary to register them into a GameStateFactory (**Listing 62**). This allows the creation of custom states for the game.

**Listing 62.** A GameStateFactory for the tutorial game.

|  |
| --- |
| namespace sg  {  namespace GameState  {  class GameStateFactory : public uge::GameStateFactory  {  public:  GameStateFactory()  {  }  ~GameStateFactory()  {  }  protected:  virtual void vInitFactory() override  {  m\_StateFactory.Register<GameState::Initializing>(  GameState::Initializing::g\_Name);  m\_StateFactory.Register<GameState::SplashScreen>(  GameState::SplashScreen::g\_Name);  m\_StateFactory.Register<GameState::Paused>(  GameState::Paused::g\_Name);  m\_StateFactory.Register<GameState::MainMenu>(  GameState::MainMenu::g\_Name);  m\_StateFactory.Register<GameState::NewGame>(  GameState::NewGame::g\_Name);  m\_StateFactory.Register<GameState::Running>(  GameState::Running::g\_Name);  m\_StateFactory.Register<GameState::GameOver>(  GameState::GameOver::g\_Name);  m\_StateFactory.Register<GameState::Exiting>(  GameState::Exiting::g\_Name);  }  };  }  } |

The GameStateFactory create game states using their hashed-names values as keys. The Register() method from the m\_StateFactory attribute in vInitFactory() is registering the states constructors to manage the creation of game states.

The Game Logic uses the GameStateFactory whenever there is a request for a new game state. To choose which implementation the GameLogic shall use, it is necessary to implement its vCreateGameStateFactory() method (**Listing 63**).

**Listing 63.** Defining the GameStateFactory for the Game Logic layer.

|  |
| --- |
| virtual uge::GameStateFactory\* vCreateGameStateFactory() override  {  return LIB\_NEW sg::GameState::GameStateFactory;  } |

After a call to the method, all states transitions will request the new state from the desired GameStateFactory.

#### Creating a Custom Actor Factory and Actor Components

Although UGE provides many useful default components, most games probably will need to define their own components, with appropriate data for their specific mechanics. As stated in Section 4.2, an ActorFactory creates the game actors; thus, it is necessary to define one to use custom components.

##### Creating Some Custom Components

Before defining the factory, let us define some components[[13]](#footnote-13). As UGE’s components are data only, to define the components, the developers must examine the game design and decide what data is necessary to achieve their mechanics and gameplay goals. For instance, examining the following tasks of **Figure 26**:

1. Destroy aliens;
2. Fire bullet, avoid bombs;
3. Move spaceship, move aliens;
4. Generate shields.

It is possible to figure some of the required data members – and, thus, infer what information the components should store. Task 1 suggests a component to track the alien health. Tasks 1 and 2 suggest that the bullet must have some kind of power to reduce the health – and the game should have a way to detect the collision between bullets and aliens. Task 3 requires a world position. Tasks 1 and 4 suggest a way to increase the health or offer an alternative to enhance it.

The default components cover some of the requirements. The TransformableComponent can provide movement data. The CollidableComponent and the BulletPhysics component can help in collision detection and movement.

As there are not default components for the other data requirements, it is necessary to create them. For instance, a HealthComponent for player and alien heath; a DamageSoakingComponent for the shield’s protection and a DamageInflictingComponent for the bullets’ damage[[14]](#footnote-14).

These are examples; the developers are free to create their own or to merge/divide the data. For instance, it would be possible to merge the HealthComponent and DamageSoakingComponent into a single one. However, this assumes every actor with health also have a shield.

As the components are data-driven, it may help to consider a sample actor archetype to plan and brainstorm the data members before implementing the code (**Listing 64**).

**Listing 64.** Brainstorming the components data-driven layout.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Brainstorm" resource="brainstorm.xml">  <HealthComponent>  <InitialHealthPoints value="100"/>  <MaximumHealthPoints value="100"/>  </HealthComponent>  <DamageSoakingComponent>  <InitialProtection value="50"/>  <MaximumProtection value="100"/>  </DamageSoakingComponent>    <DamageInflictingComponent>  <DamageOutput value="5"/>  </DamageInflictingComponent>    </Actor> |

**Listing 64** contains a suggestion for the components, stating all their data members. The next listings describes how to create each of them. As they are data-only, after creating a few components, it is easy to create new ones.

**Listing 65** provides the header of a possible implementation for the HealthComponent. It is possible to observe it is data-only, defining two values for the current and the maximum health points of the actor.

**Listing 65.** Defining the HealthComponent.

|  |
| --- |
| namespace sg  {  namespace Component  {  class HealthComponent;  typedef std::shared\_ptr<HealthComponent>  HealthComponentSharedPointer;  class HealthComponent : public uge::Component::ActorComponent  {  public:  /// The name of the component.  static const char\* g\_ComponentName;  HealthComponent();  ~HealthComponent();  bool vInit(uge::XMLElement\* pInitXMLData) override;  void vPostInit() override {}  void vUpdate(const unsigned long dt) override {}  void vOnChange() override {}  const std::string vGetName() const override;  int GetHealthPoints() const;  void SetHealthPoints(const int healthPoints);  void IncrementHealthPoints(const int value);  void DecrementHealthPoints(const int value);  int SetMaximumHealthPoints() const;  void SetMaximumHealthPoints(const int healthPoints);  private:  /// The health points of the entity.  int m\_HealthPoints;  /// The max health points the entity can have.  int m\_MaxHealthPoints;  };  }  } |

The HealthComponent implementation is outlined in **Listing 66**.

**Listing 66.** Implementing the data-driven HealthComponent.

|  |
| --- |
| namespace sg  {  namespace Component  {  const char\* HealthComponent::g\_ComponentName =  "HealthComponent";  HealthComponent::HealthComponent()  : m\_HealthPoints(0), m\_MaxHealthPoints(0)  {  }  HealthComponent::~HealthComponent()  {  }  bool HealthComponent::vInit(uge::XMLElement\* pInitXMLData)  {  assert(pInitXMLData != nullptr  && "Invalid initialization data!");  uge::XMLElement xmlElement = pInitXMLData->  GetFirstChildElement("InitialHealthPoints");  if (xmlElement.IsGood())  {  xmlElement.GetIntAttribute("value",  &m\_HealthPoints);  uge::XMLElement xmlElement = pInitXMLData->  GetFirstChildElement("MaximumHealthPoints");  if (xmlElement.IsGood())  {  xmlElement.GetIntAttribute("value",  &m\_MaxHealthPoints);  return true;  }  }  return false;  }  const std::string HealthComponent::vGetName() const  {  return g\_ComponentName;  }  int HealthComponent::GetHealthPoints() const  {  return m\_HealthPoints;  }  void HealthComponent::SetHealthPoints(const int healthPoints)  {  m\_HealthPoints = healthPoints;  }  void HealthComponent::IncrementHealthPoints(const int value)  {  m\_HealthPoints += value;  }  void HealthComponent::DecrementHealthPoints(const int value)  {  m\_HealthPoints -= value;  }  int HealthComponent::SetMaximumHealthPoints() const  {  return m\_MaxHealthPoints;  }  void HealthComponent::SetMaximumHealthPoints(  const int healthPoints)  {  m\_MaxHealthPoints = healthPoints;  }  }  } |

The most important method of **Listing 66** is vInit(). This method receives an XMLElement to fill the component’s attributes with the desired initial data values.

When defining a new component, it is also useful to define a constant and static attribute with the name of the component (**Listing 65**), as it is used as a hashed-key to fetching the component. This way, should the developers change the name, the references to the component’s name would still be valid in the codebase.

The other two components have similar implementations.

##### Creating the Actor Factory

After creating the components, it is necessary to define an ActorFactory (**Listing 67** and **Listing 68**).

**Listing 67.** The tutorial game's ActorFactory.

|  |
| --- |
| namespace sg  {  class ActorFactory : public uge::ActorFactory  {  public:  ActorFactory();  ~ActorFactory();  private:  void RegisterComponents();  };  } |

**Listing 68.** Registering the components to the ActorFactory.

|  |
| --- |
| #include "ActorFactory.h"  #include "DamageInflictingComponent.h"  #include "DamageSoakingComponent.h"  #include "HealthComponent.h"  namespace sg  {  ActorFactory::ActorFactory() : uge::ActorFactory()  {  RegisterComponents();  }  ActorFactory::~ActorFactory()  {  }  void ActorFactory::RegisterComponents()  {  m\_ComponentFactory.Register<  Component::DamageInflictingComponent>(  uge::Component::GetComponentID(  Component::DamageInflictingComponent::g\_ComponentName));  m\_ComponentFactory.Register<  Component::DamageSoakingComponent>(  uge::Component::GetComponentID(  Component::DamageSoakingComponent::g\_ComponentName));  m\_ComponentFactory.Register<  Component::HealthComponent>(  uge::Component::GetComponentID(  Component::HealthComponent::g\_ComponentName));  }  } |

C++ templates makes the code of RegisterComponents() in **Listing 68** to appear hard. The Register() method of the m\_ComponentFactory attribute receives a string containing the name of the component as parameter and the component’s class as the template parameter. The factory’s internal implementation hashes the name into a number and uses its value as a key for creating and retrieving components.

To start using the new components and ActorFactory, it is just necessary to set up the tutorial game’s GameLogic’s vCreateActorFactory() method (**Listing 69**).

**Listing 69.** Creating the custom factory in the tutorial’s game logic.

|  |
| --- |
| virtual uge::ActorFactory\* vCreateActorFactory() override  {  return LIB\_NEW sg::ActorFactory;  } |

Now the developers are able to use both the custom and default UGE’s components. Next section debuts the new factory creating some game actors.

#### Creating the Game Entities and the Game Scene

From this subsection until the remaining of the section, this tutorial will focus on the Running game state and the GameLogic for the implementation of the Game Logic layer. This tutorial will add private methods to its class and call them using the IGameState’s abstract interface methods.

##### Creating the First Actor

The game world is still empty. With an empty world, there is no simulation and, consequently, no game. It is time to create the first game actor and insert it into the game world to start prototyping the game.

**Listing 70** leads the implementation of the Running game state with the CreateGameActors() method.

**Listing 70.** Creating the first actors.

|  |
| --- |
| class Running : public uge::GameState::Running  {  friend class uge::BaseGameLogic;  public:  Running();  virtual ~Running();  virtual bool vInit(uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(  const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(unsigned long timeElapsed) override;  virtual bool vOnRender(unsigned long timeElapsed) override;  private:  bool CreateGameActors();  }; |

The new method should be called in the vInit() method (**Listing 71**).

**Listing 71.** Initializing the game actors.

|  |
| --- |
| bool Running::vInit(uge::BaseGameLogic\* pGameLogic)  {  uge::GameState::BaseGameState::vInit(pGameLogic);  if (!CreateGameActors())  {  return false;  }  return true;  } |

The player’s spaceship is a good candidate for a first actor. To keep it simple, **Listing 72** provides a simple prototype. This listening also illustrates the data-driven architecture of UGE: the spaceship, as any other concrete actor, does have code – it is just a XML resource. The following listings will add more components to the actor and show the results in the game world.

**Listing 72.** A draft for the first game actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource="data/game/actors/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  </Actor> |

The Running game state is a subclass of the BaseGameState. As such, during its initialization (**Listing 71**), the super class stores a pointer to game logic in the m\_pGameLogic protected attribute, which is a pointer to the BaseGameLogic implementation of the game (sg::GameLogic – or simply GameLogic from this point on).

**Listing 73** uses this pointer to create the actor and add it to the game logic (m\_pGameLogic->vCreateActor()). The GameLogic has an instance of the ActorFactory defined in **Listing 67** and, thus, it has access for all the UGE’s default components and the registered game’s specific components.

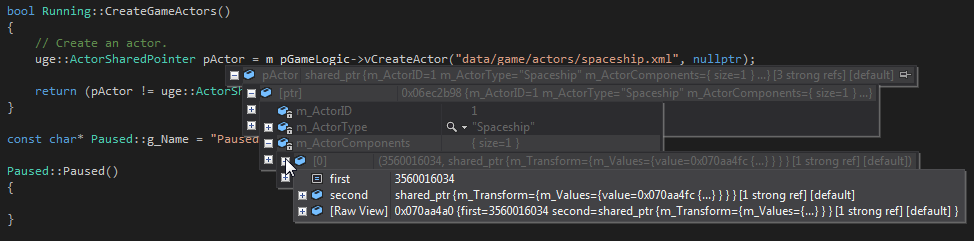
**Listing 73.** Creating and adding the actor to the game logic.

|  |
| --- |
| bool Running::CreateGameActors()  {  // Create an actor.  uge::ActorSharedPointer pActor =  m\_pGameLogic->vCreateActor(  "data/game/actors/spaceship.xml", nullptr);  return (pActor != uge::ActorSharedPointer());  } |

If successful, the return Actor pointer is non-null. This pointer references a new game actor created according to its archetype definition (**Listing 72**). Therefore, it has all the desired components created with the specified initial data. For instance, its world position stored in its TransformableComponent is (0.0f, 0.0f, 0.0f).

For quick access to actor, it might be a good idea to save the pointer in an attribute. Otherwise, it is necessary to save its ActorID (pActor->GetActorID()) and query the GameLogic for the pointer whenever the actor is needed.

Running the application and setting a breakpoint after the creation of actor will show it has only one component: as expected, the specified TransformableComponent (**Figure 30**).



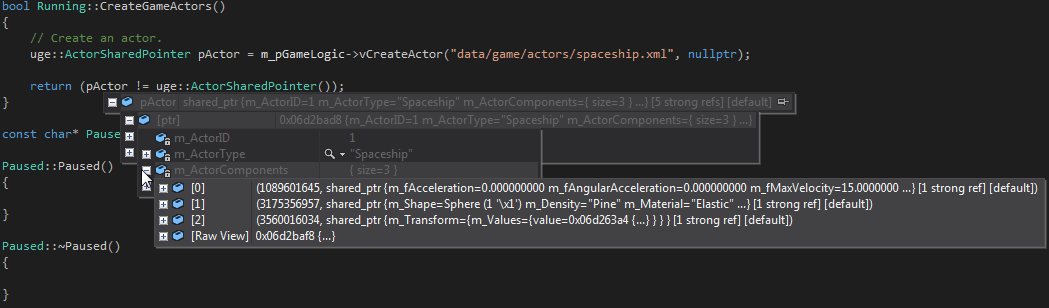
**Figure 30.** The first actor created into the tutorial game.

It still has no output representation, though. **Listing 74** adds a few more components to the Actor. This data was added to the actor without requiring a single change to the code base, demonstrating the usefulness of the entity-component approach combined with a data-driven architecture.

**Listing 74.** Attaching a few more components into the actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource="data/game/actors/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Sphere">  <Radius r="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="10.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>  </Actor> |

After running the application again[[15]](#footnote-15), the actor now has the three components stated in **Listing 74**: a TransformableComponent, a CollidableComponent and a BulletPhysicsComponent (**Figure 31**).



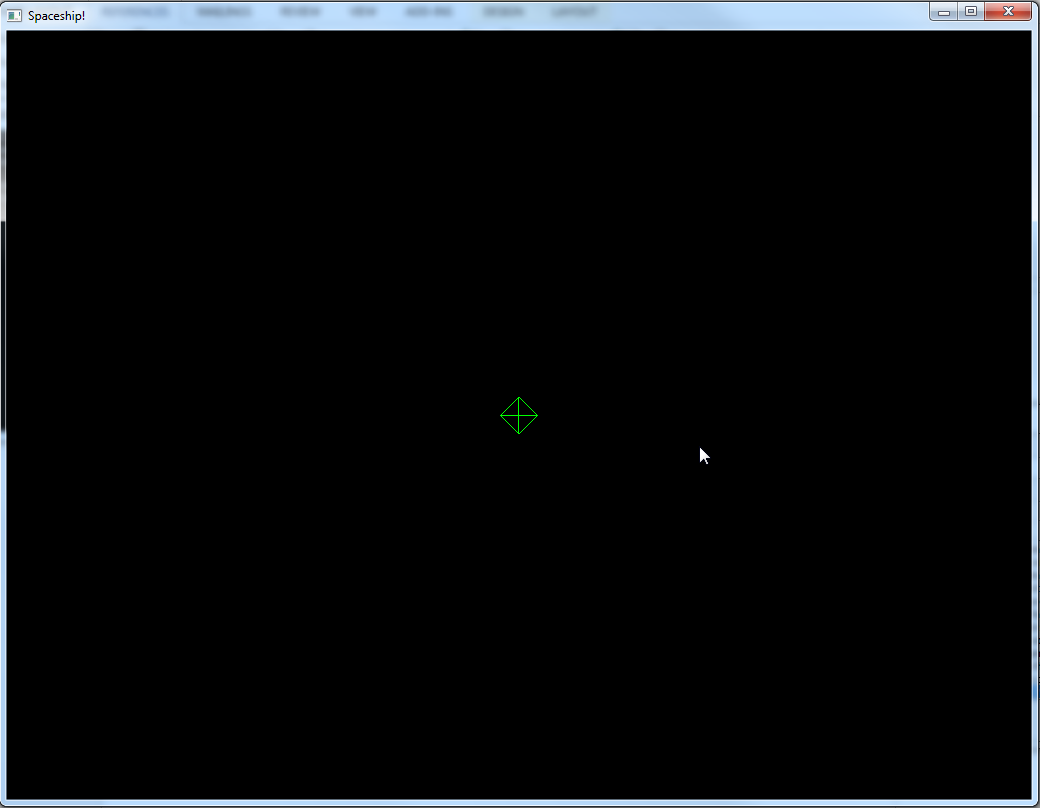
**Figure 31.** A new iteration on the first actor – adding new components to the archetype.

There is still no output component, so there is not any output representation. However, now the entity has all the required components for the Physics subsystem. Thus, it is possible to visualize the game scene using the Physics subsystem debugger – it only requires adding registering the actor to the subsystem, as shown in **Listing 75**.

**Listing 75.** Adding the actor the Physics subsystem.

|  |
| --- |
| bool Running::CreateGameActors()  {  // Create an actor.  uge::ActorSharedPointer pActor =  m\_pGameLogic->vCreateActor("data/game/actors/spaceship.xml",  nullptr);  if (pActor == uge::ActorSharedPointer())  {  return false;  }  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vSetGravity(uge::Vector3(0.0f, 0.0f, 0.0f));  // Add the actor to the physics simulation.  pPhysics->vAddActor(pActor);  return true;  } |

The result is illustrated in **Figure 32**: it shows the physics rigid-body that represents the actor in the game world[[16]](#footnote-16).



**Figure 32.** Using the Physics subsystem debugger to visualize the game world.

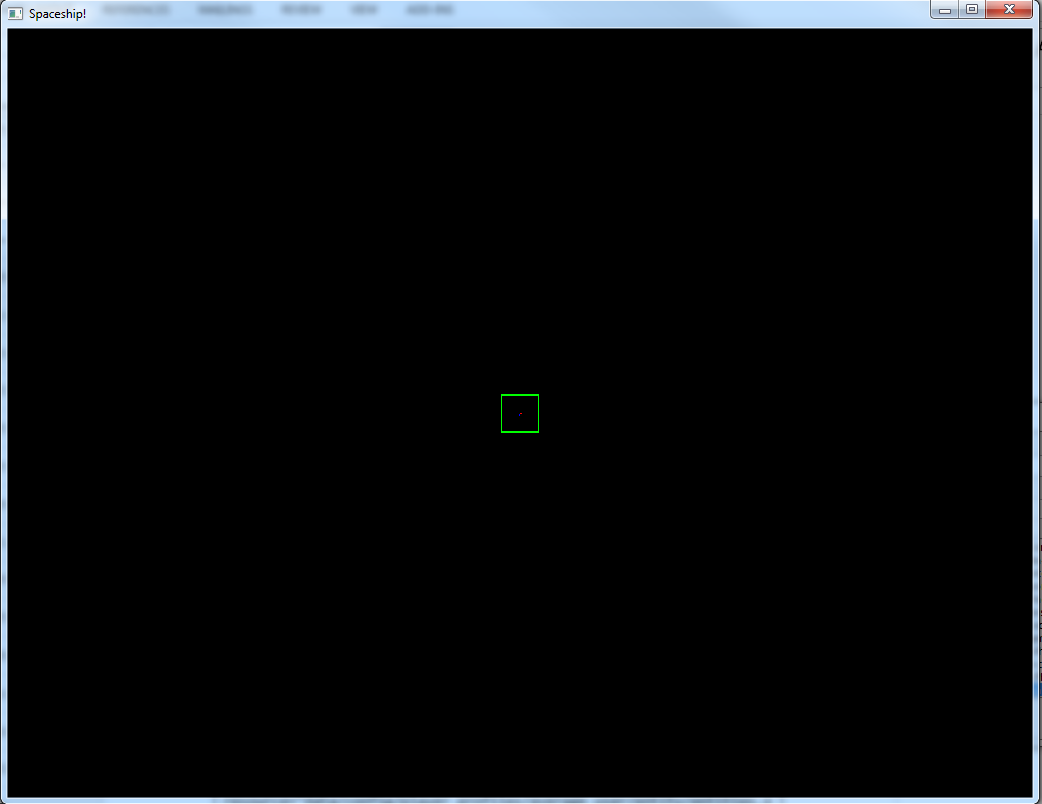
**Figure 32** illustrates UGE’s approach to implement UA-Games: the game logic does not depends on its representation. The game world exists and simulates itself[[17]](#footnote-17).

A spherical spaceship might sounds strange. **Listing 76** changes the shape to a box; it also adds the custom game logic components created in Section 7.2.5.3.

**Listing 76.** Changing the collision shape to a box.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource="data/game/actors/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Box">  <Dimension x="1.0f" y="1.0f" z="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="10.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>  <HealthComponent>  <InitialHealthPoints value="100"/>  <MaximumHealthPoints value="100"/>  </HealthComponent>  <DamageSoakingComponent>  <InitialProtection value="50"/>  <MaximumProtection value="100"/>  </DamageSoakingComponent>  </Actor> |

Once again, without recompiling the code, the actor’s collision shape is now a box (**Figure 33**). The actor also has a HealthComponent and a DamageSoakingComponent, although those do not do anything so far: contrarily to the Physics subsystem for the TransformableComponent, CollidableComponent and BulletPhysicsComponent components, there is not default subsystem for the HealthComponent and the DamageSoakingComponent components. Thus, we will need to define these subsystems as appropriate.



**Figure 33.** The result of changing the collision shape in **Listing 76**.

Next section creates some additional actors before implementing gameplay code. Before moving on, it might be interesting to change the values in the call to vSetGravity() in **Listing 75** and recompile the code to see the results – for instance, setting the gravity to (0.0f, -9.8f, 0.0f) will make the spaceship fall down.

##### Creating More Actors

The XML resource of **Listing 76** defined a spaceship archetype. As it is an archetype, it can also be a model for other spaceships – all spaceships created using its resource will have the defined components.

To ease the implementation of the game logic, this section will also describes possible archetypes for the remaining archetypes of **Figure 26**: enemies (aliens) and for the projectiles (bullets and bombs).

**Listing 77** describes a possible definition for the aliens.

**Listing 77.** The XML resource for enemy actors.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="20.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Box">  <Dimension x="1.0f" y="1.0f" z="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="15.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>  <HealthComponent>  <InitialHealthPoints value="100"/>  <MaximumHealthPoints value="100"/>  </HealthComponent>  </Actor> |

With the exception of the missing DamageSoakingComponent[[18]](#footnote-18), it is the same as the spaceship defined in **Listing 76**, save for the default values. This shows another benefit of components: provided components are general enough, they are usually reusable for many different actors. Every actor which needs a world transform will have a TransformableComponent. For Physics, the actor will have both a CollidableComponent and a BulletPhysicsComponent and so on.

For this same reason, both the bullets (**Listing 78**) and bombs (**Listing 79**) are also similar between each other. This implementation considered bombs slightly stronger and larger than bullets, as it is possible to observe in the Scale value of the TransformableComponent and in the DamageOutput value of the DamageInflictingComponent.

**Listing 78.** The XML resource for bullet actors.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Bullet" resource="data/game/actors/bullet.xml">  <TransformableComponent>  <Position x="10.0f" y="0.0f" z="-30.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="3.0f" y ="3.0f" z="3.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Sphere">  <Radius r="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="20.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>  <DamageInflictingComponent>  <DamageOutput value="5"/>  </DamageInflictingComponent>  </Actor> |

**Listing 79.** The XML resource for bomb actors.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Bomb" resource="data/game/actors/bomb.xml">  <TransformableComponent>  <Position x="-10.0f" y="0.0f" z="-30.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="5.0f" y ="5.0f" z="5.0"/>  </TransformableComponent>  <CollidableComponent>  <Shape type="Sphere">  <Radius r="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <MaxVelocity v="20.0f"/>  <MaxAngularVelocity v="0.0f"/>  </BulletPhysicsComponent>  <DamageInflictingComponent>  <DamageOutput value="10"/>  </DamageInflictingComponent>  </Actor> |

If it seems strange to think of projectiles as actors, it is important to note actors are everything that participates in the simulation and interacts with game world and with others actors.

The code to add these actors to the game logic is similar to the one in **Listing 73**. To avoid code duplication, **Listing 80** and **Listing 81** refactor and divide the code from CreateGameActors() into different methods: CreateAndRegisterActor(), CreateActor() and AddActorToPhysics(). This will help evolutionary iterations in the implementation.

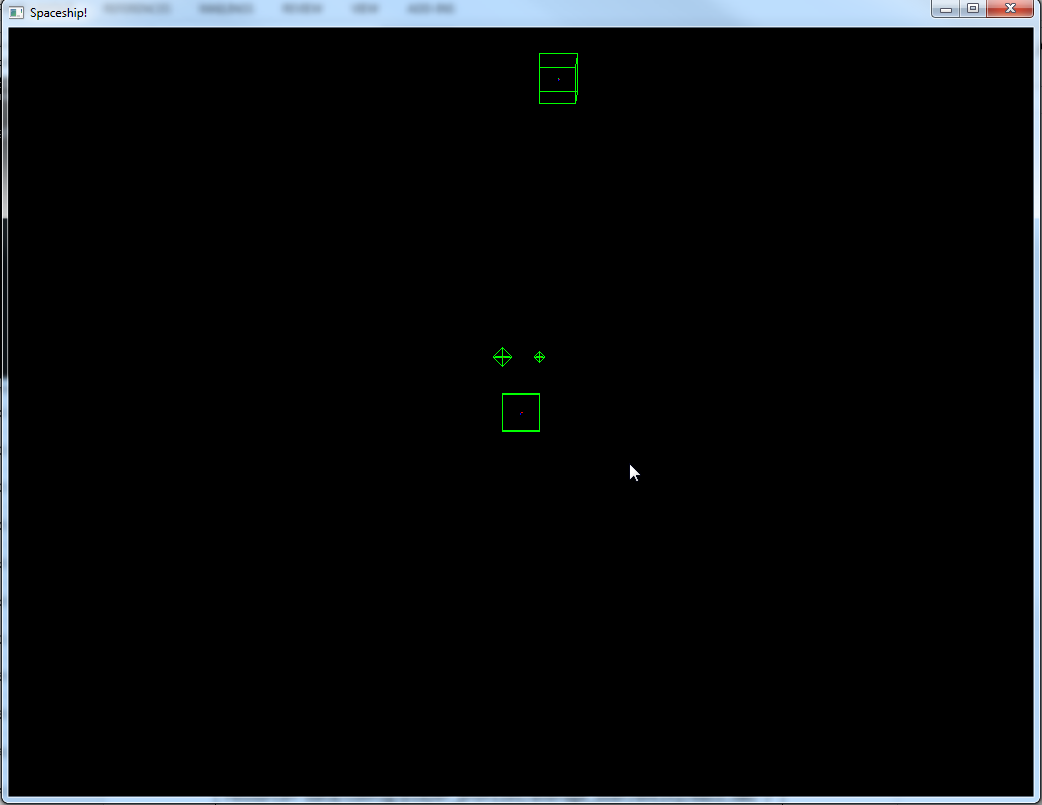
**Listing 80.** Refactoring the Running game state.

|  |
| --- |
| class Running : public uge::GameState::Running  {  friend class uge::BaseGameLogic;  public:  Running();  virtual ~Running();  virtual bool vInit(uge::BaseGameLogic\* pGameLogic) override;  virtual bool vTailorToProfile(const std::string& xmlResourceFilename) override;  virtual bool vDestroy() override;  virtual bool vOnUpdate(unsigned long timeElapsed) override;  virtual bool vOnRender(unsigned long timeElapsed) override;  private:  bool ConfigureGamePhysics();  bool CreateGameActors();  uge::ActorSharedPointer CreateAndRegisterActor(  const std::string& actorResourceFile,  uge::XMLElement\* pActorOverride = nullptr);  uge::ActorSharedPointer CreateActor(  const std::string& actorResourceFile,  uge::XMLElement\* pActorOverride = nullptr);  void AddActorToPhysics(uge::ActorSharedPointer pActor);  void CollisionStarted(uge::IEventDataSharedPointer pEventData);  void CollisionEnded(uge::IEventDataSharedPointer pEventData);  private:  uge::ActorSharedPointer m\_pSpaceship;  }; |

**Listing 81.** Implementing the new methods of the refactor.

|  |
| --- |
| bool Running::vInit(uge::BaseGameLogic\* pGameLogic)  {  std::cout << "[Running] Game is running!" << std::endl;  uge::GameState::BaseGameState::vInit(pGameLogic);  ConfigureGamePhysics();  if (!CreateGameActors())  {  return false;  }  return true;  }  bool Running::ConfigureGamePhysics()  {  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vSetGravity(uge::Vector3(0.0f, 0.0f, 0.0f));  return true;  }  bool Running::CreateGameActors()  {  bool bSuccess = CreateAndRegisterActor(  "data/game/actors/spaceship.xml");  return bSuccess;  }  uge::ActorSharedPointer Running::CreateAndRegisterActor(  const std::string& actorResourceFile,  uge::XMLElement\* pActorOverride)  {  // Create an actor.  uge::ActorSharedPointer pActor = CreateActor(actorResourceFile,  pActorOverride);  if (pActor == uge::ActorSharedPointer())  {  return uge::ActorSharedPointer();  }  // Add the actor to the physics simulation.  AddActorToPhysics(pActor);  return pActor;  }  uge::ActorSharedPointer Running::CreateActor(  const std::string& actorResourceFile,  uge::XMLElement\* pActorOverride)  {  return m\_pGameLogic->vCreateActor(  actorResourceFile, pActorOverride);  }  void Running::AddActorToPhysics(uge::ActorSharedPointer pActor)  {  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vAddActor(pActor);  } |

After compiling and running the code of **Listing 81**, **Figure 34** illustrates the game world so far.



**Figure 34.** The game world defined in **Listing 81**.

As the XML resources defined are archetypes, it is possible to create multiple actors from the same resource and override its default component values (or even to add new components).

**Listing 82** defines a list of aliens using the archetype defined in **Listing 77**.

**Listing 82.** Defining many enemies at once using the alien actor archetype.

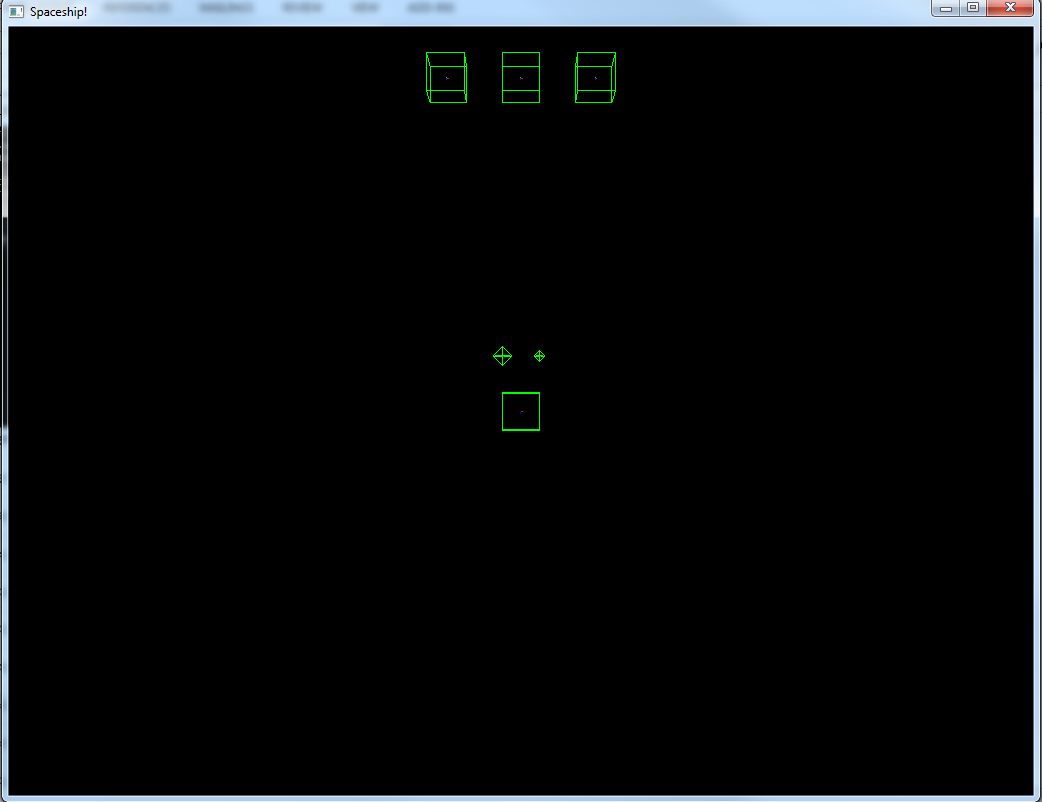
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actors resource="data/game/actors/aliens.xml">  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  </Actor>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="40.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  </Actor>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="-40.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <DamageSoakingComponent>  <InitialProtection value="50"/>  <MaximumProtection value="100"/>  </DamageSoakingComponent>  </Actor>  </Actors> |

For each actor, **Listing 82** overrides the default position with a new one. With the current values, the three actors will be next one to another; the third actor will also have a new DamageSoakingComponent. **Listing 83** shows how to load **Listing 82**’s resource into the game. The procedure is similar to creating a single actor – the only difference is an XMLElement containing the new data is also given to the vCreateActor() method.

**Listing 83.** Creating many enemies at once using the alien actor archetype.

|  |
| --- |
| bool Running::CreateGameActors()  {  m\_pSpaceship = CreateAndRegisterActor(  "data/game/actors/spaceship.xml");  if (m\_pSpaceship == uge::ActorSharedPointer())  {  return false;  }  uge::ActorSharedPointer pActor;  uge::XMLFile aliens;  aliens.OpenFile("data/game/actors/aliens.xml", uge::File::FileMode::FileReadOnly);  uge::XMLElement xmlRoot = aliens.GetRootElement();  for (uge::XMLElement xmlElement =  xmlRoot.GetFirstChildElement("Actor");  xmlElement.IsGood();  xmlElement = xmlElement.GetNextSiblingElement())  {  pActor = CreateAndRegisterActor(  "data/game/actors/alien.xml", &xmlElement);  if (pActor == uge::ActorSharedPointer())  {  return false;  }  }  aliens.CloseFile();  pActor = CreateAndRegisterActor("data/game/actors/bullet.xml");  if (pActor == uge::ActorSharedPointer())  {  return false;  }  pActor = CreateAndRegisterActor("data/game/actors/bomb.xml");  if (pActor == uge::ActorSharedPointer())  {  return false;  }  return true;  } |

As the CreateGameActors() changed, it is necessary to compile the code. After compiling, however, it is possible to add or remove as many aliens by modifying **Listing 82**’s resource. The result is illustrated in **Figure 35**.



**Figure 35.** Using an actor XML resource as an archetype to create many similar actors.

This tutorial will use this same approach as one of the strategies to tailor the game entities and presentation according to a player profile in Sections and 7.2.5.8 and 7.2.6. Those sections discuss how adding new components (for instance, output components) or changing the existing component’s values modify the run time of the game.

#### Defining Game Events

It is time to define some events for the tutorial game. As discussed in Section 4.3, events are useful for UA-Games in many ways, improving both the gameplay and game presentation.

The design **Figure 26** is useful once again: it suggest some key events, such as hits and collisions, enemies spawning and disappearing, actor movement and firing projectiles.

All these activities are interesting points for defining game events. Once the events are identified and dispatched, it is possible to handle an event in many different ways, for equally different purposes.

##### Creating Events

As it was the case with components, albeit UGE provides several default events, a game implementation usually requires many others. For instance, the tutorial game features collisions, enemies disappearing and firing projectiles. All these features are suitable to become game events. For instance, **Listing 84** describes an event for destroying an alien enemy (AlienDestroyed).

**Listing 84.** A sample event to communicate the destruction of aliens.

|  |
| --- |
| namespace sg  {  class AlienDestroyed : public uge::BaseEventData  {  public:  static const uge::EventType sk\_EventType;  explicit AlienDestroyed(uge::ActorID actorID)  : m\_ActorID(actorID)  {  }  virtual const uge::EventType& vGetEventType() const override  {  return sk\_EventType;  }  virtual uge::IEventDataSharedPointer vCopy() const override  {  return uge::IEventDataSharedPointer(  LIB\_NEW AlienDestroyed(m\_ActorID));  }  virtual void vSerialize(std::ostrstream& out) const override  {  out << m\_ActorID;  }  virtual void vDeserialize(std::istrstream& in) override  {  in >> m\_ActorID;  }  virtual const char\* vGetName() const override  {  return "AlienDestroyed";  }  uge::ActorID GetActorID() const  {  return m\_ActorID;  }  private:  uge::ActorID m\_ActorID;  };  const uge::EventType AlienDestroyed::sk\_EventType(0xe91b0343);  } |

As stated in Section 4.3, an event is characterized by its data members. It is important not to use pointers or references to the data, as it might change (or become invalid) before a listener handles the event.

It is also important to define an uge::EventType with a GUID to the event. The GUID is the event’s identifier used in the IEventManager implementation.

The definition of any other events should be similar to **Listing 84**.

##### Handling Events

To use a default or game specific event, it is necessary to define an event delegate (also known as listener or handler). A delegate is a method or a function that handles the event according to the delegate’s implementation. For instance, it might use the event to add or remove actors or components or to provide feedback to the user about an action.

To register a delegate to an event, it is necessary to use the IEventManager’s method vAddListener(). **Listing 85** registers the event created in Section 7.2.5.5.1 and a few default ones to the global event manger.

**Listing 85.** Registering game events handlers.

|  |
| --- |
| #include "Events/GameEvents.h"  class Running : public uge::GameState::Running  {  // ...  private:  // ...  void RegisterEvents();  void UnregisterEvents();  void CollisionStarted(uge::IEventDataSharedPointer pEventData);  void CollisionEnded(uge::IEventDataSharedPointer pEventData);  // ...  };  void Running::RegisterEvents()  {  // Creating and registering the event handlers.  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(this, &Running::CollisionStarted);  uge::IEventManager::Get()->vAddListener(functionDelegate,  uge::EvtData\_PhysCollision::sk\_EventType);  functionDelegate = fastdelegate::MakeDelegate(  this, &Running::CollisionEnded);  uge::IEventManager::Get()->vAddListener(functionDelegate,  uge::EvtData\_PhysSeparation::sk\_EventType);  functionDelegate = fastdelegate::MakeDelegate(  this, &Running::AlienDestroyed);  uge::IEventManager::Get()->vAddListener(functionDelegate,  sg::AlienDestroyed::sk\_EventType);  }  void Running::UnregisterEvents()  {  // Removing the delegates.  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(this, &Running::CollisionStarted);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, uge::EvtData\_PhysCollision::sk\_EventType);  functionDelegate = fastdelegate::MakeDelegate(  this, &Running::CollisionEnded);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, uge::EvtData\_PhysSeparation::sk\_EventType);  functionDelegate = fastdelegate::MakeDelegate(  this, &Running::AlienDestroyed);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, sg::AlienDestroyed::sk\_EventType);  }  void Running::CollisionStarted(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<uge::EvtData\_PhysCollision> pData =  std::static\_pointer\_cast<uge::EvtData\_PhysCollision>(  pEventData);  printf("Actors %u and %u collided!\n",  pData->GetActorA(), pData->GetActorB());  }  void Running::CollisionEnded(  uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<uge::EvtData\_PhysSeparation> pData =  std::static\_pointer\_cast<  uge::EvtData\_PhysSeparation>(pEventData);  printf("Actors %u and %u stopped colliding!\n",  pData->GetActorA(), pData->GetActorB());  }  void Running::AlienDestroyed(  uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::AlienDestroyed> pData =  std::static\_pointer\_cast<sg::AlienDestroyed>(pEventData);  printf("Alien with ID %u was destroyed!\n", pData->GetActorID());  // Could increase score or difficult, etc.  } |

With **Listing 85** code, whenever a collision starts or ends or an alien actor is destroyed, the event delegate will run the specified code do handle the event. As an event may have several delegates, it is possible to handle the event in different ways. For instance, the game logic might create a new, stronger enemy, whilst a game view might display a graphical animation and play an explosion sound. **Listing 85** has a much simpler treatment, as it only writes a message to stdout.

To illustrate the event handler code, **Listing 86** adds a new method call to vOnUpdate() to check and remove destroyed actors (actors whose current health points on the HealthComponent are equal of below zero). The actors that satisfy this condition are removed from the game and an AlienDestroyed event is dispatched.

**Listing 86.** Removing destroyed actors from the game.

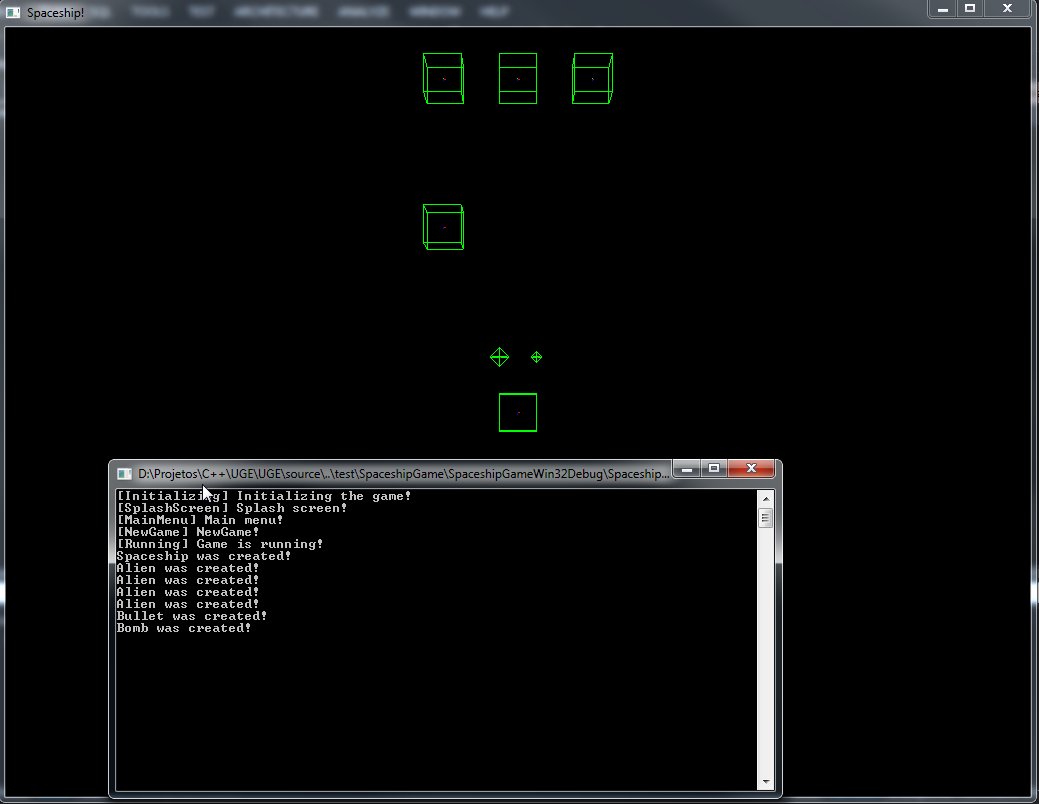
|  |
| --- |
| bool Running::vOnUpdate(unsigned long timeElapsed)  {  bool bSuccess = uge::GameState::Running::vOnUpdate(timeElapsed);  RemoveDestroyedActors();  return bSuccess;  }  void Running::RemoveDestroyedActors()  {  std::vector<uge::ActorID> destroyedActorIDs;  sg::GameLogic\* pGameLogic =  dynamic\_cast<sg::GameLogic\*>(m\_pGameLogic);  for (const auto& actorIt : pGameLogic->m\_Actors)  {  uge::ActorID actorID = actorIt.first;  uge::ActorSharedPointer pActor = actorIt.second;  std::weak\_ptr<Component::HealthComponent> pComponent =  pActor->GetComponent<sg::Component::HealthComponent>(  sg::Component::HealthComponent::g\_ComponentName);  // Check if the actor has a HealthComponent.  if (!pComponent.expired())  {  Component::HealthComponentSharedPointer  pSharedComponent = pComponent.lock();  if (pSharedComponent->GetHealthPoints() <= 0)  {  destroyedActorIDs.push\_back(actorID);  }  }  }  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  for (uge::ActorID actorID : destroyedActorIDs)  {  pPhysics->vRemoveActor(actorID);  pGameLogic->vRemoveSceneNode(actorID);  pGameLogic->vDestroyActor(actorID);  std::shared\_ptr<sg::AlienDestroyed> pEvent(  LIB\_NEW sg::AlienDestroyed(actorID));  // Event will be triggered during the vUpdate() call.  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  } |

To test the code in **Listing 86**, it is possible to add a new actor to **Listing 82**’s resource overriding the HealthComponent initial health points’ value (**Listing 87**).

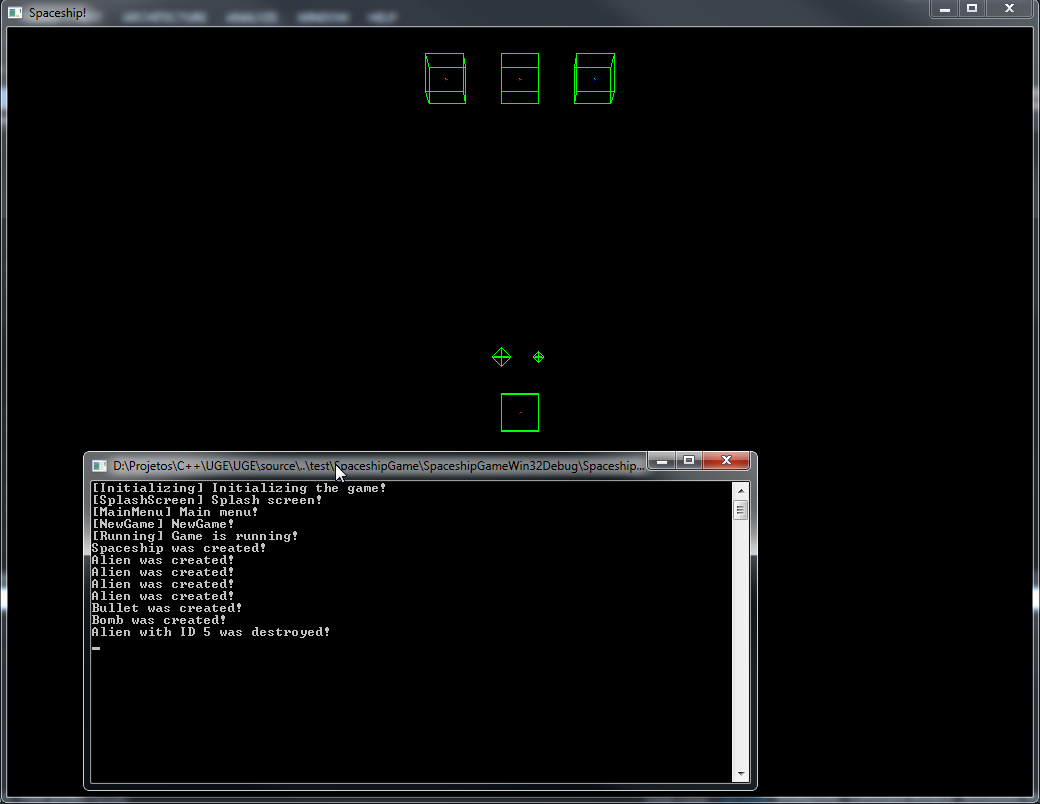
**Listing 87.** Adding a destroyed alien to test the event.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actors resource="data/game/actors/aliens.xml">  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  </Actor>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="40.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  </Actor>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="-40.0f" y="0.0f" z="-180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <DamageSoakingComponent>  <InitialProtection value="50"/>  <MaximumProtection value="100"/>  </DamageSoakingComponent>  </Actor>  <Actor type="Alien" resource="data/game/actors/alien.xml">  <TransformableComponent>  <Position x="-40.0f" y="0.0f" z="-100.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <HealthComponent>  <InitialHealthPoints value="0"/>  <MaximumHealthPoints value="100"/>  </HealthComponent>  </Actor>  </Actors> |

It is interesting to change the InitialHealthPoints value to a positive number then back to zero and see the differences (**Figure 36** and **Figure 37**, respectively). In **Figure 36**, the new alien has a positive value for its hit points and, thus, is still in the game world simulation. In **Figure 37**, however, the alien is removed from the game world after the first call to vOnUpdate() during the game loop.



**Figure 36.** A positive value assigned to InitialHealthPoints (100).



**Figure 37.** A zero value assigned to InitialHealthPoints (0).

This kind of change has a more subtle benefit, explained in Section 7.2.5.6.

#### Defining Game Commands

**Listing 87** has a subtle benefit: it is possible to see the game execution changed when the health value was modified to a non-positive value. When the alien health was zero, an event was dispatched **Listing 86**. When the AlienDestroyed() method in Running received the event, it wrote a message to stdout (**Listing 85**). This was a very simple example. However, it is possible to define several other event handlers, including more complex one. For instance, one event could have gameplay consequences, such as an event handler creating a new alien after one was destroyed.

For a UA-Game, there is an even better benefit: it is possible to use events as game commands to control the behavior of game entities. Instead of binding a mechanic to a physical-level interaction (such as pressing the space bar to fire), it is possible to bind a mechanic to an event (an entity fires a projectile after handling a FireProjectile event).

This is the goal of UGE’s game commands (Section 4.4): to provide a high-level mechanics to control game entities. Combined with a lower-level mechanism (UGE’s lower level command), it is possible to define data-driven game controllers for the game.

This section will illustrate game commands by defining two examples: the MoveActor and the FireProjectile events.

##### Moving an Actor with a Game Command

The design illustrated in **Figure 26** restricts the actor’s movements to one axis of freedom, allowing movement in a single axis. For a tutorial, this is a good choice, as it simplifies the implementation. However, as UGE is a 3D game engine, it would also be possible to move the entity in any of the X, Y and/or Z axis of a 3D world.

As the design define left and right as directions, it might be a good idea to restrict the movement to the X-axis. **Listing 88** contains a possible implementation for a MoveActor game command.

**Listing 88.** Defining a game command to move actors.

|  |
| --- |
| class MoveActor : public uge::BaseEventData  {  public:  enum class Direction : char  {  Left,  Right  };  static const uge::EventType sk\_EventType;  explicit MoveActor(uge::ActorID actorID,  MoveActor::Direction direction)  : m\_ActorID(actorID), m\_Direction(direction)  {    }  virtual const uge::EventType& vGetEventType() const override  {  return sk\_EventType;  }  virtual uge::IEventDataSharedPointer vCopy() const override  {  return uge::IEventDataSharedPointer(  LIB\_NEW MoveActor(m\_ActorID, m\_Direction));  }  virtual void vSerialize(std::ostrstream& out) const override  {  out << m\_ActorID << static\_cast<char>(m\_Direction);  }  virtual void vDeserialize(std::istrstream& in) override  {  char directionValue;  in >> m\_ActorID >> directionValue;    m\_Direction =  static\_cast<MoveActor::Direction>(directionValue);  }  virtual const char\* vGetName() const override  {  return "MoveActor";  }  uge::ActorID GetActorID() const  {  return m\_ActorID;  }  MoveActor::Direction GetDirection() const  {  return m\_Direction;  }  private:  uge::ActorID m\_ActorID;  MoveActor::Direction m\_Direction;  };  const uge::EventType MoveActor::sk\_EventType(0x9c4753d0); |

The event is simple: it only provides the ActorID and the movement direction. Thus, the event does not move the actor: it lets a more adequate class to handle the movement. One good candidate is the GameLogic or a GameState, such as Running (**Listing 89**).

**Listing 89.** Handling a game command.

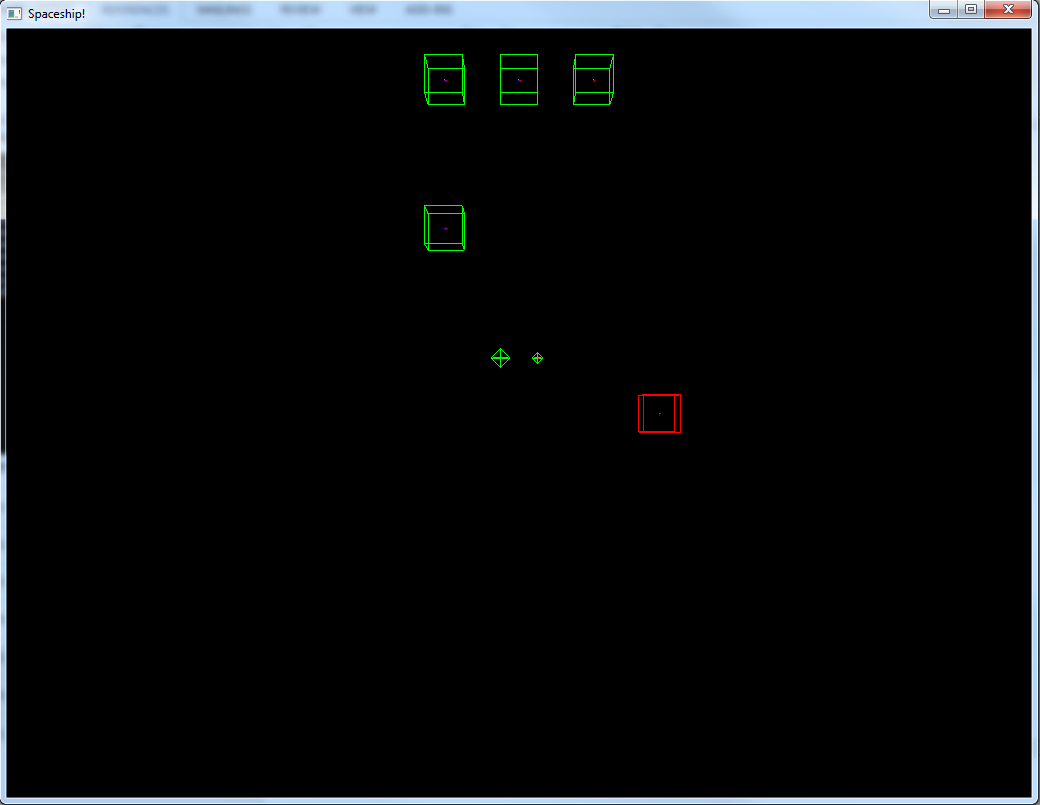
|  |
| --- |
| void Running::RegisterEvents()  {  // Creating and registering the event handlers.  // ...  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(this, &Running::MoveActor);  uge::IEventManager::Get()->vAddListener(functionDelegate, sg::MoveActor::sk\_EventType);  }  void Running::UnregisterEvents()  {  // Creating and registering the event handlers.  // ...  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(this, &Running::MoveActor);  uge::IEventManager::Get()->vRemoveListener(functionDelegate,  sg::MoveActor::sk\_EventType);  }  void Running::MoveActor(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::MoveActor> pData =  std::static\_pointer\_cast<sg::MoveActor>(pEventData);  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  MoveActor::Direction direction = pData->GetDirection();  if (direction == MoveActor::Direction::Left)  {  pPhysics->vApplyForce(pData->GetActorID(),  uge::Vector3(1.0f, 0.0f, 0.0f), 10.0f);  }  else  {  pPhysics->vApplyForce(pData->GetActorID(),  uge::Vector3(-1.0f, 0.0f, 0.0f), 10.0f);  }  } |

To test this command, **Listing 90** applies creates a random MoveActor game command to the spaceship actor on every game logic update.

**Listing 90.** Dispatching a game command with a random direction.

|  |
| --- |
| bool Running::vOnUpdate(unsigned long timeElapsed)  {  bool bSuccess = uge::GameState::Running::vOnUpdate(timeElapsed);  RemoveDestroyedActors();  // Create the move actor event.  MoveActor::Direction direction = (std::rand() % 2) ?  MoveActor::Direction::Left : MoveActor::Direction::Right;  std::shared\_ptr<sg::MoveActor> pEvent(  LIB\_NEW sg::MoveActor(m\_pSpaceship->GetActorID(), direction));  uge::IEventManager::Get()->vQueueEvent(pEvent);  return bSuccess;  } |

**Figure 38** shows the spaceship position after letting the simulation run for a while.



**Figure 38.** The spaceship (in red) after several random game commands.

It is possible to improve the speed by using the max speed attribute of the BulletPhysicsComponent (**Listing 91**).

**Listing 91.** Setting a constant speed to the actor.

|  |
| --- |
| #include <Core/EntityComponent/Component/Implementation/BulletPhysicsComponent.h>  void Running::MoveActor(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::MoveActor> pData =  std::static\_pointer\_cast<sg::MoveActor>(pEventData);  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  MoveActor::Direction direction = pData->GetDirection();  if (direction == MoveActor::Direction::Left)  {  pPhysics->vApplyForce(pData->GetActorID(),  uge::Vector3(1.0f, 0.0f, 0.0f), 1.0f);  }  else  {  pPhysics->vApplyForce(pData->GetActorID(),  uge::Vector3(-1.0f, 0.0f, 0.0f), 1.0f);  }  uge::ActorSharedPointer pActor = m\_pGameLogic->vGetActor(  pData->GetActorID()).lock();  uge::Component::BulletPhysicsComponentSharedPointer  pActorPhysicsComponent = pActor->GetComponent<  uge::Component::BulletPhysicsComponent>(  uge::Component::BulletPhysicsComponent::g\_ComponentName).lock();  float fMaxVelocityMagnitude =  pActorPhysicsComponent->vGetMaxVelocity();  uge::Vector3 velocity = pActorPhysicsComponent->vGetVelocity();  float fCurrentVelocityMagnitude = velocity.Length();  velocity \*= (fMaxVelocityMagnitude / fCurrentVelocityMagnitude);  pActorPhysicsComponent->vSetVelocity(velocity);  } |

As the MoveActor game command provides a unified mechanism to command the game actors, the real implementation would not use a random generated command. Rather, a GameController would send the command – it could be human or AI controlled. This is discussed in Section 7.2.6.5.

##### Firing Projectiles with a Game Command

The same strategy applies to firing projectiles. The game logic would create a new actor (the desired projectile) with its initial position in front of the actor. To move the actor, it would only be necessary to apply an initial impulse.

#### Defining the Gameplay

At this moment, this section is left as an exercise to the reader. This tutorial will provide an overview of the remaining UGE’s features before implementing the gameplay.

The gameplay implementation is available at Section 7.2.7.

#### Player Profile: Entity Specializations

Section 7.2.4 (more particularly Section 7.2.4.1) mentioned player profiles (described in Section 4.6). However, the game logic had not used a player profile yet.

It is time to use the active player profile to customize the gameplay for user needs. The idea is similar to the one described in Section 7.2.5.4.2: the profile allows modifying the game actors’ component data or to add new components.

Many games might be able to use the same implementation for tailoring the game’s actors with overloaded data or new components with little to no change. A possible implementation is shown in **Listing 92**.

**Listing 92.** Implementing player profile support to the game logic.

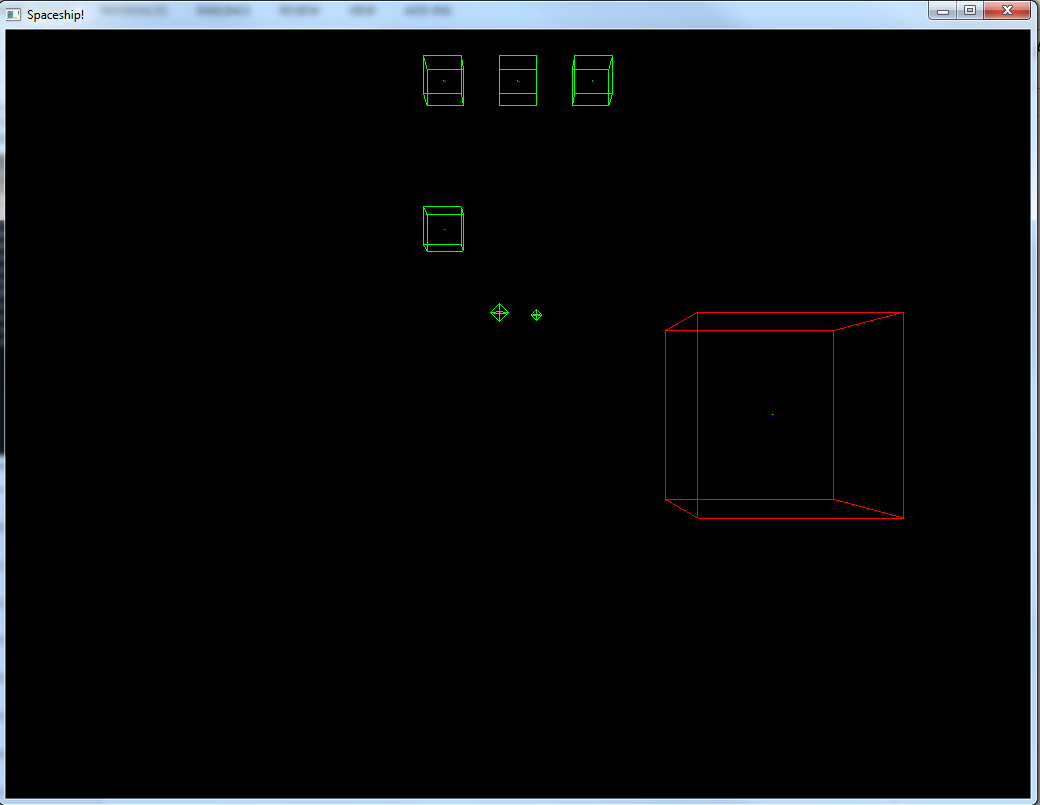
|  |
| --- |
| class Running : public uge::GameState::Running  {  // ...  private:  // ...  void AddActorToPhysics(uge::ActorSharedPointer pActor);  void RemoveActorFromPhysics(uge::ActorID actorID);      void LoadProfile(const std::string& xmlResourceFilename);  // This should be a multi-map if using an actor's  // resource as archetype.  std::map<std::string, uge::ActorID> m\_ActorNameToID;  };  bool Running::vTailorToProfile(  const std::string& xmlResourceFilename)  {  // Overrides the presentation data using the chosen profile.  LoadProfile(xmlResourceFilename);  return true;  }  uge::ActorSharedPointer Running::CreateAndRegisterActor(  const std::string& actorResourceFile,  uge::XMLElement\* pActorOverride)  {  // Create an actor.  uge::ActorSharedPointer pActor = CreateActor(actorResourceFile,  pActorOverride);  if (pActor == uge::ActorSharedPointer())  {  return uge::ActorSharedPointer();  }  // Add the actor to the physics simulation.  AddActorToPhysics(pActor);  // Allows querying an actor using its archetype.  m\_ActorNameToID[pActor->GetActorType()] = pActor->GetActorID();  return pActor;  }  void Running::RemoveActorFromPhysics(uge::ActorID actorID)  {  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vRemoveActor(actorID);  }  void Running::LoadProfile(const std::string& xmlResourceFilename)  {  uge::XMLFile entityListFile;  entityListFile.OpenFile(xmlResourceFilename,  uge::File::FileMode::FileReadOnly);  uge::XMLElement entityListRoot = entityListFile.GetRootElement();  for (uge::XMLElement entityElement =  entityListRoot.GetFirstChildElement();  entityElement.IsGood();  entityElement = entityElement.GetNextSiblingElement())  {  std::string entityName;  entityElement.GetAttribute("name", &entityName);  std::string entityResourceFileName;  entityElement.GetAttribute("resource",  &entityResourceFileName);  uge::XMLFile entityResource;  entityResource.OpenFile(entityResourceFileName,  uge::File::FileMode::FileReadOnly);  uge::ActorID actorID = m\_ActorNameToID[entityName];  m\_pGameLogic->vModifyActor(actorID, &entityResource.GetRootElement());  uge::ActorSharedPointer pActor =  m\_pGameLogic->vGetActor(actorID).lock();  pActor->PostInit();  // Re-add the actor to the physics simulation  // (its transform or shape might have changed).  RemoveActorFromPhysics(actorID);  AddActorToPhysics(pActor);  entityResource.CloseFile();  }  entityListFile.CloseFile();  } |

**Listing 92** uses the IGameState’s method vTailorToProfile() to adapt the game to the active profile. The method LoadProfile() does the tailoring: it reads a list of entities to (such as the one in **Listing 93**) and uses the Actor’s vModify() method to update its components or add new ones.

**Listing 93.** Tailoring the entities.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship"  resource=  "data/config/player\_profiles/average\_user/entity/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="50.0f" y ="50.0f" z="50.0"/>  </TransformableComponent>  <HealthComponent>  <InitialHealthPoints value="1000"/>  <MaximumHealthPoints value="1000"/>  </HealthComponent>  </Actor> |

For instance, **Listing 93** changes the Spaceship actor, increasing its size by 5 times the original size. It also makes it bulkier by raising its health points by 10 times (compare **Figure 38** to **Figure 39**).



**Figure 39.** The game logic actors, after being overridden by the player profile.

The first change might be used for making the game easier (for instance, when setting difficult levels). The second might provide larger entities to help low-vision users play the game.

It is also possible to add physical-interaction components to the entities at this point. Section 7.2.6 discusses and shows how to.

### Game View Layer

In Section 7.2.5, The Game Logic layer created and shaped the game world. That section used the Physics debugger rendering to show the game world, which is unsuitable for any serious game (or tutorial, for the matter). Although it evidenced an important point (an UGE game does not depends on the Game View layer), it does not diminish the importance of the Game View layer.

The Game View layer is actually the most important layer to the player interaction, as this provides IO specialization to the game. At one hand, it translates the player’s input into high-level game commands. At the other hand, it acts as a storyteller or a cinematographer: it captures the current game world state and describes it to the user. With game commands, the Game View layer it might even act as a bot and play the game with, against or for the player.

As this layer provides IO specialization, it also does offer accessibility barriers to the player. This section shows an example for implementing the IGameView abstract interface for a specific user profile. The same strategy allows the developers to create as many other IGameView’s implementation as they wish. This way, it is possible to suit different interaction abilities and to make the game suitable and accessible for a wider range of users.

#### Player Profile: General Settings

**Listing 94** shows the PlayerPreferences section of the player profile.

**Listing 94.** The PlayerPreferences section of the player profile, describing general settings.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <PlayerPreferences player\_name="user"  resource=  "data/config/player\_profiles/average\_user/preferences.xml">  <Language name="English" />  </PlayerPreferences> |

This section provides settings for the entire game application. At this moment, it is very simple and only contains a single element. Thus, it is expected to evolve in future versions of UGE.

Nevertheless, the only element of PlayerPreferences is an important one: the game language. Regardless of disabilities, the language is always an accessibility barrier: for any game that requires written or speech content comprehension, being unable to understand the language often makes the game impossible to a player.

#### Game Scene

UGE uses a scene graph to define the world game scene. To allow the use of different output renderers, there are two types of scenes: the Scene and the SceneRenderer. The Scene is a representation free scene containing only the hierarchical relationships between actors stored in scene nodes. It belongs to the Game Logic Layer (Section 7.2.5.4).

On the other hand, the SceneRenderer allows implementing different scene renderers to present the scene to the user. A scene renderer is an output engine, framework, SDK or API to convey the game data to the user through an output device. UGE has three different default implementations for scene renderer: OGRE for graphical rendering and OpenAL Soft and YSE for spatial (3D) audio playback.

To register an actor to the abstract Scene, the BaseGameLogic provides the method vCreateAndAddSceneNode() (called in vTailorToProfile() in **Listing 95**).

**Listing 95.** Adding actors to the game scene.

|  |
| --- |
| bool Running::vTailorToProfile(  const std::string& xmlResourceFilename)  {  // Overrides the presentation data using the chosen profile.  LoadProfile(xmlResourceFilename);  // Add actors to the scene.  //m\_pGameLogic->vCreateAndAddSceneNode(pPlane);  sg::GameLogic\* pGameLogic =  dynamic\_cast<sg::GameLogic\*>(m\_pGameLogic);  for (const auto& actorIt : pGameLogic->m\_Actors)  {  uge::ActorSharedPointer pActor = actorIt.second;    pGameLogic->vCreateAndAddSceneNode(pActor);  }    std::shared\_ptr<uge::EvtData\_Set\_Controlled\_Actor> pEvent(  LIB\_NEW uge::EvtData\_Set\_Controlled\_Actor(  m\_pSpaceship->GetActorID()));  uge::IEventManager::Get()->vQueueEvent(pEvent);  return true;  } |

This method creates a scene node from the Actor’s pointer and adds this node to the scene. Once an actor is added to the IO-free Scene, the engine creates the SceneNodeRenderers for the registered SceneRenderers automatically, provided the actor has the necessary output component (for instance, an OgreGraphicalComponent for the Ogre SceneRenderer). This is illustrated in Section 7.2.6.3.

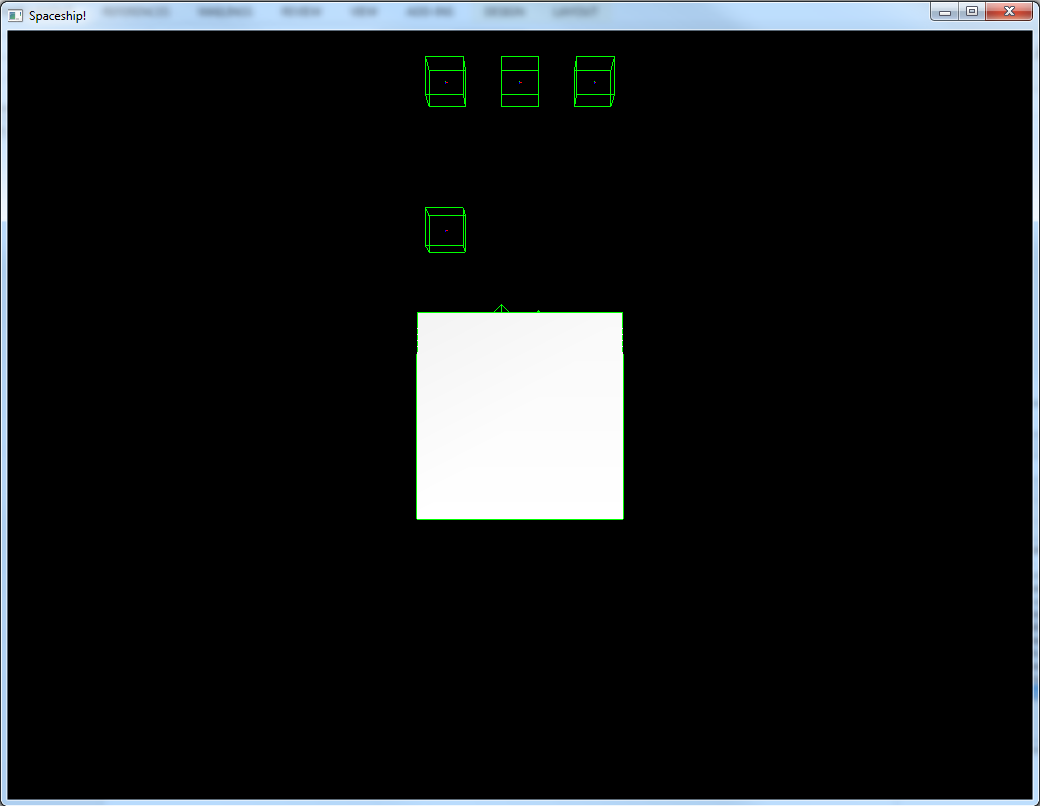
#### Player Profile: Entity Specializations

After registering a scene node to the Scene, it is possible to display or reproduce its representation in the output subsystems. To do this, it is necessary to add output components to the actors. **Listing 96** shows an example of how to attach a graphical component to the actor previously specialized in **Listing 93**.

**Listing 96.** Adding an output component to the actor.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource=  "data/config/player\_profiles/average\_user/entity/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  <Scale x="50.0f" y ="50.0f" z="50.0"/>  </TransformableComponent>  <HealthComponent>  <InitialHealthPoints value="500"/>  <MaximumHealthPoints value="500"/>  </HealthComponent>  <OgreGraphicalComponent>  <NodeName n="Spaceship-Graphics" />  <MeshFileName m="box.mesh" />  <MaterialFileName n="" />  </OgreGraphicalComponent>  </Actor> |

The spaceship actor finally has a graphical representation. **Figure 40** illustrates the result of using programmer’s art as the output representation.



**Figure 40.** The graphical actor.

By employing this same approach for all the actors, it is possible to specialize an entire profile output. If other profiles use different components, the game may have a completely different presentation – even with different modalities.

#### Creating the Camera

If a player uses an IGameView implementation, it will require a Camera and a GameController. In fact, the HumanGameView reference implementation forces its subclasses to define them using pure virtual methods (vCreateCamera() and vCreateController(), respectively).

To aid developers to create the Camera, UGE provides a Frustum class and a very simple follow-up camera. It is possible to attach the follow-up to an actor; doing so, UGE assumes the position of the actor’s front vector to present the game world from the actor’s perspective. Thus, it does not matter if the game implementation is audio only (such as with an AudioView, which is an oxymoron); it still requires a camera (although it would be more convenient to think of it as an ear).

The default HumanView implementation uses the vCreateCamera() method to determine the game’s camera (**Listing 97**).

**Listing 97.** The default camera used in the tutorial.

|  |
| --- |
| virtual uge::ICameraNodeSharedPointer vCreateCamera() override  {  uge::Frustum viewFrustum;  float fAspectRatio = 1024.0f / 768.0f;  viewFrustum.Init(45.0f, fAspectRatio, 5.0f, 10000.0f);  // fov, aspect ratio, near plane, far plane  const uge::Vector3 position(0.0f, 500.0f, 0.0f);  const uge::Vector3 rotation(-1.57f, 0.0f, 0.0f);  uge::Matrix4 cameraInitialTransform;  cameraInitialTransform.MakeRotationMatrix(  uge::Quaternion(rotation));  cameraInitialTransform.SetPositionFromVector(position);  uge::ICameraNodeSharedPointer pCamera(  LIB\_NEW uge::CameraNode(cameraInitialTransform, viewFrustum));  return pCamera;  } |

The previous sections used a top-view camera. It is possible to change its position and rotation to any values as desired. For first person games, such as audio-only games, UGE provides a simple follow-up camera implementation using the vSetTarget() method.

It is possible to call this method after the creation of the actor controlled by the player with the method vSetControlledActor() (**Listing 98**).

**Listing 98.** Defining a follow-up camera.

|  |
| --- |
| virtual void vSetControlledActor(uge::ActorID actorID, bool bSetCameraTarget) override  {  uge::HumanGameView::vSetControlledActor(actorID, bSetCameraTarget);  if (bSetCameraTarget)  {  m\_pCamera->vSetCameraOffset(  uge::Vector4(-10.0f, 2.0f, 0.0f, 0.0f));  m\_pCamera->vSetCameraOrientation(  -1.57f, -0.0f, 0.0f);  m\_pCamera->vSetTarget(  m\_SceneRenderManager.GetSceneNode(actorID));  }  } |

This method position all scene node renderers to convey the information to the player from the actor’s perspective defined by its front vector. This helps pre-setting different implementations for the game views. It is important to note, however, that this usually is usually not by itself – a stimulus usually have implicit feedback, which is not always obvious until it is completely removed from the game. For instance, the collision between a bomb and an alien might be clear in a graphical presentation; however, it requires a sound in an audio-only presentation. Game events are very helpful in disregard (they are discussed in Sections 7.2.6.7 and 7.2.6.8).

#### Creating the Controller

The HumanGameView also requires an implementation of a GameController. The GameController selects input devices to offer to the player and translates the received input into high-level game commands (Section 7.2.5.6).

UGE has the IInputDevice abstract interface, which has the purpose of abstracting input devices. This interface serves as a common base handler for all input devices – it is possible to create interfaces subclasses for input devices. For instance, two default device interfaces are available: Mouse and Keyboard. The default implementation for these abstract interfaces uses OIS.

**Listing 99** provides a simple GameController using a mouse and a keyboard input devices.

**Listing 99.** A GameController using mouse and keyboard.

|  |
| --- |
| #include <Engine/GameController/GameController.h>  #include <Core/PlayerProfile/GraphicalPreferences.h>  #include <IO/Input/InputDevice/Implementation/OIS/OISMouse.h>  #include <IO/Input/InputDevice/Implementation/OIS/OISKeyboard.h>  #include "InputTypes.h"  namespace sg  {  class GameController : public uge::GameController  {  public:  GameController(  const uge::GraphicalPreferences::WindowSettings&  windowSettings, size\_t windowHandle);  ~GameController();  virtual bool vInit() override;  virtual bool vDestroy() override;  virtual bool vUpdate(unsigned long timeElapsed) override;  void InputCallback(uge::InputMapping::MappedInput& inputs);  protected:  virtual const std::string  vGetInputContextListFilename() override;  virtual const  uge::InputMapping::RawInputToInputTypeCallbacks  vGetInputConverterDelegates() override;  virtual const std::vector<std::string>  vGetInputContexts() override;  virtual std::vector<  uge::GameController::MappedInputDelegate>  vGetMappedInputDelegates() override;  private:  uge::GraphicalPreferences::WindowSettings m\_WindowSettings;  size\_t m\_WindowHandle;  bool m\_LastPlayerMoveLeft;  bool m\_LastPlayerMoveRight;  uge::InputDevice::OISKeyboard m\_Keyboard;  uge::InputDevice::OISMouse m\_Mouse;  };  } |

The most important method of this class is InputCallback() (**Listing 100**). This function translates the user input into game commands. However, this is not done directly: it uses a low-level data-driven game command to allow input mapping (**Listing 101**). In fact, it is possible to note that **Listing 100** does not have any reference to a specific mouse or keyboard key.

**Listing 100.** A sample implementation of the InputCallback() method.

|  |
| --- |
| void GameController::InputCallback(uge::InputMapping::MappedInput& inputs)  {  bool bState = inputs.IsStateEnabled(  uge::InputMapping::State::PlayerMoveLeft);  if (bState != m\_LastPlayerMoveLeft)  {  if (bState)  {  std::shared\_ptr<sg::MoveActor> pEvent(  LIB\_NEW sg::MoveActor(  m\_ActorID, sg::MoveActor::Direction::Left));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  else  {  std::shared\_ptr<sg::StopActor> pEvent(  LIB\_NEW sg::StopActor(m\_ActorID));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  m\_LastPlayerMoveLeft = bState;  }  bState = inputs.IsStateEnabled(  uge::InputMapping::State::PlayerMoveRight);  if (bState != m\_LastPlayerMoveRight)  {  if (bState)  {  std::shared\_ptr<sg::MoveActor> pEvent(  LIB\_NEW sg::MoveActor(  m\_ActorID, sg::MoveActor::Direction::Right));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  else  {  std::shared\_ptr<sg::StopActor> pEvent(  LIB\_NEW sg::StopActor(m\_ActorID));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  m\_LastPlayerMoveRight = bState;  }  if (inputs.IsActionEnabled(uge::InputMapping::Action::Fire))  {  // Send fire event.  }  } |

**Listing 101.** Low-level game commands: actions, states and ranges.

|  |
| --- |
| #include <IO/Input/InputMapping/InputTypes.h>  enum class uge::InputMapping::Action : unsigned int  {  Fire,  };  enum class uge::InputMapping::State : unsigned int  {  PlayerMoveRight,  PlayerMoveLeft,  };  enum class uge::InputMapping::Range : unsigned int  {  }; |

This way, it is possible to change the control scheme binding without changing the GameController code (input remapping). To interact to the game, it is necessary to define an input mapping (described in Section 7.2.6.6).

#### Player Profile: Input and Input Mapping

There are three needed resources for input mapping: a list of input contexts (**Listing 102**), a list of low-level game commands (**Listing 103**) and the low-level command to physical interaction mapping (**Listing 104**).

**Listing 102.** A list of input contexts.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <InputContexts resource="data/game/commands/input\_context\_list.xml">  <InputContext name="game\_commands">  <Commands resource="data/game/commands/game\_commands.xml"/>  <InputMapping resource="data/game/commands/default\_command\_mapping.xml"/>  </InputContext>  </InputContexts> |

The list of input contexts allows defining different ways for interaction with the game in different contexts. For instance, the commands used during gameplay are usually different from the ones used during menu navigation. Each input context allows defining the low-level commands and inputting mapping for each context.

**Listing 103.** All the low-level game commands for a context.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource="data/game/commands/game\_commands.xml">  <Actions>  <Action name="Fire"/>  </Actions>  <States>  <State name="PlayerMoveLeft"/>  <State name="PlayerMoveRight"/>  </States>  <Ranges>  </Ranges>  </Commands> |

The list of low-level game commands (**Listing 103**) should match the code enumeration order (**Listing 101**). This allows the input mapping to be changed without changing the game code – as long as the order is the same, the mapping will work correctly.

**Listing 104.** The input mapping for the low-level game commands.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource="data/game/commands/default\_command\_mapping.xml">  <Actions>  <State game\_command\_name="Fire"  device="mouse" input\_value="MB\_LEFT"/>  </Actions>  <States>  <State game\_command\_name="PlayerMoveLeft"  device="keyboard" input\_value="KC\_LEFT"/>  <State game\_command\_name="PlayerMoveRight"  device="keyboard" input\_value="KC\_RIGHT"/>  </States>  <Ranges>    </Ranges>  </Commands> |

Finally, **Listing 104** provides the input mapping. This resource lists the devices and input values used for each game command. For instance, in **Listing 104**, the left arrow should be used to move the player actor to the left, as it is related to the PlayerMoveLeft low-level game command.

It is also possible to define input mappings for specific profiles (**Listing 105**).

**Listing 105.** User specific input mapping.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource=  "data/config/player\_profiles/average\_user/input\_mapping.xml">  <Actions>  <State game\_command\_name="Fire"  device="mouse" input\_value="MB\_LEFT"/>  </Actions>  <States>  <State game\_command\_name="PlayerMoveLeft"  device="keyboard" input\_value="KC\_A"/>  <State game\_command\_name="PlayerMoveRight"  device="keyboard" input\_value="KC\_D"/>  </States>  <Ranges>    </Ranges>  </Commands> |

This way, it is possible to define the best input devices and settings for each profile – or for each game user.

#### Registering View Events

The second approach to providing feedback to the users is using events. The same events defined in Section 7.2.5.5 are useful here. **Listing 106** contains a sample implementation of a class that plays sounds when an actor moves.

**Listing 106.** A class to provide aural feedback.

|  |
| --- |
| #pragma once  #include <Core/Events/IEventManager.h>  #include <Core/Resource/ResourceCache.h>  #include <Core/Resource/XMLResource.h>  #include <Core/Resource/ZipFileResource.h>  #include <Core/Script/Lua/ScriptResource.h>  #include <Engine/GameView/ViewFeedback/IViewFeedback.h>  #include <IO/Output/Audio/Audio.h>  #include <IO/Output/Audio/Implementation/OpenALSoft/OpenALSoftAudioResource.h>  #include "../../Logic/Events/GameEvents.h"  namespace sg  {  class AuralFeedback : public uge::IViewFeedback  {  public:  static const char\* g\_Name;  struct Options  {  Options()  : bEnableOnMoveActor(false),  bEnableOnStopActor(false)  {  }  bool bEnableOnMoveActor;  bool bEnableOnStopActor;  };  AuralFeedback()  : m\_Options(), m\_pAudio(), m\_pMovingAudioBuffer(nullptr)  {  }  ~AuralFeedback()  {  m\_pAudio.reset();  }  virtual bool vInit(  const std::string& resourceFileName) override  {  uge::XMLFile xmlFile;  xmlFile.OpenFile(resourceFileName,  uge::File::FileMode::FileReadOnly);  assert(xmlFile.IsGood()  && "XML file could not be opened!");  uge::XMLElement xmlRootElement(  xmlFile.GetRootElement());  assert(xmlRootElement.IsGood()  && "Invalid Root in XML File!");  AuralFeedback::Options afOptions;  uge::XMLElement eventsElement =  xmlRootElement.GetFirstChildElement("Events");  assert(eventsElement.IsGood()  && "Events element not found!");  for (uge::XMLElement xmlElement =  eventsElement.GetFirstChildElement("Event");  xmlElement.IsGood();  xmlElement = xmlElement.GetNextSiblingElement())  {  std::string eventName;  xmlElement.GetAttribute("name", &eventName);  bool bEventEnabled;  xmlElement.GetBoolAttribute("enabled",  &bEventEnabled);  if (eventName == "OnMoveActor")  {  afOptions.bEnableOnMoveActor = bEventEnabled;  }  else if (eventName == "OnStopActor")  {  afOptions.bEnableOnStopActor = bEventEnabled;  }  }  xmlFile.CloseFile();  RegisterDelegates(afOptions);  InitResourceCache("data/", 10);  return true;  }  virtual void vSetAudioSystem(uge::IAudioWeakPointer pAudio)  {  m\_pAudio = pAudio;  }  private:  void RegisterDelegates(const AuralFeedback::Options& options)  {  if (options.bEnableOnMoveActor)  {  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(  this, &AuralFeedback::OnMoveActor);  uge::IEventManager::Get()->vAddListener(  functionDelegate, sg::MoveActor::sk\_EventType);  }  if (options.bEnableOnStopActor)  {  uge::EventListenerDelegate functionDelegate =  fastdelegate::MakeDelegate(  this, &AuralFeedback::OnStopActor);  uge::IEventManager::Get()->vAddListener(  functionDelegate, sg::StopActor::sk\_EventType);  }  }  void InitResourceCache(  const std::string& fileName, unsigned int sizeMB)  {  uge::IResourceFile\* pResourceFile =  LIB\_NEW uge::ZipFileDevelopmentResource(  fileName, "./",  uge::ZipFileDevelopmentResource::Mode::Editor);  bool bSuccess = m\_ResourceCache.Init(  sizeMB, pResourceFile); // 10 MB  assert(bSuccess  && "Could not create the resource cache!");  uge::IResourceLoaderSharedPointer pCreateXMLLoader =  uge::XMLResourceLoader::CreateLoader();  m\_ResourceCache.RegisterLoader(pCreateXMLLoader);  uge::IResourceLoaderSharedPointer pCreateLuaScriptLoader  = uge::LuaScriptResourceLoader::CreateLoader();  m\_ResourceCache.RegisterLoader(pCreateLuaScriptLoader);  // OGG loader.  m\_ResourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(  LIB\_NEW uge::OpenALSoftOggResourceLoader));  // WAVE loader.  m\_ResourceCache.RegisterLoader(  std::shared\_ptr<uge::IResourceLoader>(  LIB\_NEW uge::OpenALSoftWaveResourceLoader));  }  uge::IAudioBuffer\* PlaySoundEffect(  const std::string& fileName,  float fVolume, bool bLoop,  const uge::Vector3& position = uge::Vector3::g\_Zero)  {  uge::Resource pResourceFile(fileName);  uge::ResourceHandleSharedPointer pResource =  m\_ResourceCache.GetHandle(&pResourceFile);  if (!m\_pAudio.expired())  {  uge::IAudioBuffer\* pAudioBuffer =  m\_pAudio.lock()->vInitAudioBuffer(pResource);  pAudioBuffer->vSetPosition(position);  pAudioBuffer->vPlay(fVolume, bLoop);  return pAudioBuffer;  }  return nullptr;  }  void OnMoveActor(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::MoveActor> pData =  std::static\_pointer\_cast<sg::MoveActor>(pEventData);  printf("[AuralFeedback] Actor %u started moving!\n",  pData->GetActorID());  PlaySoundEffect("data/audio/effects/sound.wav",  1.0f, false);  }  void OnStopActor(uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::StopActor> pData =  std::static\_pointer\_cast<sg::StopActor>(pEventData);  printf("[AuralFeedback] Actor %u stopped!\n",  pData->GetActorID());  }  private:  uge::IAudioWeakPointer m\_pAudio;  AuralFeedback::Options m\_Options;  uge::ResourceCache m\_ResourceCache;  uge::IAudioBuffer\* m\_pMovingAudioBuffer;  };  } |

It is possible to have several other classes similar to the one in **Listing 106**. To create one during run-time, a factory pattern is useful – UGE provides a ViewFeedbackFactory for this purpose (**Listing 107**).

**Listing 107.** A factory for providing event feedback.

|  |
| --- |
| #include "AuralFeedback.h"  namespace sg  {  class FeedbackFactory : public uge::ViewFeedbackFactory  {  public:  FeedbackFactory() { }  ~FeedbackFactory() { }  protected:  virtual void vInitFactory() override  {  m\_StateFactory.Register<sg::AuralFeedback>(  sg::AuralFeedback::g\_Name);  }  };  } |

**Listing 108** provides an example of how to use the factory in an IGameView implementation.

**Listing 108.** Initializing the aural feedback.

|  |
| --- |
| class HumanView : public uge::HumanGameView  {  public:  // ...  virtual bool vInit(uge::IScene\* pScene) override  {  if (!uge::HumanGameView::vInit(pScene))  {  return false;  }  RegisterEventDelegates();  // Rendering subsystems.  uge::OgreSceneRendererSharedPointer pOgreSceneRenderer(  LIB\_NEW uge::OgreSceneRenderer(  m\_pGraphics, m\_ResourceCache));  pOgreSceneRenderer->Load();  m\_GraphicalRendererID =  vAddSceneRenderer(pOgreSceneRenderer);  uge::OpenALSoftSceneRendererSharedPointer  pOpenALSoftSceneRenderer(  LIB\_NEW uge::OpenALSoftSceneRenderer(  m\_pAudio, m\_ResourceCache));  m\_AuralRendererID =  vAddSceneRenderer(pOpenALSoftSceneRenderer);  m\_FeedbackFactory.Init();  SetEventFeedback();  return true;  }  // ...  private:  void SetEventFeedback()  {  uge::GameplayPreferences::GameplaySettings  gameplaySettings = m\_PlayerProfile.  GetGameplayPreferences().GetGameplaySettings();  const auto& specializations =  gameplaySettings.eventSpecializationFileNames;  for (const auto& specialization : specializations)  {  std::string name = specialization.first;  std::string resource = specialization.second;  m\_pViewFeedback =  m\_FeedbackFactory.CreateViewFeedback(name);  assert(m\_pViewFeedback != nullptr && "Invalid event specializaton!");  m\_pViewFeedback->vInit(resource);  m\_pViewFeedback->vSetAudioSystem(m\_pAudio);  m\_pViewFeedback->vSetGraphicsSystem(m\_pGraphics);  }  }  private:  // ...  PongFeedbackFactory m\_FeedbackFactory;  uge::IViewFeedback\* m\_pViewFeedback;  }; |

This factory can be configured with a profile, as shown in Section 7.2.6.8.

#### Player Profile: Event Specializations

The events from the IGameView can also benefit from the player profile. **Listing 109** defines a list of all the event specializations that should be used in a profile.

**Listing 109.** A list of event feedback resources.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <EventSpecializations resource=  "data/config/player\_profiles/average\_user/events/events.xml">  <EventSpecialization name="AuralFeedback" resource=  "data/config/player\_profiles/average\_user/events/aural\_events.xml"/>    </EventSpecializations> |

**Listing 110** shows an example of one feedback configuration’s file. The developers are free to define the format, setting elements to the required data.

**Listing 110**. A sample configuration file for event feedback.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <EventSpecialization resource=  "data/config/player\_profiles/average\_user/events/aural\_events.xml">  <Events>  <Event name="OnMoveActor" enabled="true">  <!--<FileName n="data/audio/effects/sound.wav" />  <Volume v="1.0f" />  <InitialProgress p="0.0f" />  <Loop l="true" />-->  </Event>  <Event name="OnStopActor" enabled="true"/>  </Events>    </EventSpecialization> |

For instance, **Listing 110** allows enabling and disabling the feedback from an event without recompiling the code. The commented code offers another possibility: to set the sound file and parameters from within the configuration file – this would have the same benefits to an event as an audio component has to actors.

### Defining the Gameplay

Section 7.2.5 outlined how to use UGE’s features to implement a game. It is important to note, however, the implementation does not have any gameplay code so far – currently it is only possible to move the player spaceship. This section works on this issue to turn the basic project structure obtained in the previous sections into a playable game prototype.

#### Firing Projectiles

Before creating the code to fire a bullet or a bomb, it is necessary to define an event for the Game Logic layer. As the player should be able to fire projectiles, the event should be a game command. Section 7.2.5.6 suggested a FireProjectile game command. **Listing 111** shows a possible implementation for the high-level game command.

**Listing 111.** A game command to fire projectiles (bullets and bombs).

|  |
| --- |
| class FireProjectile : public uge::BaseEventData  {  public:  enum class Type : char  {  Bullet,  Bomb  };  static const uge::EventType sk\_EventType;  explicit FireProjectile(uge::ActorID actorID,  FireProjectile::Type type)  : m\_ActorID(actorID), m\_Type(type)  {  }  virtual const uge::EventType& vGetEventType() const override  {  return sk\_EventType;  }  virtual uge::IEventDataSharedPointer vCopy() const override  {  return uge::IEventDataSharedPointer(  LIB\_NEW FireProjectile(m\_ActorID, m\_Type));  }  virtual void vSerialize(std::ostrstream& out) const override  {  out << m\_ActorID << static\_cast<char>(m\_Type);  }  virtual void vDeserialize(std::istrstream& in) override  {  char directionValue;  in >> m\_ActorID >> directionValue;  m\_Type = static\_cast<FireProjectile::Type>(directionValue);  }  virtual const char\* vGetName() const override  {  return "FireProjectile";  }  uge::ActorID GetActorID() const  {  return m\_ActorID;  }  FireProjectile::Type GetType() const  {  return m\_Type;  }  private:  uge::ActorID m\_ActorID;  FireProjectile::Type m\_Type;  };  const uge::EventType FireProjectile::sk\_EventType(0xd8c91a61); |

The implementation is very similar to the MoveActor game command from **Listing 88**. It might seems strange to add an ActorID as an attribute to the game command. However, as it will be described in Section 7.2.7.4, this allows using the FireProjectile event for enemies as well.

After creating the high-level game command, it is necessary to allow the game logic to handle it according to its purpose – that is, spawning the desired projectile (**Listing 112**).

**Listing 112.** Creating and adding the projectile to the game world.

|  |
| --- |
| void Running::RegisterEvents()  {  // Creating and registering the event handlers.  // ...    functionDelegate =  fastdelegate::MakeDelegate(this, &Running::FireProjectile);  uge::IEventManager::Get()->vAddListener(  functionDelegate, sg::FireProjectile::sk\_EventType);  }  void Running::UnregisterEvents()  {  // Removing the delegates.  // ...  functionDelegate =  fastdelegate::MakeDelegate(this, &Running::FireProjectile);  uge::IEventManager::Get()->vRemoveListener(  functionDelegate, sg::FireProjectile::sk\_EventType);  }  // …  void Running::FireProjectile(  uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<sg::FireProjectile> pData =  std::static\_pointer\_cast<sg::FireProjectile>(pEventData);  // Create the desired projectile.  std::string actorResourceFile("");  if (pData->GetType() == FireProjectile::Type::Bullet)  {  actorResourceFile = "data/game/actors/bullet.xml";  }  else  {  actorResourceFile = "data/game/actors/bomb.xml";  }  uge::ActorSharedPointer pActorProjectile =  CreateActor(actorResourceFile);  assert(pActorProjectile != uge::ActorSharedPointer());  // Set position.  uge::ActorSharedPointer pActorOwner =  m\_pGameLogic->vGetActor(pData->GetActorID()).lock();  uge::Component::TransformableComponentSharedPointer  pOwnerTransformableComponent =  pActorOwner->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  const uge::Vector3& ownerPosition =  pOwnerTransformableComponent->GetPosition();  const uge::Vector3& ownerRotation =  pOwnerTransformableComponent->GetRotationVector();  const uge::Vector3& ownerScale =  pOwnerTransformableComponent->GetScale();  uge::Vector3 projectilePosition = ownerPosition;  // Account for half size (center).  projectilePosition.z -= (2.0f \* ownerScale.z);  uge::Vector3 projectileRotation = ownerRotation;  // Force a straight shot.  projectileRotation.z += (100.0f \* ownerScale.z);  uge::Vector3 direction = projectileRotation;  // Reverse the vector's sense.  direction.z \*= -1.0f;  uge::Component::TransformableComponentSharedPointer  pProjectileTransformableComponent =  pActorProjectile->GetComponent<  uge::Component::TransformableComponent>(  uge::Component::TransformableComponent::g\_ComponentName).lock();  pProjectileTransformableComponent->SetRotation(  projectileRotation);  pProjectileTransformableComponent->SetPosition(  projectilePosition);  // Add the actor to the physics simulation.  AddActorToPhysics(pActorProjectile);  m\_ActorNameToID[pActorProjectile->GetActorType()] =  pActorProjectile->GetActorID();  // Add the actor to the scene.  m\_pGameLogic->vCreateAndAddSceneNode(pActorProjectile);  // Set velocity and apply the initial impulse.  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vApplyForce(pActorProjectile->GetActorID(),  direction.GetNormalized(), 1.0f);  uge::Component::BulletPhysicsComponentSharedPointer  pActorPhysicsComponent = pActorProjectile->GetComponent<  uge::Component::BulletPhysicsComponent>(  uge::Component::BulletPhysicsComponent::g\_ComponentName).lock();  float fMaxVelocityMagnitude =  pActorPhysicsComponent->vGetMaxVelocity();  uge::Vector3 velocity = pActorPhysicsComponent->vGetVelocity();  float fCurrentVelocityMagnitude = velocity.Length();  velocity \*= (fMaxVelocityMagnitude / fCurrentVelocityMagnitude);  pActorPhysicsComponent->vSetVelocity(velocity);  } |

It is possible to test the game command before mapping it to an input by dispatching the event during any time of the game (such as in **Listing 90** for the MoveActor game command). This is outlined in the temporary code of **Listing 113**. As the code will be executed every game tick, **Listing 113** will fill the game world with random bullets and bombs.

**Listing 113.** Dispatching the game command’s event triggers its execution in the game logic.

|  |
| --- |
| bool Running::vOnUpdate(unsigned long timeElapsed)  {  bool bSuccess = uge::GameState::Running::vOnUpdate(timeElapsed);  RemoveDestroyedActors();  FireProjectile::Type type = (std::rand() % 2) ?  FireProjectile::Type::Bullet :  FireProjectile::Type::Bomb;  std::shared\_ptr<sg::FireProjectile> pEvent(  LIB\_NEW sg::FireProjectile(  m\_pSpaceship->GetActorID(), type));  uge::IEventManager::Get()->vQueueEvent(pEvent);  return bSuccess;  } |

As in Section 7.2.6.6, it is necessary to map the game command to physical-level interaction a game input. In order to do this, it is necessary to create the low-level game commands (**Listing 114**).

**Listing 114.** Register the new game commands to their low-level representation.

|  |
| --- |
| enum class uge::InputMapping::Action : unsigned int  {  FireBullet,  FireBomb,  };  enum class uge::InputMapping::State : unsigned int  {  PlayerMoveRight,  PlayerMoveLeft,  };  enum class uge::InputMapping::Range : unsigned int  {  }; |

Then, it is necessary to register the game commands in **Listing 103** (to register the low-level game commands), **Listing 104** (to define the input-mapping) and **Listing 105** (to define a input mapping to a specific profile) according to the names given in **Listing 114**. This is illustrated in **Listing 115**, **Listing 116** and **Listing 117**, respectively.

**Listing 115.** Register the new low-level game commands.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource="data/game/commands/game\_commands.xml">  <Actions>  <Action name="FireBullet"/>  <Action name="FireBomb"/>  </Actions>  <States>  <State name="PlayerMoveLeft"/>  <State name="PlayerMoveRight"/>  </States>  <Ranges>  </Ranges>  </Commands> |

It is interesting to note that, in **Listing 115**, a single command was split in two: FireBullet and FireBomb.

**Listing 116.** Default input mapping with the new game commands.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource="data/game/commands/default\_command\_mapping.xml">  <Actions>  <State game\_command\_name="FireBullet"  device="mouse" input\_value="MB\_LEFT"/>  <State game\_command\_name="FireBomb"  device="mouse" input\_value="MB\_RIGHT"/>  </Actions>  <States>  <State game\_command\_name="PlayerMoveLeft"  device="keyboard" input\_value="KC\_LEFT"/>  <State game\_command\_name="PlayerMoveRight"  device="keyboard" input\_value="KC\_RIGHT"/>  </States>  <Ranges>  </Ranges>  </Commands> |

**Listing 117.** Profile specific input mapping with the new game commands.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Commands resource=  "data/config/player\_profiles/average\_user/input\_mapping.xml">  <Actions>  <State game\_command\_name="FireBullet"  device="mouse" input\_value="MB\_LEFT"/>  <State game\_command\_name="FireBomb"  device="mouse" input\_value="MB\_RIGHT"/>  </Actions>  <States>  <State game\_command\_name="PlayerMoveLeft"  device="keyboard" input\_value="KC\_LEFT"/>  <State game\_command\_name="PlayerMoveRight"  device="keyboard" input\_value="KC\_RIGHT"/>  </States>  <Ranges>  </Ranges>  </Commands> |

Finally, in order to enable the game commands to the user, it is necessary to update the GameController (Section 7.2.6.5). This time, as both low-level commands are Actions instead of States instead of states, it is not necessary to store a variable with state data – it is only necessary to dispatch the event (**Listing 118**) according to the low-level game command names (defined in **Listing 114**).

**Listing 118.** Updating the InputCallback() to add the new game events.

|  |
| --- |
| void GameController::InputCallback(uge::InputMapping::MappedInput& inputs)  {  // ...  if (inputs.IsActionEnabled(  uge::InputMapping::Action::FireBullet))  {  std::shared\_ptr<sg::FireProjectile> pEvent(  LIB\_NEW sg::FireProjectile(  m\_ActorID, FireProjectile::Type::Bullet));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  if (inputs.IsActionEnabled(uge::InputMapping::Action::FireBomb))  {  std::shared\_ptr<sg::FireProjectile> pEvent(  LIB\_NEW sg::FireProjectile(  m\_ActorID, FireProjectile::Type::Bomb));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  } |

Before testing the new code, it is interesting to tweak the entities and remove obsolete ones. For instance, CreateGameActors() does not need to create sample projectiles anymore (**Listing 119**).

**Listing 119.** Removing obsolete code.

|  |
| --- |
| bool Running::CreateGameActors()  {  m\_pSpaceship =  CreateAndRegisterActor("data/game/actors/spaceship.xml");  if (m\_pSpaceship == uge::ActorSharedPointer())  {  return false;  }  uge::ActorSharedPointer pActor;  uge::XMLFile aliens;  aliens.OpenFile("data/game/actors/aliens.xml",  uge::File::FileMode::FileReadOnly);  uge::XMLElement xmlRoot = aliens.GetRootElement();  for (uge::XMLElement xmlElement =  xmlRoot.GetFirstChildElement("Actor");  xmlElement.IsGood();  xmlElement = xmlElement.GetNextSiblingElement())  {  pActor = CreateAndRegisterActor(  "data/game/actors/alien.xml", &xmlElement);  if (pActor == uge::ActorSharedPointer())  {  return false;  }  }  aliens.CloseFile();  return true;  } |

It is also a good time to set a reasonable size for the player’s spaceship and change its position and orientation to make it face the enemies (**Listing 120**).

**Listing 120.** The new spaceship archetype.

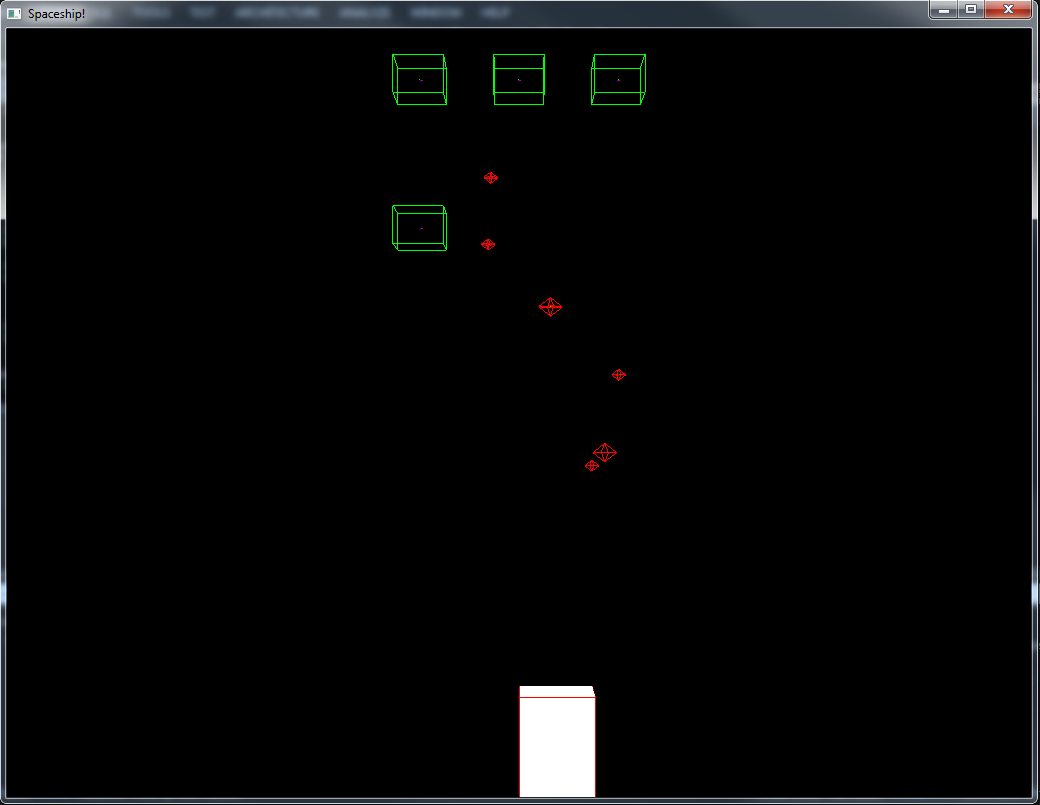
|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource="data/game/actors/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="3.1415f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <MoveableComponent>  <Velocity vcs="0.0f" vdx="0.0f" vdy="0.0f" vdz="0.0f"/>  <Acceleration ax="0.0f" ay="0.0f" az="0.0f"/>  </MoveableComponent>  <CollidableComponent>  <Shape type="Box">  <Dimension x="1.0f" y="1.0f" z="1.0f"/>  </Shape>  <CenterOfMassOffset>  <Position x="0.0f" y="0.0f" z="0.0f"/>  <Rotation yaw="0.0f" pitch="0.0f" roll="0.0f"/>  </CenterOfMassOffset>  <Density type="Pine"/>  <Material type="Elastic"/>  </CollidableComponent>  <BulletPhysicsComponent>  <LinearFactor x="1.0f" y="1.0f" z="1.0f"/>  <AngularFactor x="0.0f" y="0.0f" z="0.0f"/>  <!--<InitialVelocity x="0.0f" y="0.0f" z="0.0f">-->  <MaxVelocity v="15.0f"/>  <MaxAngularVelocity v="0.0f"/>  <!--<MaxAcceleration a="0.0f"/>-->  </BulletPhysicsComponent>  <HealthComponent>  <InitialHealthPoints value="100"/>  <MaximumHealthPoints value="100"/>  </HealthComponent>  <DamageSoakingComponent>  <InitialProtection value="50"/>  <MaximumProtection value="100"/>  </DamageSoakingComponent>  </Actor> |

After changing the stereotype, it might be necessary to update the existing profile specializations (**Listing 121**).

**Listing 121.** The physical-level specialization for the average user profile.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <Actor type="Spaceship" resource=  "data/config/player\_profiles/average\_user/entity/spaceship.xml">  <TransformableComponent>  <Position x="0.0f" y="0.0f" z="180.0f"/>  <!-- YXZ order (yaw, pitch, roll) -->  <Rotation yaw="3.1415f" pitch="0.0f" roll="0.0f"/>  <Scale x="10.0f" y ="10.0f" z="10.0"/>  </TransformableComponent>  <HealthComponent>  <InitialHealthPoints value="500"/>  <MaximumHealthPoints value="500"/>  </HealthComponent>  <OgreGraphicalComponent>  <NodeName n="Spaceship-Graphics" />  <MeshFileName m="box.mesh" />  <MaterialFileName n="" />  </OgreGraphicalComponent>  </Actor> |

After compiling and running the code, it should be possible to fire bullets using the left mouse button and to fire bombs using the right button (**Figure 41**).



**Figure 41.** Firing some projectiles.

However, as there is no logic for the hit, the enemies only move due to the collision. Next section shows how to fix this.

#### Attacking the Enemies

Section 7.2.5.5 implemented a very basic collision event handler. The handlers CollisionStarted and CollisionEnded served only illustrative purposes so far, as there was nothing to collide on the existing game actors – unless the collision was forced in the components initial data.

With Section 7.2.7.1’s projectiles, this changed. The user’s spaceship became able to fire bullets and bombs on other actors. However, as there is no collision handling game code, the Physics subsystem handled the collision itself, updating the actors’ transforms and velocities values.

This section shows how to handle the collision for gameplay purposes. It is time to use HealthComponent, DamageSoakingComponent and DamageInflictingComponent in the game. The projectile collision with another actor involves all these components, updating their data values and, consequently, the game logic.

**Listing 122** presents a possible implementation for handling the collision.

**Listing 122.** Handling the collision between an actor and a projectile.

|  |
| --- |
| static const std::string g\_kActorTypeAlien = "Alien";  static const std::string g\_kActorTypeBomb = "Bomb";  static const std::string g\_kActorTypeBullet = "Bullet";  static const std::string g\_kActorTypeSpaceship = "Spaceship";  void Running::CollisionStarted(  uge::IEventDataSharedPointer pEventData)  {  std::shared\_ptr<uge::EvtData\_PhysCollision> pData =  std::static\_pointer\_cast<uge::EvtData\_PhysCollision>(pEventData);  printf("Actors %u and %u collided!\n",  pData->GetActorA(), pData->GetActorB());    uge::ActorID actorA = pData->GetActorA();  uge::ActorID actorB = pData->GetActorB();    uge::ActorSharedPointer pActorA =  m\_pGameLogic->vGetActor(actorA).lock();  uge::ActorSharedPointer pActorB =  m\_pGameLogic->vGetActor(actorB).lock();  const std::string actorTypeA = pActorA->GetActorType();  const std::string actorTypeB = pActorB->GetActorType();  if (actorTypeA == g\_kActorTypeBomb ||  actorTypeA == g\_kActorTypeBullet)  {  if (actorTypeB == g\_kActorTypeAlien)  {  HandleProjectileCollision(pActorB, pActorA);  }  else if (actorTypeB == g\_kActorTypeSpaceship)  {  HandleProjectileCollision(pActorB, pActorA);  }  }  else if (actorTypeB == g\_kActorTypeBomb ||  actorTypeB == g\_kActorTypeBullet)  {  if (actorTypeA == g\_kActorTypeAlien)  {  HandleProjectileCollision(pActorA, pActorB);  }  else if (actorTypeA == g\_kActorTypeSpaceship)  {  HandleProjectileCollision(pActorA, pActorB);  }  }  }  void Running::HandleProjectileCollision(  uge::ActorSharedPointer pTarget,  uge::ActorSharedPointer pProjectile)  {  // All projectiles should have a DamageInflictingComponent.  sg::Component::DamageInflictingComponentSharedPointer  pDamageComponent =  pProjectile->GetComponent<  sg::Component::DamageInflictingComponent>(  sg::Component::DamageInflictingComponent::g\_ComponentName).  lock();  int totalDamage = pDamageComponent->GetDamageOutput();  // A DamageSoaking is optional, so we have to  // check before converting to a shared pointer (lock()).  sg::Component::DamageSoakingComponentWeakPointer  pWeakSoakingComponent = pTarget->GetComponent<  sg::Component::DamageSoakingComponent>(  sg::Component::DamageSoakingComponent::g\_ComponentName);  if (!pWeakSoakingComponent.expired())  {  sg::Component::DamageSoakingComponentSharedPointer  pSoakingComponent = pWeakSoakingComponent.lock();  int shieldProtection =  pSoakingComponent->GetCurrentProtection();  if (shieldProtection > 0)  {  pSoakingComponent->DecrementProtection(totalDamage);  totalDamage -= shieldProtection;  printf("The shield reduced the damage by: %d!\n",  pDamageComponent->GetDamageOutput() - shieldProtection);  }  }  // All targets should have a HealthComponent.  sg::Component::HealthComponentSharedPointer  pTargetHeathComponent = pTarget->GetComponent<  sg::Component::HealthComponent>(  sg::Component::HealthComponent::g\_ComponentName).  lock();  if (totalDamage > 0)  {  pTargetHeathComponent->DecrementHealthPoints(totalDamage);  printf("The actor took %d points of damage!\n", totalDamage);  }  } |

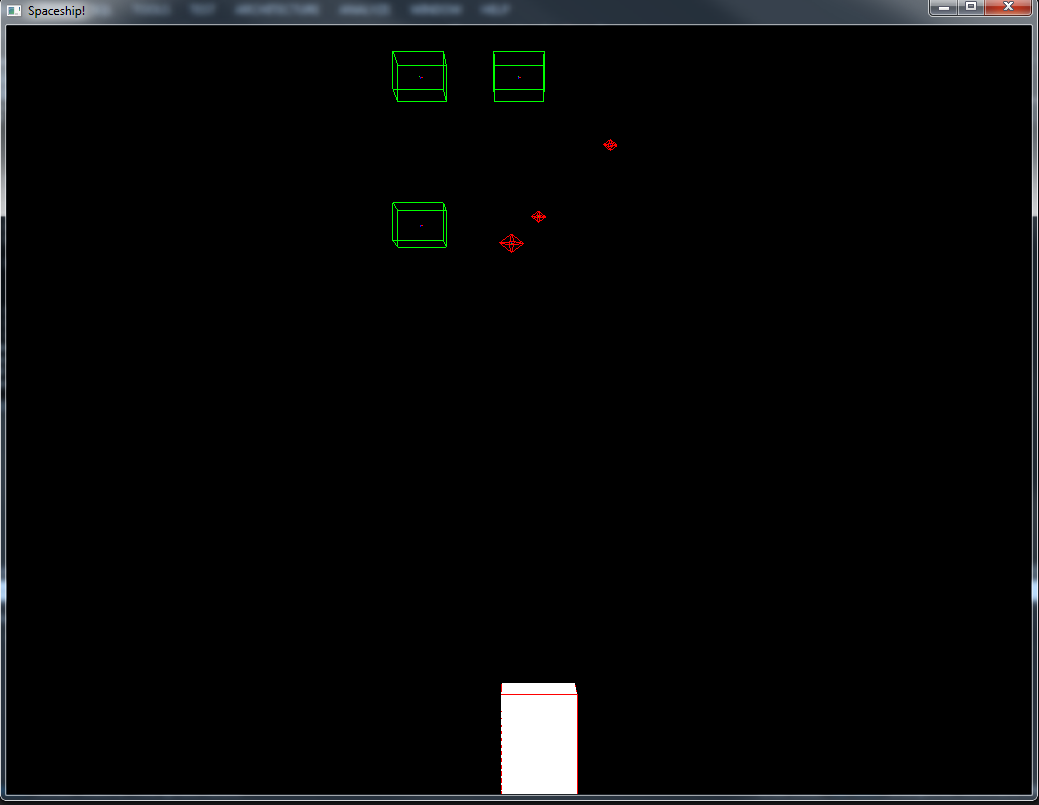
In **Listing 122**, CollisionStarted() is an event handler for the start of the collision. When two actors collide, the implementation verifies whether one of them is a projectile. It only one actor is a projectile, the verifications continue: the other actor must be either the enemies (aliens) or the player’s spaceship (this same implementation is used in Section 7.2.7.4). If both conditions are true, HandleProjectileCollision() updates the components, decreasing the protection of actor’s shields (DamageSoakingComponent) if any, and the remaining health points (HealthComponent) by the damage (DamageInflictionComponent). If, after the collision, the HealthComponent points become zero or less, the actor will be removed in the vUpdate()’s RemoveDestroyedActors() method.

There are two important omissions in **Listing 122**: after a collision, the bullets are not yet removed and both the bullets and actors keep moving. This is fixed in **Listing 123**.

**Listing 123.** Handling the side effects of the collision.

|  |
| --- |
| void Running::HandleProjectileCollision(  uge::ActorSharedPointer pTarget,  uge::ActorSharedPointer pProjectile)  {  // ...  // Remove the projectile from the game.  const uge::ActorID projectileID = pProjectile->GetActorID();  pPhysics->vRemoveActor(projectileID);  m\_pGameLogic->vRemoveSceneNode(projectileID);  m\_pGameLogic->vDestroyActor(projectileID);  } |

After compiling and running the code, **Figure 42** illustrates a possible game situation after a few game commands. In this figure, the right most enemy from **Figure 41** is dead.



**Figure 42.** Shooting some projectiles at the enemies.

Sections 7.2.7.2 and 7.2.7.3 add simple behavior to control the enemies. They make use of the game commands defined so far to ease the implementation.

#### Moving the Enemies

Section 7.2.5.6.1 defined a high-level game command to move actors (**Listing 88**). As that section suggested, it would be possible to use that same event to move other actors, such as the enemy. This section explores this possibility in **Listing 124**.

**Listing 124.** Moving the game enemies using the game command from **Listing 88**.

|  |
| --- |
| void Running::MoveEnemies()  {  // Move the enemies in a random sense  // with probability kProbabilityToMove.  sg::GameLogic\* pGameLogic =  dynamic\_cast<sg::GameLogic\*>(m\_pGameLogic);  for (const auto& actorIt : pGameLogic->m\_Actors)  {  uge::ActorID actorID = actorIt.first;  uge::ActorSharedPointer pActor = actorIt.second;  if (pActor->GetActorType() != g\_kActorTypeAlien)  {  continue;  }  const unsigned int kProbabilityToMove = 5;  const unsigned int kProbabilityToStop = 70;  if (std::rand() % 100 > kProbabilityToMove)  {  continue;  }  else if (std::rand() % 100 > kProbabilityToStop)  {  uge::IPhysicsSharedPointer pPhysics =  m\_pGameLogic->vGetPhysics();  pPhysics->vStopActor(actorID);  continue;  }  MoveActor::Direction direction = (std::rand() % 2) ?  MoveActor::Direction::Left :  MoveActor::Direction::Right;  std::shared\_ptr<sg::MoveActor> pEvent(  LIB\_NEW sg::MoveActor(actorID, direction));  uge::IEventManager::Get()->vQueueEvent(pEvent);  }  } |

**Listing 124** uses the same random movement implementation from **Listing 90**. It is possible to improve the AI with some basic heuristic – such as moving closer the player’s spaceship –, or using a more sophisticated technique – perhaps a decision tree using Lua.

#### Allowing Enemies to Attack the Player

### Tailoring the Game to Different Interaction Abilities

#### Defining a New Profile

#### Choosing a Profile

#### Implementing the Profile

### Results and Discussion

# References

1. Fox, M. Game Engines 101: The Entity/Component Model. *Gamasutra*, 2010. http://www.gamasutra.com/blogs/MeganFox/20101208/6590/Game\_Engines\_101\_The\_EntityComponent\_Model.php.

2. Gamadu.com. Artemis Entity System Framework. 2012. http://gamadu.com/artemis/.

3. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

4. Garcia, F.E. and Neris, V.P. de A. Design de Jogos Universais: Apoiando a Prototipação de Alta Fidelidade com Classes Abstrata. *Proceedings of the 12th Brazilian Symposium on Human Factors in Computing Systems*, Brazilian Computer Society (2013), 82–91.

5. Garcia, F.E. and Neris, V.P. de A. Design Guidelines for Audio Games. In M. Kurosu, ed., *Human-Computer Interaction. Applications and Services*. Springer Berlin Heidelberg, 2013, 229–238.

6. Garcia, F.E. and Neris, V.P. de A. A Data-Driven Entity-Component Approach to Develop Universally Accessible Games. (2014).

7. Grammenos, D., Savidis, A., Georgalis, Y., and Stephanidis, C. Access Invaders: Developing a Universally Accessible Action Game. In K. Miesenberger, J. Klaus, W.L. Zagler and A.I. Karshmer, eds., *Computers Helping People with Special Needs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, 388–395.

8. Grammenos, D., Savidis, A., and Stephanidis, C. Unified design of universally accessible games. *Proceedings of the 4th international conference on Universal access in human-computer interaction: applications and services*, Springer-Verlag (2007), 607–616.

9. Grammenos, D., Savidis, A., and Stephanidis, C. Designing universally accessible games. *Magazine Computers in Entertainment (CIE) - SPECIAL ISSUE: Media Arts and Games 7*, (2009), 29.

10. Grammenos, D., Savidis, A., and Stephanidis, C. Unified Design of Universally Accessible Games. In C. Stephanidis, ed., *Universal Access in Human-Computer Interaction. Applications and Services*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, 607–616.

11. Gregory, J. *Game Engine Architecture*. A K Peters, 2009.

12. McShaffry, M. and Graham, D. *Game Coding Complete, Fourth Edition*. Course Technology PTR, 2012.

13. McShaffry, M. *Game Coding Complete, Third Edition*. Charles River Media, 2009.

14. Nystrom, R. Component. *Game Programming Patterns*, 2012. http://gameprogrammingpatterns.com/component.html.

15. Pallister, K. *Game Programming Gems 5*. Charles River Media, Hingham, Mass., 2005.

16. Scott Bilas. A Data-Driven Game Object System. 2002. http://scottbilas.com/games/dungeon-siege/.

17. Story, M.F., Mueller, J.L., and Mace, R.L. *The Universal Design File: Designing for People of All Ages and Abilities. Revised Edition.* 1998.

1. Unity. Available at: <<http://unity3d.com/>>. [↑](#footnote-ref-1)
2. GamePlay. Available at: <<https://github.com/blackberry/GamePlay>>. [↑](#footnote-ref-2)
3. GCC4. Available at: <<http://www.mcshaffry.com/GameCode/>>. [↑](#footnote-ref-3)
4. OGRE. Available at: <<http://www.ogre3d.org/>>. [↑](#footnote-ref-4)
5. YSE. Available at: <<http://attr-x.net/yse/> >. [↑](#footnote-ref-5)
6. TinyOAL. Available at: <<https://code.google.com/p/tinyoal/>>. [↑](#footnote-ref-6)
7. OIS. Available at: <<http://sourceforge.net/projects/wgois/>>. [↑](#footnote-ref-7)
8. Bullet Physics. Avaiable at <<http://bulletphysics.org>>. [↑](#footnote-ref-8)
9. Arguably, it is possible to considerer pure aesthetical entities as well. However, the proposed definition simplifies relating an entity to the entity-component model. [↑](#footnote-ref-9)
10. This is necessary as UGE is implemented in C++, a strongly typed language which lacks Run-Time Type Information (RTTI). [↑](#footnote-ref-10)
11. Here it might not be a good idea to create automatic movement for the player, as simple Ping-Pong games usually restricts the player interaction to moving the paddle. [↑](#footnote-ref-11)
12. A complete description of the game design and the Unified Design framework is available at <<http://www.gamasutra.com/view/feature/1764/unified_design_of_universally_.php>>. [↑](#footnote-ref-12)
13. It is not necessary to define all the components before creating the game, though – in fact, the tutorial game might need many other components. The entity-component model provides great flexibility to the implementation as it is always possible to create new components add attach them to actors, or to modify existing ones. [↑](#footnote-ref-13)
14. The components could have simpler names, such as WeaponComponent or ShieldComponent. It is important to remember, however, that components are not entities. A WeaponComponent is not a weapon – it allows its entity owner to deal damage. Thus, a Laser actor that has an attached WeaponComponent would be able to deal damage. [↑](#footnote-ref-14)
15. It is not necessary to recompile, as this example explores the data-driven architecture of the engine. As the actor stereotype is an external resource (a XML file), adding or removing initial components only requires changing the contents of the file. [↑](#footnote-ref-15)
16. It looks as a diamond at this angle. It, however, represents a sphere. [↑](#footnote-ref-16)
17. It is important to note the final game should use appropriate art assets instead of the Physics debugger simple representation. The debugger is being used here to illustrate how to develop the game without physical-level interaction references, such as 3D models for graphical stimuli. [↑](#footnote-ref-17)
18. In other words, it does not have a shield. [↑](#footnote-ref-18)