MotorDead (codename FightRace)
Technical Documentation

Jakub Kýpeť, Matej Marko, Jan Navrátil, Šimon Soták
Supervisor: Jan Beneš
http://www.motordead.com/

March 18, 2013

Contents

Chapter 1

Introduction

This document contains the programming documentation of the game MotorDead (codename FightRace). MotorDead is an action racing game inspired by DeathRally[1]. It aims to offer a fun, dynamic and immersive gaming experience.

The primary goal of this document is to familiarize the reader with the technical design of the project, to the extent of being able to orient himself in the systems of the game and their source code.

Our secondary goal is to outline the various obstacles we encountered during the development of MotorDead and discuss the non-trivial decisions that arose from these problems.

1.1 Team

MotorDead was developed as part of the Software Project course taught at the Department of Mathematics and Physics at the Charles University in Prague. However, considering the nature of such a project, its requirements exceed the competence of computer science students and therefore we collaborated with people outside of Charles Univesity, mainly for design and content-creation purposes. In the end, as much as 10 people contributed to the development.

1.1.1 MFF team members

Jan Beneš
: Project supervisor. He participated at our weekly meetings, keeping the big picture realistic, warning us of feature creep and doing sanity checks on our time prognoses. He also helped with planning and time management.

Šimon Soták
: Team leader and programmer. He spent at least half the time on management, organization, communication and maintaining a cohesion between team members. He implemented the particle effects system, some game logic and other features.

[1] `http://en.wikipedia.org/wiki/Death_Rally`

Matej Marko
Lead programmer and physics expert. He probably did more programming than the rest of the MFF portion of the team combined. He designed the and implemented the application's architecture, the game entity system, the physics simulation, a major part of the game logic and the Fibix plugin. He also composed one of the songs.

Jakub Kýpeť
AI and web programmer. He created the bots that drive opponents' cars in the game, the AI portion of the Fibix plugin, the deployment script and our webpage.

Jan Navrátil
GUI programmer. He modified an existing GUI library to suit our needs, created actual screens and was responsible for the GUI logic and its rendering, including fonts. His other responsibility was persistent data, mainly user profiles.

1.1.2 The rest of the team

Erik Veselý
Author of Fibix engine and editor (see Chapter 6). He provided support for the Fibix engine and general technical support during the development. He also added several new features into the engine solely for the game's purposes.

Peter Nespešný
Game and level designer and 3D artist. He took the initial game concept and core mechanics and expanded them into a full game design. He designed the Docks level and created some of the 2D and 3D content used in the game.

Jan Kudrnáč
3D and visual effects artist. He created most of the meshes and textures used in the game. He also designed all of the particle effects and created their textures.

Lukáš Tvrdoň
Sound effects. He created all of the sounds used in the game with the exception of the engine sounds.

Braňo Fábry
Music production and recording.

Tomáš Peruňský
Music composition.

1.2 Project timeline

February 2012

Šimon Soták and Erik Veselý agreed on collaboration with the goal of creating a game using Fibix engine.

March 2012

Game ideas were being discussed by the team and a potential game designer. We agreed on creating an action racing game. The first official meeting was held on 21.3.2012. There, we agreed with Jan Beneš for him to supervise the project. We divided responsibilities among the team members and set a deadline for completing the prototype at 6 weeks from then.

April 2012

The prototype was being developed. Car physics were being integrated into it. Existing GUI libraries were being analyzed for usage in the game. AI algorithms for racing games were being studied. Our game designer was turning out to be irresponsible.

May 2012

AI sandbox (a proof-of-concept for the AI methods we have chosen) was being developed side-by-side with the main project. We started looking for 3D artists and considering getting a new game designer. The prototype was finished and we started designing the architecture of the game. We chose a GUI library and started transforming it to fit our needs.

June 2012

We found a new game designer Peter Nespešný and with him a 3D artist Jan Kudrnáč. We were working on the game core and integrating the GUI library into our game. Also, it was the exam period.

July 2012

We started integrating the AI systems into the game. We created a plugin for Fibix to be able to add navigation information for the AI into our levels (waypoints, track borders). Actual GUI screens were being created for the game.

August 2012

User profiles were implemented, GUI screens were created according to game designer's mockups. A basic HUD was created. Our 3D artist created the first actual model – a car. The AI-controlled bots started using the navigation info from the plugin and were for the first time able to finish a lap. We got our first sound working in the game. We had basic particle effects working in the editor. We started the implementation of weapons. We were also polishing the vehicle physics.

September 2012

Our designer created a prototype of the Docks level - the one to appear in the final game. More GUI screens were created. We changed our audio system to use XACT[2]. Bots started

[2] `http://en.wikipedia.org/wiki/Cross-platform_Audio_Creation_Tool`

using weapons and avoiding obstacles. Triggers were implemented. Car hitzones were implemented. The Fibix plugin was extended to a working game editor – entities such as debris and triggers can be created in Fibix, the level can be exported for use in game and the game actually runs inside the editor for quick iteration.

October 2012

Work continues on HUD – we added texture rotation for the speed indicator. Bots' personalities can be customized via game GUI. Bots' collective behavior and overtaking was being polished. Skid marks were implemented. Particles were integrated into the game. Shop and garage screens and logic was being worked on.

November 2012

We have a working website. We have a one-click deployment pipeline. Camera switching is now working. Various game logic is implemented. Most of the 3D content for the Docks level is finished. There's a lot of work importing the content into the editor and then to the game. Particle effects are being designed. Car deformations are working. Various sounds are implemented.

December 2012

We successfully presented MotorDead at the Game Developers Session 2012[3]. Then we started working on the project documentation. We were cleaning up code, optimizing, polishing, testing and debugging.

January 2013

GUI polishing, debugging and writing documentation. We successfully presented MotorDead at the MFF open day as an ongoing project of the Computer Graphics Group. We were also asked to represent the Charles University at the Gaudeamus expo[4] with the project. We successfully did so on January 29. and 30. in the MFF booth.

[3] `http://gds2012.ceske-hry.cz/`
[4] `http://www.gaudeamus.cz/`

Chapter 2

Project overview

2.1 Technology

The target platform of MotorDead is Windows Vista or higher with .NET Framework 4.0. 64-bit build is not supported due to a lack of support in our graphics engine. Windows XP is not supported as the engine requires DirectX 10 support.

2.1.1 Languages

Usually, the primary choice of programming language for developing non-casual 3D games has been C++. However, this is mainly due the low-level nature of graphics and physics engines' needs. The downside to using C++ is the lack of high-level features that are so popular nowadays with modern languages (e.g. garbage collection). We have decided to combine these approaches by keeping the performance-critical code in C++ and using C++/CLI wrappers for use in C#.

2.1.2 Libraries

We are using the Fibix engine[1] for 2D and 3D rendering. The rendering core which interfaces with DirectX is written in C++ and wrapped in C++/CLI. For physics simulation, we are using the Bullet engine[2] wrapped in C++/CLI by the BulletSharp wrapper[3].

For input, sounds, math and miscellaneous utilities, we are using Microsoft XNA. Note that this set of tools is usually used as a game and rendering framework, because it includes content and rendering pipelines, but we only use its base (*Microsoft.Xna.Framework*), XACT sound (*Microsoft.Xna.Framework.Audio*) and input (*Microsoft.Xna.Input*) modules, out of a total of twelve available modules.

[1] `http://fibix.eu/`
[2] `http://bulletphysics.org/`
[3] `http://code.google.com/p/bulletsharp/`

For GUI logic, we chose an XNA-based GUI library "SimpleGUI"[4]. We had to heavily modify this library in order to extract its rendering logic into an interface, since we are not using XNA for rendering. We then implemented this interface by calls to the Fibix renderer.

2.1.3 Licensing

Fibix is a proprietary software. For the development of MotorDead , the four programmers (see 1.1.1) signed a licensing agreement with the proprietor of Fibix Engine. Thanks to the agreement, we had access to the source code of Fibix but we cannot disclose it along with this document. This was stated in the project specification and approved by the Software Project committee.

Bullet and BulletSharp are open source projects released under a custom license[5] and the MIT license[6] respectively. Neither of these licenses require the project that uses Bullet and BulletSharp to be open source.

SimpleGUI is an open source project available under the LGPL license[7]. This license requires that modifications of the original library must be released under the same license. Therefore, the modified source code is available at the project's website `www.motordead.com`.

2.2 Requirements review

In this Section, we will list the proposed requirements for the success of this project and comment briefly on each one.

2.2.1 Minimum required features

Dynamic gameplay with weapons, collisions and explosions

We are quite happy with the resulting gameplay. The feedback we have received on MotorDead's "fun-factor" from people who played the game for the first time was mostly positive. Weapons, collisions and explosions are all present in the game.

3D graphics with modern effects (HDR lighting, ambient occlusion)

By incorporating the Fibix engine and its content pipeline, we have achieved this goal (see Chapter 6). Moreover, the graphical detail and level of fidelity are highly configurable, so the game looks good on powerful PC's but can be configured to run well on older machines.

Particle effects

As part of our work, we have implemented a particle effects system into the Fibix engine and successfully used it in MotorDead. See Chapter 7.

[4] `http://simplegui.codeplex.com/`

[5] For the license of Bullet, see the source code at `http://code.google.com/p/bullet/`

[6] `http://opensource.org/licenses/mit-license.php`

[7] `http://simplegui.codeplex.com/license`

Vehicle physics model allowing for configuration of the resulting behavior

The way we modelled vehicle physics results in a balanced compromise between a realistic and a fun-to-play experience. The cars are easily controllable with a keyboard, but modelling various physical phenomena (such as different friction forces on different types of surfaces) is still straightforward and does not require "hacks". The configuration of the resulting behavior is demonstrated by various vehicle upgrade levels available in the game. See Chapter 11.

In-game vehicle editor with configurable equipment

Users can view their cars and change their equipment in the garage and buy new goods in the shop.

GUI allowing for control of the vehicle editor and game options

We use our GUI system for all of the menus. All of the game input and output that is not directly in the race is handled via the GUI. See Chapter 10.

Artificial intelligence controlling the opponents' cars

The bots in MotorDead are fierce and unforgiving, contributing to an immersive experience. They are also quite robust in navigation and obstacle avoidance. See Chapter 13.

Sound effects

The game supports various kinds of sounds. See Chapter 15.

Quick race game mode

The only game mode we support is quick race: select a map, number of opponents, number of laps and play.

2.2.2 Optimum features

Aesthetic and consistent game visuals

We are very happy with the world our artists have designed and created. However, the ideas they conceived required a lot of work on the technical side as well, so they kept the programmers occupied.

AI capable of different characteristics and/or difficulty levels

Every opponent driver in the game is represented by a persistent character with a name. Their different personalities are achieved by a set of characteristics configurable in-game by a designer, such as aggressiveness or cautiousness. See Section 13.5.

High-fidelity sound effects and music

Our audio effects designer provided us with a wide variety of sound effects. For a detailed list, see Table B.1. Our team members composed several songs, out of which three were chosen by our music producer Braňo Fábry and recorded by him as well.

Career game mode

Due to lack of content and due to time constraints, this feature was skipped.

Split-screen multiplayer

We decided not to implement split-screen and rather focused on other features.

2.3 Testing

After finishing all of the features, polishing and debugging, we have tested the game thoroughly by having two people go through our test cases (see Appendix 17). These generated additional bug reports which we have resolved.

In addition to the test cases, the game was being tested continuously by programmers and designers and also by people who have never seen MotorDead before, during the game's presentation at the MFF open day and at Gaudeamus (see Section 1.2).

2.4 Bugs and known issues

The version distributed to non-developers includes an error-reporting mechanism: In the case of an exception in the managed portion of the code, the application collects all available data about the machine and sends it together with the application's trace log to the team leader's email address, if the user agrees. This is common practice in the industry and makes crash reporting painless.

Although we worked hard to polish the game, there are a few remaining issues. However, most of these are limitations of the libraries we use.

Fullscreen window mode

Most games support a rendering mode called Fullscreen, in which the GPU is dedicated to the game entirely. Unfortunately, we weren't able to overcome a problem with DirectX11 rendering inside a .NET's Windows Form and switching to fullscreen. An alternative fullscreen mode is present in the game - using a borderless fullscreen window.

Imperfect collisions

Due to the discrete nature of physics simulation used in games, resolving collisions of objects with high velocities does not always yield correct results. Since the vehicles in MotorDead are composed of several primitive shapes (see Section 11.13), occasionally a player can get stuck in static geometry (see Figure 2.1). Fortunately this happened very rarely during our playtesting and never between two vehicles (which is the most frequent type of collision).

Lack of keyboard navigation in the GUI

The SimpleGUI library (2.1.2) had poor support for non-mouse navigation and control, and we did not have enough resources to implement it anew. Therefore, the GUI can be only controlled by the mouse whereas the game can only be controlled by the keyboard, so the player has to switch between these two input devices. See Chapter 9 for more details about input.

Figure 2.1: A vehicle stuck in a pillar.

Open game data

In order to allow for easy addition of game content, we keep our game data files in text form and editable (see Section 5.5). It is therefore possible to break the integrity of the game data by making mistakes or on purpose. We therefore recommend to adhere tightly with the directions provided in Appendix E.

2.5 Project statistics

A rough estimate of time invested into the project is 370 to 420 man-days, including programming, content creation, management and documentation.

We were using Assembla[8] for project management. We were using Assembla's "Tickets" tool to keep track of tasks, assign responsibilities and report bugs. We have created more than 400 tickets, out of which 50 are still open, 320 are finished and 30 are marked as invalid.

2.5.1 Source code

We were using Apache Subversion[9] for version control. We made more than 1600 commits into the repository. We have written more than 70 thousand lines of code including whitespace lines and comments (43 thousand excluding them). The graphs 2.2, 2.3 and 2.4 visualize various aspects of code versus time. These graphs were generated by StatSVN[10].

[8] `http://www.assembla.com/`
[9] `http://subversion.apache.org/`
[10] `http://www.statsvn.org/`

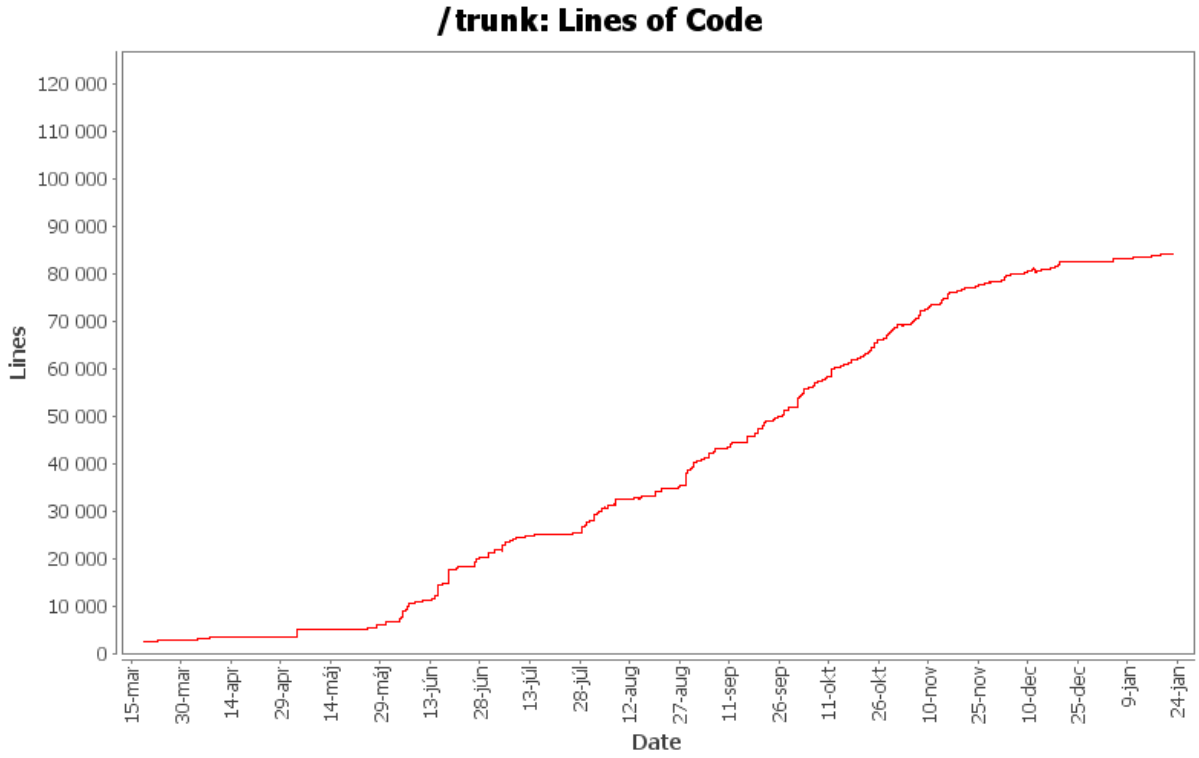

Figure 2.2: Lines of code (vertical axis) vs. time (horizontal axis).

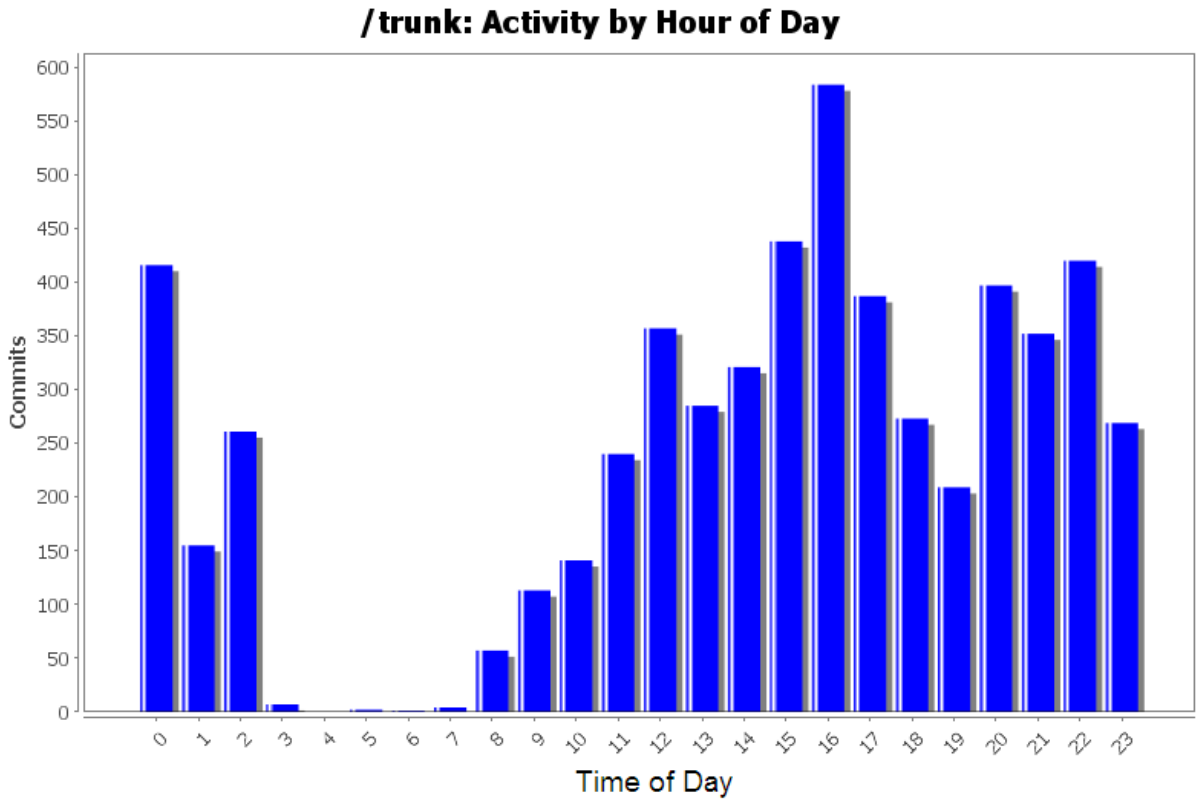

Figure 2.3: Commit activity by hour of day. Some of us were most productive after midnight.

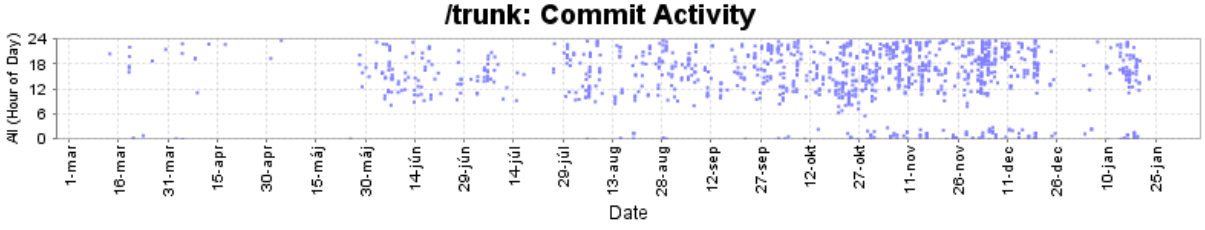

Figure 2.4: Commit activity vs. time (horizontal axis), by time of day (vertical axis).

Chapter 3

Build process

3.1 Development environment

We used Microsoft Visual Studio 2010 (VS2010) for the development of MotorDead. You need it to build the application.

Additional packages that need to be installed:

- Microsoft DirectX End-User Runtimes (February 2010 or newer)[1]
- Microsoft XNA Game Studio 4.0[2]

After you have these and you have the project development files (either from our SVN server or from a physical medium), open config_default.ini in the root of the project and set the dataFolder variable to contain the absolute path to the "data" folder that is located in the project root directory and the `systemDataFolder` to the "systemData" folder accordingly. Then open `src\FightRaceStandalone.sln` in VS2010. Select either Debug or Release configuration and you should be able to build and run the game.

3.2 Deployment

To deploy MotorDead, we use InnoSetup[3] for creating an installer file. We have created a batch script to do the whole process of creating a setup executable in one click.

1. Open `build-config.bat` located in the root folder in a text editor and configure the paths to correspond to the install locations of MS Visual Studio 2010 and XNA Game Studio on your computer.
2. Run `build-FightRace-Standalone.bat` located in the root folder and wait until the setup executable named motordead-setup.exe is created in the root folder.

[1] `http://www.microsoft.com/en-us/download/details.aspx?id=35`
[2] `http://www.microsoft.com/download/en/details.aspx?id=23714`
[3] `http://www.jrsoftware.org/isinfo.php`

Chapter 4

Architecture

4.1 Overview

The overall view on the architecture of MotorDead is shown in Figure 4.1. It is a centralized / hierarchic hybrid where the *FightRaceGame* module is the centre as well as the root of the hierarchy. *FightRaceGame* provides access to various game services and most of the classes can communicate directly with this central module.

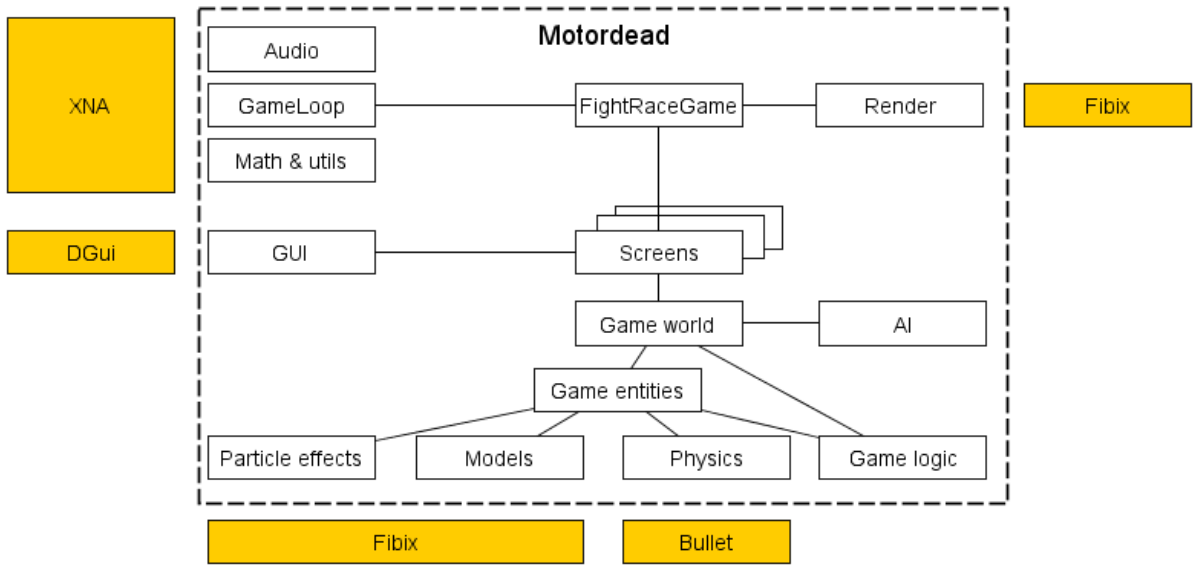

Figure 4.1: Overall view on the architecture of MotorDead. Modules of MotorDead are displayed in white. The dashed line is the boundary between the MotorDead source code and the third-party libraries used in the project. The third-party libraries are displayed in yellow.

During a race, the game simulates the virtual world (implemented by the *GameWorld* class, see Chapter 5) and the objects in that world. These object include vehicles, weapons, static

objects, etc. An object is represented by a game entity (see Chapter 5). Each game entity has a physical form, a graphical appearance (most often a model and/or particle effects) and game logic. The game world also contains a great deal of game logic that deals with starting the race, resetting the vehicles, race results, etc.

The AI module is used to control the vehicle of the opponets in a race. The AI is used at the level of the game world - the entities don't contain any AI, rather they are controlled by the *AIManager* class that is created with the *GameWorld* class. For more details about the AI in MotorDead, see Chapter 13).

Screens and the screenstack is the system that is used instead of a state machine to represent the game states. Each screen represents a certain game state and implementes the logic of that state. Screens are added to (or removed) from the screenstack in order to change the game state. Screens use GUI module for the UI components (e.g. buttons, labels, textboxes, etc.). The details about the screens and the screen stack can be found in Section 4.4. The game state for race session (i.e. the state inducing that the race session is in progress) is also implemented by a screen that holds the game world.

The FightRaceGame module represents the top level of the architecture. It is the main module of the project, that holds "everything together". It is implemented by the *FightRaceGame* class (see Section 4.3). The *FightRaceGame* class is a singleton that implements the game loop and contains a screenstack to manage the game states. Note that the time counting mechanism for the game loop is part of the XNA framework (see details in Section 4.2).

The Render module is used to access rendering from Fibix. It also manages the rendering scene. The render module is used by the *FightRaceGame* class (i.e. the main module). For more details, see Chapter 6.

The Audio and Math & utils modules are displayed without any connection to other modules. It is because they are not used in any specific place. They are used widely throughout the modules and classes of MotorDead. Both the Audio and Math & utils modules use parts from the XNA framework. More details about audio in MotorDead can be found in Chapter 15. The Math & utils module is not a compact module with specified functionality. It is displayed to illustrate the fact that MotorDead uses various parts of the XNA framework (e.g. vectors and quaternions, math utilities, etc.).

4.2 Game loop

One of the many goals of any good video game is providing a user with a pleasant experience. To do so the game must appear fluid (i.e. it must render at least 25 frames per second) and respond to the user's input without any significant lag.

A common approach used in the interactive video games is the concept called "Game Loop". The core of the game is formed by a never ending loop which, in each iteration, determines the

input from the user, updates the game and renders new frame according to the current state of the game. The loop is sketched in pseudocode in Algorithm 1.

Algorithm 1: Pseudocode of the game loop

```
while not user exits the game do
    Determine input from the user
    Update the game according to the input
    Display updated state of the application (i.e. render new frame)
end
```

4.3 FightRaceGame class

Implementation of the game loop is in the *FightRaceGame* class. It is divided into two separate phases - update and draw. During the update phase the game updates its state (for example it updates the movement of the vehicles, the information displayed on the HUD, etc.). During the draw phase new frame is rendered and showed to the user.

The *FightRaceGame* class contains the *Update(float deltaT)* and *Draw(float deltaT)* methods which correspond to these two phases. The parameter in both of the methods have the same meaning - time elapsed since the last call. The game also needs a target window which is used to display the rendered frames.

To support the creation of the game's content a custom plugin to Fibix Editor has been created. Fibix edtitor is an editor that comes with the rendering engine used in MotorDead. For more information about the plugin, see Chapter 16. The decision to create custom plugin to Fibix Editor gives the different contexts in which the game may exist - standalone application and editor plugin.

When the game runs as a standalone application, the XNA framework is used for the time counting. To use the XNA's implementation of the game loop mechanism, one must derive a class from the XNA's *Game* class and override its *Update(GameTime gametime)* and *Draw(GameTime gametime)* methods. Both are called as often as possible, depending on the power of the computer and current performance of the game. The parameter *gameTime* can be used to determine the time elapsed since the last call. Together with the time counting mechanism the *Game* class also provides a window which can be used to display the rendered frames.

Both the time counting mechanism and the target window are used through the *XNAWindowGame* class. It is derived from *Game* class and it holds instance of the *FightRaceGame* class (the private *_game* field). The *XNAWindowGame* class overrides the *Update* and *Draw* methods (of the *Game* class). In the overridden methods it calls the corresponding method of the *FightRaceGame* instance. The target window is used to initialize the Fibix engine, so that the rendered frames are displayed in this window (for more details about the engine, see Chapter 6). The relationship between the *XNAWindowGame* and the *FightRaceGame* class is depicted in Figure 4.2.

In the plugin the target window is given provided by the editor. Therefore it is not necessary to create a new window for the game. Also the update and draw calls come from the editor (for more information about the plugin, see Chapter 16).

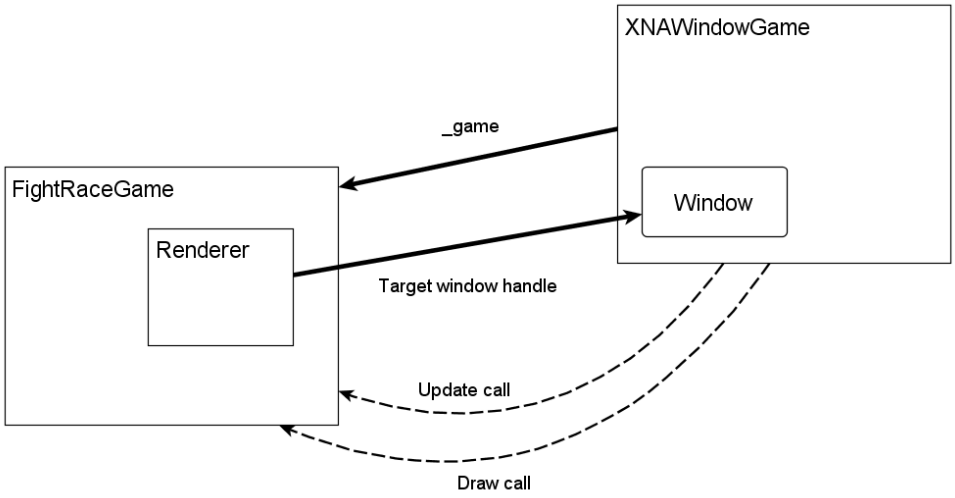

Figure 4.2: The relationship between the *FightRaceGame* class and the *XNAWindowGame* class. The *XNAWindowGame* class uses the XNA's game loop mechanism and call the *Update* and *Draw* methods of the *FightRaceGame* class. The *XNAWindowGame* provides window that is used to display the rendered frames.

4.4 Screenstack

A common approach to managing the game state is to use a finite automaton. A game state determines what the game displays, what the user can do and which transitions to the other states are possible. Typical game states include: main menu, options, playing the game, etc.

In MotorDead, a different approach is used. Instead of a finite automaton, a stack of screens is used. A screen is a horizontal layer that has its own graphical representation and logic. Every screen in the game is derived from the abstract class *Screen*. It has three boolean properties: *Active*, *Visible* and *PassInput* and three methods: *HandleInput(Input input)*, *Update(GameTime gameTime)* and *Draw(GameTime gameTime)*. These properties and methods are important during the update and drawing of the screen stack. The value of *Active* controls whether the screen is updated (it is update only if the value is *true*). The *PassInput* property controls whether the input is passed to the screens below the screen and *Visible* controls whether the screen is drawn.

The screen stack (with the above mentioned functionality) is implemented by the *ScreenStack* class. The update of the stack is done in the *Update(GameTime gameTime, Input input)* method. This method iterates through the screens from the top of the stack to the bottom. For each screen that has the value of *Active* set to *true* it calls the screen's *Update* method. If the input hasn't been blocked by any screen above, the *HandleInput* method of the screen is also called. The

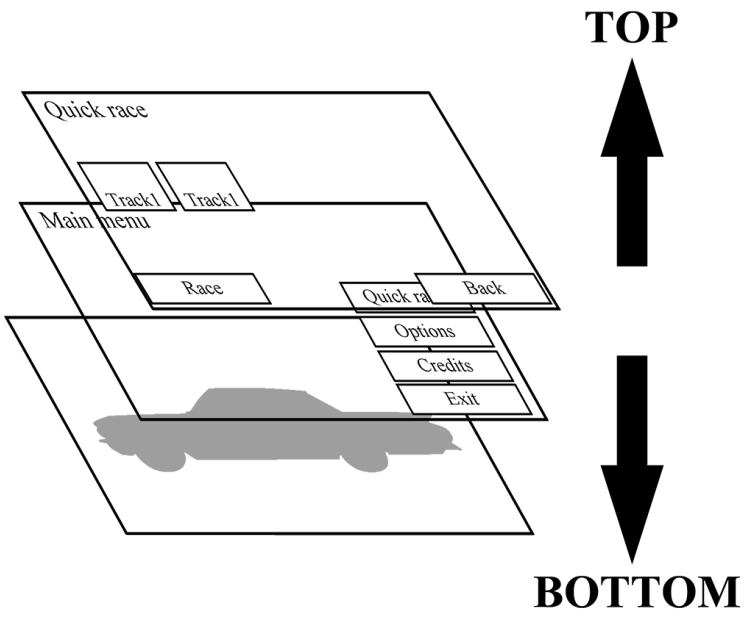

Figure 4.3: Horizontal character of the screens.

input is blocked when the screen's *PassInput* value is *false*. Pseudocode for the update of the screen stack is summarized in Algorithm 2.

Algorithm 2: Pseudocode for ScreenStack.Update method

```
Data: gameTime – duration of the update step, input – information about the user's input
boolean passingInput = true;
for each screen from the top to the bottom of the stack do
    if screen.Active then
        if passingInput then
            screen.HandleInput(input);
        end
        screen.Update(gameTime);
    end
    passingInput = passingInput && screen.PassInput;
end
```

This update procedure allows the screen to change its value of *PassInput* according to the current input from the user. For example, a screen can pass input all the time and only when certain key is pressed, it blocks the input and handles it on its own. However, currently there isn't any screen using this possibility.

Drawing of the screen stack is done by the *Draw(GameTime gameTime)* method of the *ScreenStack* class. It iterates through the screens on the stack from the bottom to the top. For each screen, that has value of the *Visible* property set to *true*, it calls the *Draw* method. Going

through the screen in this order automatically resolves the ordering in the Z-axis when multiple screens on the stack are visible. It means that top most screen is drawn last and therefore it is not obscured by the screens that are below it. The *Draw* method is summarized in Algorithm 3.

Algorithm 3: Pseudocode for ScreenStack.Draw method

```
Data: gameTime
for each screen from the bottom to the top of the stack do
    if screen.Visible then
        screen.Draw(gameTime);
    end
end
```

The current game state is never stored explicitly. It is implicitly defined by the screens that are on the stack, their order and values of the *Active*, *PassInput* and *Visible* properties. The game state can be changed by changing the values of the properties of the screens that are already on the stack. For example the topmost screen can pass input only in certain situations or it may become temporarily invisible.

The most common way of changing the state is to either push a new screen on the stack or to remove a screen from the stack (methods *Push(Screen screen) and Remove(Screen screen)*. Both operations can be done using the *Show* and *Close* methods of the *Screen* class. The *Show()* method pushes the screen on the stack and the *Close()* method removes it from the stack. Both methods use the instance of *ScreenStack* that is given in the constructor of the screen. This makes it impossible for any screen to be added to two different stacks during its life. However, there is only one instance of *ScreenStack* present in the game (the screen stack is one of the game's services - for more details about the services, see Section 4.5).

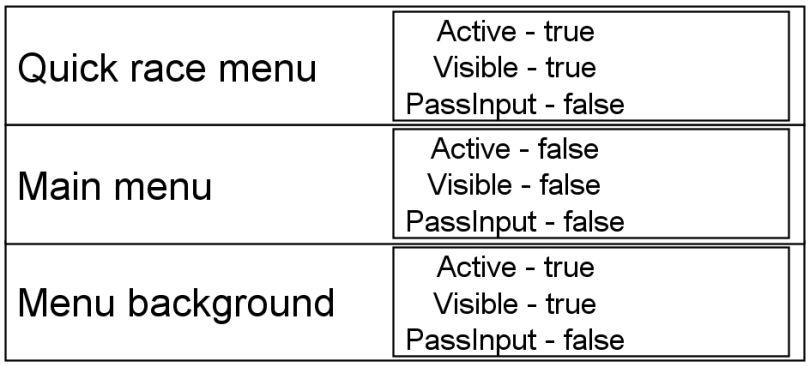

Figure 4.4: Screen stack when then user is choosing the car and the track for the race.

Figures 4.5 and 4.4 depict two different example states of the screen stack. Figure 4.4 contains the screen stack configuration when the user is choosing the vehicle and the track for the race. Please note that there are two active screens. Quick race menu screen implements the logic

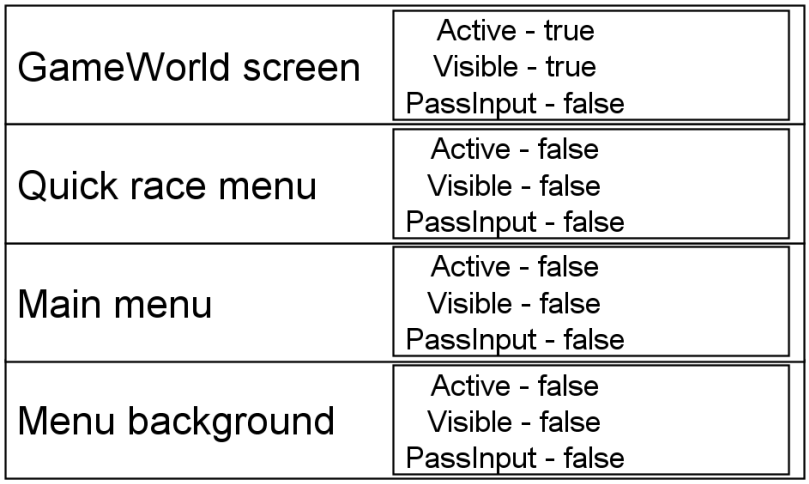

Figure 4.5: Screen stack during a race session.

of choosing the vehicle and the track. Menu background is used to display the currently selected vehicle. This is done by rendering the 3D scene in which the vehicle is placed.

Figure 4.5 illustrates the situation when the user is playing the race. After the vehicle and track were chosen, the GameWorld screen is pushed to the top of the stack. It is the screen responsible for displaying the 3D scene during the race.

Please note that there is only a single 3D scene in the whole game (for details about managing the 3D scene see Chapter 6). Consequently there can't be two visible screens on the stack that want to render the scene at the same time. However this doesn't limit anything since only the scene rendered by the topmost of these screen would be visible and in MotorDead there is no need to have such a screen stack state. Overall the screen stack approach is very flexible and allows easy expansion of the game.

4.5 FightRaceGame - services

The *FightRaceGame* class not only implements the logic in the game loop, it is also a way for accessing services. A service is an object that offers specific functionality and this functionality can be used by other objects.

An example of a service is the *ModelFactory* class. It is able, among other things, to load 3D models from secondary memory and instantiate them. The *FightRaceGame* class holds one instance of *ModelFactory* and in the application it is accessible via the *ModelFactory* property of the *FightRaceGame* class.

All services from the *FightRaceGame* class are created at the start of the game (*Initialize* method of the *FightRaceGame* class) and live until the game is exited. Each of the services is accessible through a property of the *FightRaceGame* class. Therefore, an instance of *Fight-*

RaceGame is passed to every object that might need to use the services. These services are listed in Table 4.1, together with a short description and the name of the property that is used to access the service.

Property name	Class name	Description
ActiveUserProfile	*UserProfile*	Holds information about currently logged user. For more details, see Chapter 14.
Config	*Config*	Configuration from config.ini file.
CoordinateManager	*CoordinateManager*	Manages GUI coordinate conversions and normalization. See Section 10.6.
EntityFactory	*IEntityFactory*	Creates game entities such as vehicles, weapons and vehicle upgrades. For more details, see Section 5.6.
EntityMaterialManager	*EntityMaterialManager*	Manages the materials of the physical entities. It is able to return the properties of the material with a given identifier - *EntityMaterial GetMaterial(uint id)* method. For more details, see Section 5.7.
FontManager	*FontManager*	Provides fonts by name, loads if necessary, keeps the fonts cached. For more details about fonts, see Section 6.7.
GuiManager	*GuiManager*	Boss manager for GUI-related service managers. Contains *CoordinateManager*, *GuiTextureManager*, *SoundPlayer* and *FontManager*.
Input	*GameInput*	Provides information about input from the user. It is used to get the information about key presses, mouse position and mouse button presses. For more details, see Chapter 9.
InventoryManager	*InventoryManager*	A helper class used to manage item buying and selling in the garage and in the shop.
ModelFactory	*IModelFactory*	Loads 3D models from files and instances them. For more details, see Section 6.6.
OpponentsManager	*OpponentsManager*	Loads ,manages and provides access to the AI opponents that are used in the race. For more details, see Section 13.6.

ParticleEffectsManager	*ParticleEffectsManager*	Loads particles effects from file, instances them and destroys them after the usage. For more details, see Section 7.5.
Renderer	*Renderer*	Manages 3D engine (at a very high level) and 3D scene. For more details, see Section 6.2.
ScreenStack	*ScreenStack*	Main screen stack that is updated and drawn. It is used for all screens in the game. For more details, see Section 4.4.
SoundManager	*SoundManager*	Wraps XACT and allows both the 2D and 3D sounds to be played. For more details, see Chapter 15.
TimeProfiler	*SimpleTimeProfiler*	A simple profiler for non-hierarchical time measurements. We used it to see how much frame time was spent in which modules (AI, physics, render, etc.).
VehicleBuilder	*VehicleBuilder*	Builds vehicles according to the given description. The description contains the vehicle identifier, the health of the vehicle and the installed upgrades. For more details, see Section 11.18.

Table 4.1: List of services from the *FightRaceGame* class.

4.6 Per-race services

The services described in the Section 4.5 live throughout the whole life the application - from the start of the game, until the user exits the game. There are, however, additional services that are active only when the race is on. These services can be found in the *GameWorld* class. The *GameWorld* class is responsible for managing and updating the physical entities (for more information about the physical entities, see Chapter 5. An instance of *GameWorld* is created for each race. After the race has ended, it is no longer used and it is destroyed. Per-race services are listed in Table 4.2.

Property name	Class name	Description
AiManager	*AIManager*	Manages the AI drivers (for more details, see Section 13.3).
PhysicsManager	*PhysicsManager*	Wraps Bullet's *DynamicsWorld* class and updates the physics of game entities (more details can be found in Section 8.5).
SkidMarksManager	*SkidMarksManager*	Manages skid marks produced by the vehicles during the race (for more details, see Section 12.10).

Table 4.2: List of services that live only during the race.

4.7 Unmanaged resources

The MotorDead uses the .Net platform (the source code is written in C#). Therefore the memory is managed by the garbage collector. This, however, doesn't mean that all of the allocated memory is released automatically.

Both Fibix (3D engine) and Bullet (physics engine) are the native libraries that are used through wrappers. Naturally, they allocate unmanaged memory, which has to be released explicitly. Therefore, every class that creates new object from the wrappers must dispose the object when it is no longer to be used. An example of releasing the unmanaged resources can be seen in Section 5.2.2.

Chapter 5

Game entities

The concept of MotorDead, which is an action racing game, requires simulating real world objects (e.g. vehicles, weapons, etc.) in a fictional context. The simulated world is represented by two separate scenes:

1. Rendering scene - contains graphical representations of the objects in the game (3D models and particle effects)

2. Physics scene - contains physical forms of the objects (rigid bodies)

This approach is enforced by the integration of the two independent libraries: Bullet for physics simulation and Fibix for rendering. Moreover, it allows the independence of the physics simulation from the appearance. This is needed for performance reasons. The physical form of an object is most often a rigid body (see Section 8.1). Rigid bodies are used in the collision detection process (see Section 8.1) which is very CPU consuming. Therefore the shapes of the rigid bodies must be kept as simple as possible - much simpler than the 3D models.

5.1 Frame update and physics update

As mentioned in Chapter 4 the game is regularly updated in the game loop. Using this approach "as is" has a big disadvantage: the update time of the underlying physics simulation changes according to the current performance of the game. If the performance is good, much time is wasted in many small updates. On the other hand, when the performance drops, the update time becomes longer. This is much worse, because it can lead to an instability in physics simulation.

To keep the physics simulation stable it is necessary to update the physics at a constant rate. It means that physics simulation is updated every time with the same update time. This holds true only for the physics simulation. All game components that don't need to be updated with constant update time (AI, positions of the 3D models, rendering of the new frame) are updated only once for each rendered frame.

For example let the update time of the physics be 1/60s and let the game run at 30 FPS. Then there are two update steps of the physics simulation needed for each rendered frame. On the other

hand, it is enough to update the positions of the 3D models only once, before the rendering. Note that updating the physics in the constant time steps means that the position and rotations (i.e. world matrices or world transforms) of the simulated objects are known only for the regularly distributed time points. It is necessary to interpolate the world transform of the objects to obtain the world transforms at any given time.

Use of the constant update time for physics simulation is also dictated by the Bullet physics library that is used in MotorDead. Bullet physics has a good support of this mechanism - it automatically does updates at constant rate and also interpolates the world transforms of the objects. For more details about the integration of the Bullet physics library, see Chapter 8.

5.2 Physical entities

Each of the simulated objects in the game is represented by one physical entity. Basic implementation of the physical entity is done by the abstract *PhysicsEntity* class. All entities must be derived from this class. The *PhysicsEntity* class contains an implementation (of the methods and properties) that is sufficient for most of the entities.

Every physical entity has its physical form and graphical appearance. The most common combination is a rigid body (the physical form) and a 3D model (the graphical appearance). The rigid body is used in the physics simulation and it defines the world transform of the the entity. The world transform of the entitie's 3D model is set accordingly. The *PhysicsEntity* class already has the *_rigidBody* and *_model* fields that hold the rigid body and the 3D model of the entity. Value of either of them may be *null*, in the case when the entity has a different physical form or graphical appearance.

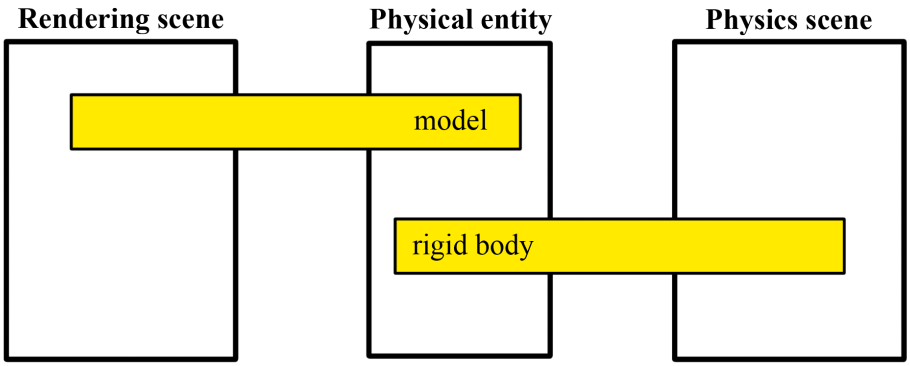

Figure 5.1: Physical form (rigid body) and graphical appearance (model) of a physical entity. Each of the two belongs to a different scene.

PhysicsEntity provides an interface for updates of the physical entity. It contains the *PhysicsUpdate(float deltaT)* method, which is called during the update of the physics simulation. In this method, custom forces that act on the entity's rigid body can be added. The basic implementation does nothing since the impulses from collisions among the rigid bodies are applied automatically by the Bullet physics library.

The *PhysicsEntity* class also contains the *GraphicsUpdate()* method. This method is called in each frame and it performs the update operations that need to be done only once per frame. Its main goal is to synchronize the world transform of the physical entity's 3D model with the interpolated world transform of the entity's rigid body.

5.2.1 Physical entity hierarchy

The construction of hierarchical structures from models is one of the basic building blocks in any computer game. Since there is no support of hierarchical structures in Fibix, the hierarchy is implemented in MotorDead at the level of physical entities, in the *PhysicsEntity* class.

An instance of *PhysicsEntity* can be assigned as a child to an another *PhysicsEntity* (see methods *AddChild* and *RemoveChild* of *PhysicsEntity*). The world transform of a child's model is then given not only by the transform of the child (it gives the local offset transform) but also by the parent's world transform.

The hierarchy mechanism also requires updates of the model's world transform. As mentioned in Section 5.2, it is done in the *GraphicsUpdate* method. Not only does it update the world transform of the model of the physical entity, it also recursively calls the *GraphicsUpdate* method on all children of the entity.

The physics update is not recursive. If any of the children needs the physics update it must be implemented as explicit call in the parent (e.g. update of weapons in the *Vehicle* class). This approach is not a general solution to the entities hierarchy. However, the task of implementing general hierarchy for physics simulation is overwhelming for the purpose of MotorDead. The only physical entity currently having children is the *Vehicle* class. The resulting hierarchy system proved to be satisfactory for the game.

Note that the children of *PhysicsEntity* are instances of the *Spatial* class. *Spatial* is the base class of *PhysicsEntity* and it deals with the transformations (both local and global) and bounding volumes of the entity. This separation of functionality is a legacy from the original source code - classes *Spatial* and *PhysicsEntity* are taken from Tesla engine[1].

5.2.2 Destroy

The *PhysicsEntity* class contains instances of the *Model* and *RigidBody* classes. Both of these classes represents objects from the native libraries used in MotorDead - Fibix (3D engine) and Bullet (physics engine). The memory allocated by these objects has to be released when they are

[1] Homepage of the Tesla engine is located at `http://www.tesla-engine.net/`, the repository can found at `http://code.google.com/p/tesla-engine/`.

no longer to be used. Moreover, derived entities might use even more unmanaged resources than the *PhysicsEntity* class.

For the purpose of releasing the unmanaged resources the *Destroy* method of the *PhysicsEntity* class is used. The default implementation disposes the instances of the *Model* and *RigidBody* classes. If a derived class uses any other unmanaged resource, it must be released in the *Destroy* method. The *Destroy* method is called only when the entity is no longer to be used.

5.3 Simulation world

The *SimulationWorld* class represents the world (the environment) that contains the simulated objects. Physical entities can be added to or removed from this world using the methods *AddEntity* and *RemoveEntity* respectively. The *SimulationWorld* class holds the list of physical entities and updates them in the *Update(GameTime gameTime)* method.

More specifically, *SimulationWorld* holds instance of the *PhysicsManager* class that manages the physics simulation of the world. When the physical entity is added to *SimulationWorld* it is also added to the *PhysicsManager* instance in the *SimulationWorld* and similarly for removal of the entity. The *Update* method of the *SimulationWorld* class updates the *PhysicsManager* by calling its *Update* method. This call updates the physics simulation. Afterwards, the *SimulationWorld* calls the *GraphicsUpdate* method on each of the entities.

Algorithm 4: Pseudocode for the update of the *SimulationWorld*.

```
Data: Update time - deltaT
PhysicsManager.Update(deltaT);
forall the entity in simulation world's entities do
    entity.GraphicsUpdate();
end
```

5.4 Game world

The *SimulationWorld* class has only basic functionality for managing and updating of the physical entities. The *GameWorld* class is derived from the *SimulationWorld* class. It adds specific functionality that is needed for running the race sessions. It has the *PrepareWorldForSession(RaceSessionDefinition session)* method that prepares the world for the race. It also holds an instance of *AIManager* that manages AI opponents during the race.

The *GameWorld* instance is held by the *GameWorldScreen* class which is added to the screen stack when the race session starts. The *GameWorldScreen* updates the *GameWorld* in the screen's *Update method*. In the screen's *Draw* method it sets the position of the view matrix, so that the 3D scene is rendered correctly. The *GameWorldScreen* class also contains a controller (the *_userVehicleController* field) that translates the user input (i.e. pressed keyd) to the commands for the user's vehicle in the *GameWorld*. The controller is updated in the screen's *Update* method.

A new instance of *GameWorld* is created for each race session. It is created automatically in the constructor of the *GameWorldScreen* class. The created game world screen is then added to the top of the screen stack. It is removed when the race session ends (see event *GameWorldScreen.RaceEnded*).

5.5 Definition of game entities

MotorDead uses an unified, data-driven way of defining the game entities. It can be used to define both physical entities and other objects in the game (e.g. vehicle upgrades). Each game entity is defined by one Entity Definition File - EDF. An EDF can be viewed as a definition of an entity specimen that is instanced (i.e. cloned) in the game. For example, an EDF file defines the machine gun weapon. In the game, there can exist multiple machine guns. However, all of them have the properties given by the EDF file - they are instances of the same entity.

EDFs are located in the game's data folder, specifically in the "data\entities" folder. EDF files are text files with a format that is depicted in the Table 5.1.

Line content	Description
Identifier	Unique string that identifies the entity. No other EDF file may contain equal identifier.
Implementing class	Name of the class that implements the entity. Given class must implement the *IEntity* interface.
Displayed name	Name of the entity that can be displayed to the user if needed.
Description	Long description of the entity.
$=<price>	Price of the entity (this line is non compulsory)
<Option name> = <Value>	Arbitrary number of lines in this format. The option names and value types are specific to the implementation class.

Table 5.1: Format of the entity definition files.

A line in Table 5.1 corresponds with a line in the EDF file. First 5 lines of the EDF file form the header. The line with the price of the entity is non compulsory. If the line is missing the price of the entity is assumed to be 0. The rest of the file (from the 6th line to the end of the file) is formed from options of the entity. The options depend on the class that implements the entity (the one that is listed on the second line of the file).

This approach is used because it allows data-driven adding of new entities. Although the adding of the entities is currently limited, new vehicles and tracks can be easily added (more details about adding new vehicles and tracks can be found in Appendix E).

5.5.1 Implementation classes of the entities

As mentioned in Section 5.5, new entities can be added purely as data without any changes in the source code. One of the limitations is that new entities can only use one of the implementation classes that are currently present in the source code. Note that very small modification would be needed to support implementation classes from externally loaded dlls. There is even support for such an approach in the *EntityImplementationLoader* class. However, since the usage of the new entities is limited due to the game design decisions (limited amount of the weapon types and vehicle upgrades) and the absence of a scripting language support, there is no need to support the dynamic loading of implementation classes.

5.6 Creating game entities

Game entities are created using the *EntityFactory* class (accessible as a service from *FightRaceGame*). It contains the *GetEntity(string id)* method that creates a new instance of the entity with given identifier.

Entity factory holds list of the known entities. For all of the entities, data needed for creating new instances are known - the identifier, implementation class and options values from EDF file. The implementation class must implement the *IEntity* interface. There is an instance of the implementation class associated with each entity. It is called "Creator instance" and it is used to create a new instance by calling the *Create(FightRaceGame game, EntityArgs args)* method.

First parameter of *Create* method is the context in which the new instance is created. It is necessary to pass the instance of *FightRaceGame* to the new instance so it can use the game services. Second parameter contain the values of the options from the EDF file.

5.7 Entity materials

Each physical entity has a material that defines its properties, both physical and graphical. The materials are managed by the *EntityMaterialManager* class which is available as a service in the *FightRaceGame* class. Each material has an unique identifier of *uint* type. It is called “Semantic ID”. Using the identifier, the material can be retrieved from *EntityMaterialManager* via the *GetMaterial* method.

The material of the entity can be determined by the *GetMaterialId* method of the *PhysicsEntity* class. The default implementation (in the *PhysicsEntity* class) return the default material[2]. Classes derived from the *PhysicsEntity* class can use the parameters of the *GetMaterialId* method and return a different material for different parts of the entity. Such a possibility is used extensively in the *Track* class. Instances of the *Track* class represent the race tracks. The race track may contain multiple surfaces (asphalt, gravel, grass, etc.).

An entity material is represented by the *EntityMaterial* class which defines the following properties of the material:

[2]The default material has ID 0. See list of materials in Appendix D.

- **Semantic ID** - unique identifier of the material.
- **Name** - the name of the material.
- **Friction** - value of the surface friction used in the vehicle simulation (this value is used only for simulating the friction between the tyres and the underlying surface, it is not used in the collision resolution).
- **Sound material** - defines the sound properties of the material. A more detailed description is in Section 15.4.
- **Bullet hits properties** - properties of the material when it is hit by a bullet (used by the *Bullet* class).
  - **Create bullet holes** - controls whether bullet holes are created on the material.
  - **Bullet hole model** - name of the model that is used for bullet holes. The model should must contain only single rendering entity - the decal used for the bullet holes. Value is used only when the holes are created.
  - **Create particle effects** - controls whether a particle effect is created.
  - **Particle effect name** - name of the particle effect used in bullet hit.
- **Contact material** - properties of the material when it is colliding with other entity (used by the *PhysicsManager* class, details can be found in Chapter 8).
  - **Create particle effects** - controls whether a particle effect is created when the entity collides with other entity.
  - **Particle effect name** - name of the particle effect used in collision.
  - **Hardness** - hardness of the material. Particle effect is created only if the entity has harder material than the other (or with the equal hardness).
- **Skid marks material** - properties of the skid marks created by the sliding tyres (used by the *Wheel* and *SkidMarksCreator* classes).
  - **Create skid marks** - controls whether the skid marks are created on the material.
  - **Skid marks material** - name of the scene dump (see Section 6.4) with the material for the skid marks. The scene contains only one render object, the material of this render object is used.
- **Wheel contact material** - properties of the contact between the material and the wheel (used by the *Wheel* class)
  - **Create wheel contact particle** - controls whether a particle effect (such as mud or dust) is created when a wheel is in contact with the material

- **Wheel contact particle effect** - name of the effect used when wheel is in contact with the material, the effect must be continuous
- **Create wheel slide particle** - controls whether a particle effect (such as tyre smoke) is created when a wheel slides on the material
- **Wheel contact particle effect** - name of the effect used when wheel slides on the material, the effect must be continuous

The entity materials use a data-driven approach. A material is defined by a xml file located in the "data\surfaces" folder. When the game is initialized all xml files from this folder are loaded and the materials from them are added to the *EntityMaterialManager*. Note that the xml files contain serialized instances of the *EntityMaterial* class.

Chapter 6

Fibix

6.1 Introduction

We are using Fibix Technology[1] for rendering, level design and managing the visual content pipeline. It has two logical parts: a rendering library and the Fibix Editor. The rendering library is used in MotorDead to display the 3D scene and the 2D UI elements. The editor is used by our designers to create levels, prepare game entities (see Chapter 16 and Appendix E) and design particle effects. The particle effects system was not part of Fibix engine in the beginning of the project, we had to add it ourselves (see Chapter 7).

6.2 Rendering library

The Fibix renderer is written in C++ and wrapped in C++/CLI. As an effect, there is some marshalling of data needed when communicating with the C++ layer from the C# layer, which takes up about 4% to 5% of frame time. Notable Fibix classes that are used in the C# layer are: *C_EngineWrapper*, *C_Scene*, *C_Viewport* and *C_RenderEntity*.

The main communication point between the game and the rendering engine is the *Renderer* class. The *Renderer* class is held as a singleton in the *FightRaceGame* class. Other components can access the instance using the *Renderer* property of the *FightRaceGame* class (it is one of the game services, see Section 4.5).

6.2.1 C_EngineWrapper

C_EngineWrapper is a singleton class that provides central access to the renderer. It is responsible for initialization and cleanup of the engine as well as for loading models and textures and changing some rendering options (e.g. resolution), etc. Clearly, it does not follow the single responsibility principle[2]. It was the class where all "miscellaneous" functions and utilities were put. The instance of the *C_EngineWrapper* is accessible from the *Renderer* class.

[1] `http://fibix.eu/`

[2] `http://en.wikipedia.org/wiki/Single_responsibility_principle`

6.2.2 C_Scene

C_Scene represents a 3D scene that can have entities added to it or removed from it. We use a singleton scene in MotorDead located in the *Renderer* class. We change its contents completely when the displayed scene is changed.

6.2.3 C_Viewport

C_Viewport represents a camera in the 3D scene - it can be moved, rotated, its field of view can be changed, etc. We use a singleton viewport in MotorDead located in the *Renderer* class. *C_Viewport* also functions as a debug-drawer. We call its methods *DrawLine*, *DrawSphere* and *DrawText* (see Section 6.7) for debug-draw purposes.

6.2.4 Render entities

C_RenderEntity is an abstract class for graphical entities. It has various derived classes representing, for instance, solid objects, light sources, particle emitters, etc. There is no in-built support for hierarchies (scene graph[3]) of render entities, so we had to implement one ourselves (see Section 5.2.1).

C_RenderObject is derived from *C_RenderEntity* and represents a solid object. It can have different levels of detail[4], at each level a different mesh and material. *C_RenderParticleEmitter* is discussed in-depth in Chapter 7.

6.3 Render material

Materials are a simple concept used not only in Fibix but in all rendering engines. A material is a collection of properties related to the appearance of a surface of an object. These properties include various kinds of textures (diffuse, specular, normal, emissive, etc.), reflection types (environmental, planar or none), transparency, and so on.

Note: not to be mistaken with the concept of a semantic / physical material. The connection between the two lies in the SemanticID property of a render material (see Section 5.7.)

6.4 Dumps

Fibix editor allows saving the current content of the editor scene into a single file. Such a file contains all render entities from the scene (i.e. render objects with a geometry, lights, particle emitters, etc.) and the setups for the materials of the render entity. This file is called a "model dump" and it represents one model used in the game. If the dump file contains additional settings for the scene (e.g. the background texture, etc.), it is called a "scene dump".

[3] `http://en.wikipedia.org/wiki/Scene_graph`
[4] `http://en.wikipedia.org/wiki/Level_of_detail`

Dump files can be viewed as models prepared in the internal format of Fibix. Fibix engine is able to load a dump file and render its content. Naturally, dump files are used also in MotorDead. All models and scenes used in the game are distributed in the form of dumps.

A dump file contains all data necessary for the rendering of the model or scene it stores (e.g geometry data, material setups, etc.). The only exceptions are the textures. The textures are stored in the separate files and the dump files reference them using file names.

6.5 Model

In MotorDead, models are stored in the dump files (see Section 6.4). One model dump file represents one model. In the game, a model is represented by an instance of the *Model* class.

A model dump file may contain (and often it does) multiple render entities. The render entities are positioned in the editor and together they form a model that is displayed in the game. The *Model* class contains list of the model's render entities and uses the transforms of the render entities from the dump as the local transforms. When the position of the model is changed, the *Model* class multiplies the given world transform and the local offsets to get the world transform of the model's entities. This approach moves the render entities (of the model) in a way that the whole model stays intact.

The *Model* class contains the *Duplicate* method. This method is used for instancing. The *Duplicate* method duplicates all render entities of the model. The world transform of the duplicated model is set to identity matrix. The local offsets of the render entities are preserved.

Render entities are represented by classes from the engine wrapper. These classes wrap native C++ classes that allocates unmanaged memory. Such a memory must be released explicitly. Therefore the *Model* class contains the *Destroy* method. This method is called when the model is no longer to be used. The *Destroy* method destroys all render entities of the model.

6.5.1 Dummy points

Among the render entities of a model might be dummy point entities. The dummy points are used to mark certain positions in the model (e.g. mounting position of a wheel on a vehicle's model). A dummy points is defined by its transform and name. The name is important, as there is no other way to determine the meaning of a dummy point, only by its name.

Dummy points are named using the following naming convention. The format of the name of a dummy points is: "<Type>:<Arguments>;". The argument part is not compulsory. If it is missing the name looks like: "<Type>;". Both the type and the arguments if the dummy point are string used to determine the particular dummy among the others.

Which dummy points must be present in the model depends on how the model is used in the game. A model of a vehicle has different dummy points than a model of a weapon. The particular dummy points can be found in Appendix F. For example, the dummy point with the "Wheel:FR;" name defines position of the front right wheel.

Note that the ";" semicolon character at the end of a name is compulsory. This is needed because Fibix requires unique names of the render entities in the scene. Fibix editor, which used

for preparing data, automatically adds number suffix to prevent the multiple usage of the same name. The ";" character is used to filter out the number suffices. As a result, it is possible to work with multiple dummy points with the same name in one model.

6.6 Model factory

The *ModelFactory* class is used for loading the dumps files (both model and scene dumps) and instancing of the models. The models can be retrieved using the *GetModel* method. The *GetModel* method automatically instantiates the models. The *ModelFactory* class keeps a list of previously loaded models. If the wanted model has been loaded already, the *GetModel* method only duplicates the model from the list. If the wanted model is not in the list, the *GetModel* method loads it, adds it to the list and then duplicates it.

Note that the instancing is applied only for the model dumps. Scene dumps are not instanced and always loaded. Since the scene dumps are loaded in only a few specific situations (e.g. before the race session), it is not a issue.

The *ModelFactory* class allows pre-loading of the models, for which it is certain that they will be instantiated in the game. For example the model of a rocket fired from the rocket launcher. It can't be loaded with the first shot, because reading from the hard drive would slow the game down. Therefore it must be pre-loaded in the initialization. This is possible by using the *PreLoadModels* method of the *ModelFactory* class.

Currently, there is only a limited number of models in MotorDead. Therefore it is not a problem to have all the models loaded and to instantiate them. However, with more content, the size of the memory needed for all of the models might get too high. In that case, before a race session it would be necessary to load only the models needed for the session and clear them after the session ends.

The *Model* class holds unmanaged resources. The *Destroy* method of the *Model* class must be called when model is no longer to be used. This also applies to the loaded models kept in the *ModelFactory* class. Destroying of these models is done by the *ClearPreLoaded* method of the *ModelFactory* class.

Model factory is one the game's services. It is accessible from the *FightRaceGame* class using the *ModelFactory* property. For more information about the services, see Section 4.5.

6.7 Fonts

Fibix provides the method *DrawTextColFont* which is used for drawing texts, both debug and GUI. This method is called from a *C_TextureFont* wrapper class *FibixFont* and accepts font and color as arguments.

Fonts in Fibix consist of two information:

- character atlas texture

  uncompressed 32-bit *.tga*, square, size is a power of two (512,1024, etc.)

grid of 16x16 characters (0-255)

characters aligned left

- character widths

  .wth file, containing widths of individual characters, in two bytes each.

  represents character's width in pixels in a 512x512px texture (widths are automatically recalculated for larger textures).

It is possible to create and add custom fonts simply by adding the above mentioned files to the Fonts folder, like this:

```
<installation folder>\Data\Fonts\<font name>\<font name>.tga
<installation folder>\Data\Fonts\<font name>\<font name>.wth
```

If a font fails to load (any file missing or invalid), it is ignored and unavailable at runtime.

Note: Because of the support of 32-bit TGA textures (which contain more information than in the 8-bit white-and-transparent format), Fibix is capable of drawing outlined fonts.

Chapter 7

Particle effects

7.1 Introduction

Particle effects are an extremely important building block for creating high-fidelity visuals in any modern 3D game. They are lively, dynamic and add a lot of life to any virtual world.

In MotorDead, anything that isn't solid geometry is modeled as a particle effect. These include volumetric phenomena such as smoke, fog or explosions but also glowing objects such as collision sparks or a bullet in mid-air.

In this chapter we will take a close look at what possibilities we offer to the visual effects designer creating the effects for the game, how the particle effects module works at low-level and how it interfaces with the higher-level game code.

The source code for the particle effects can be found in the `src\Particles` folder.

7.2 Terms

Before we start, let us explicitly define several domain-specific terms that will be used throughout this Chapter.

Particle
: The most primitive building block of any particle effects system. It's represented graphically as a textured billboard [8] with a position, velocity, size, etc. Even line-like particles (e.g. a bullet in mid-air) are represented as elongated billboards.

Particle emitter (emitter)
: An entity that generates particles of one kind (with the same material 6.3 and sharing common properties).

Emission
: The event of creating a new particle by an emitter.

Particle effect

A composition of particle emitters that can be offset in space and orientation with respect to the effect's origin. They can also be offset in time (e.g. one emitter starts emitting two seconds after the other one has). An example of a complex particle effect is an explosion: it can contain emitters for fire, sparks, smoke, glow and even pieces of dirt (Figure 7.1).

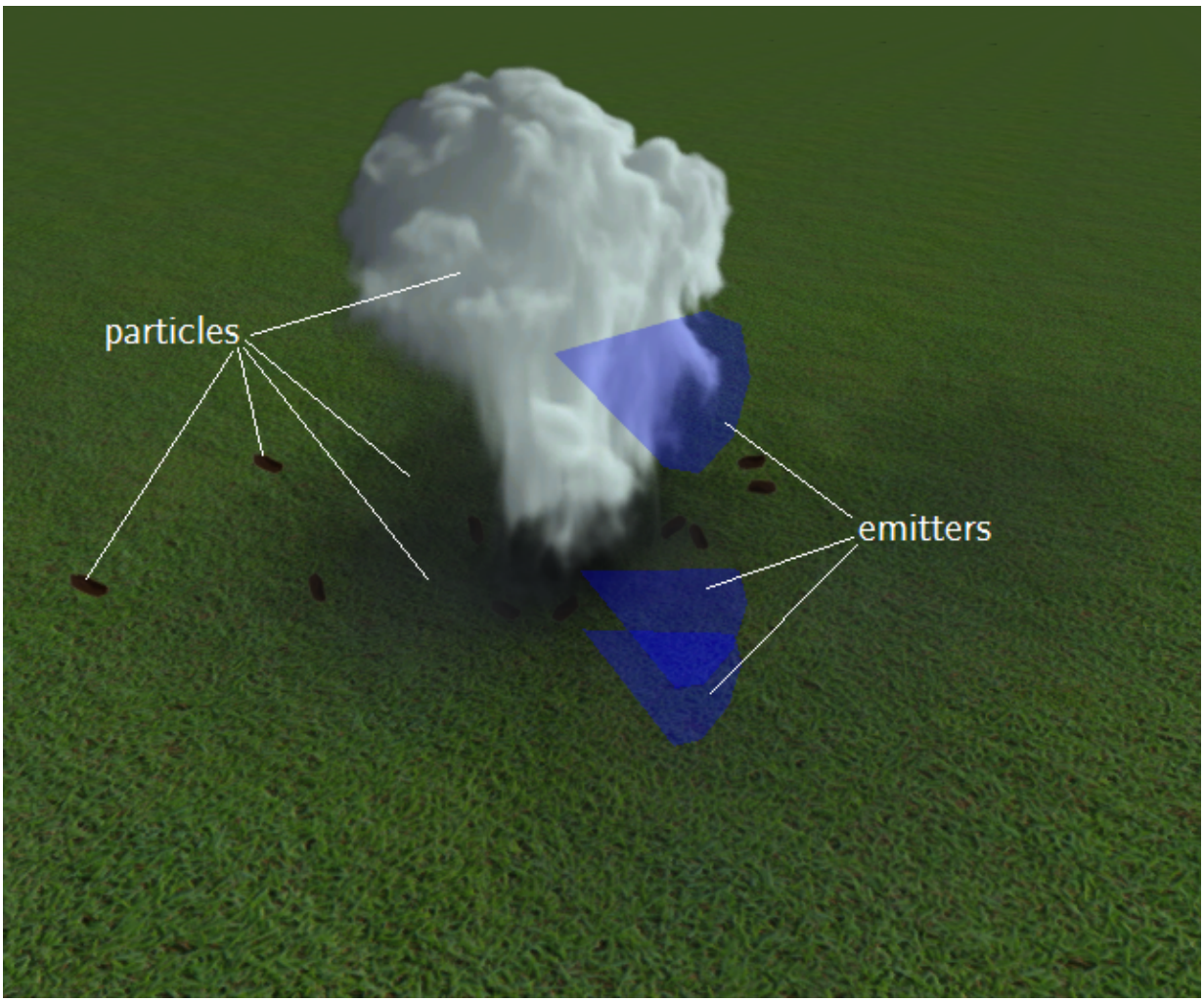

Figure 7.1: An explosion particle effect from MotorDead. It contains three emitters: explosion smoke, dust and shrapnels.

7.3 Features

This section contains an outline of the features of the particle effects system, from the perspective of the designer. In the beginning, our visual effects artist (Jan Kudrnáč, see Section 1.1.2) provided a specification of the features he would like to see in the editor. After several iterations

— in the analysis stage, in the implementation stage and while he was already designing effects — we have ended up with the following list of features: (all of the class members (names starting with *m_*) from this Section can be found in the *C_RenderParticleEmitter* class, a star means there are more members with different suffixes)

Emitter properties

These include essential properties to be able to create a wide variety of particle effects:

- Local-space / world-space particles (*m_IsEmittingLocal*)
- Emit rate per second (*m_EmitRate*)
- Maximum number of emitted particles (*m_MaxEmitCount*)
- Size of the emitter (we emit inside a box volume) (*m_EmitOffsetLocal*)
- Time offset before the start of emitting (*m_TimeOffset*)
- Particle material (see Section 6.3) (*m_MaterialResDesc*)
- Multipliers for color and alpha (transparency) of the particles. These properties seem to be part of the material, but they are important for the "Interpolation" feature (see below), because the material does not change during the lifetime of a particle (*m_Color**, *m_Alpha**)
- Particle size in game units (*m_Size**)
- Particle rotation (*m_Rotation**)
- Particle life in seconds (the lifespan of particles created by this emitter) (*m_Life**)
- Initial velocity of particles in game units per second (*m_VelocityStart**)
- Acceleration applied on particles in game units per squared second (*m_Force**)

Stochasticity

A lot of the visual attractiveness of particle effects comes from adding randomness to most of the above-mentioned properties. The way this works from the perspective of the designer is that instead of specifying a single value for a property of an emitter, you can specify a range of values.

For example, instead of defining an (x, y, z) vector for the initial velocity, the designer defines two vectors and the value used for an actual emit is a vector of per-component uniformly-distributed random values from the interval.

Interpolation

Another crucial feature for creating a particle effect is to be able to specify change in a particle's properties during the lives of individual particles. For instance, you might want your particles to grow in size, rotate at a certain speed and decrease their alpha (make it more and more transparent) during their lifetime – so that they fade away naturally.

In most commercial editors, this is made possible using a curve editor – specifying key-frames and values at these keys, etc. For our purposes, this would have been an overwhelming task we did not necessarily need. We decided to simply add an option to specify a start value and an end value for every property. The resulting value is obtained by linear interpolation between the start and the end value, based on the particle's current life.

Combined with the randomness, some emitter properties had four values to be filled in (e.g. particle size: start min, start max, end min, end max, see Figure 7.2), but for most of them, two values are enough.

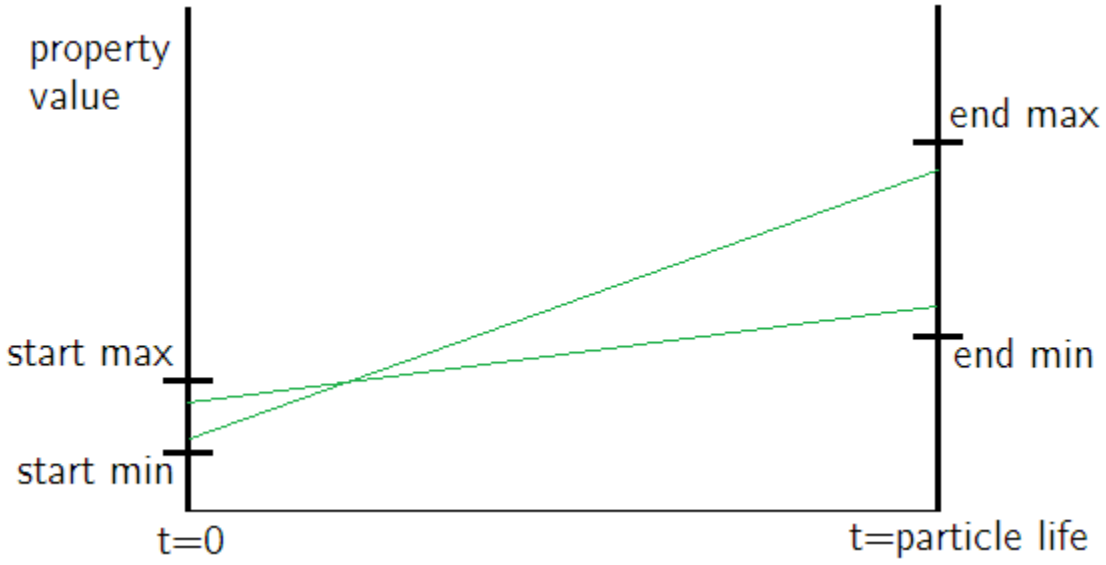

Figure 7.2: By specifying 4 values for a property: start-min, start-max, end-min and end-max, the designer defines a range of possible linear functions of time that give a value of a property at a given time during the particle's lifetime. Two possible resulting functions are shown.

The resulting interface is far from complex, which was our primary goal (see Figure 7.3).

Axis-alignment (*m_AxisAligned*)

In order to emulate the volumetric appearance of a particle effect, the particle geometry is rotated so that its plane is facing the camera. This is called billboarding. There are two types of billboards:

- Point
- Axis-aligned

Point billboards rotate freely around their position to face the camera. Axis-aligned billboards have an axis defined, around which they rotate to face the camera (Figure 7.4). As

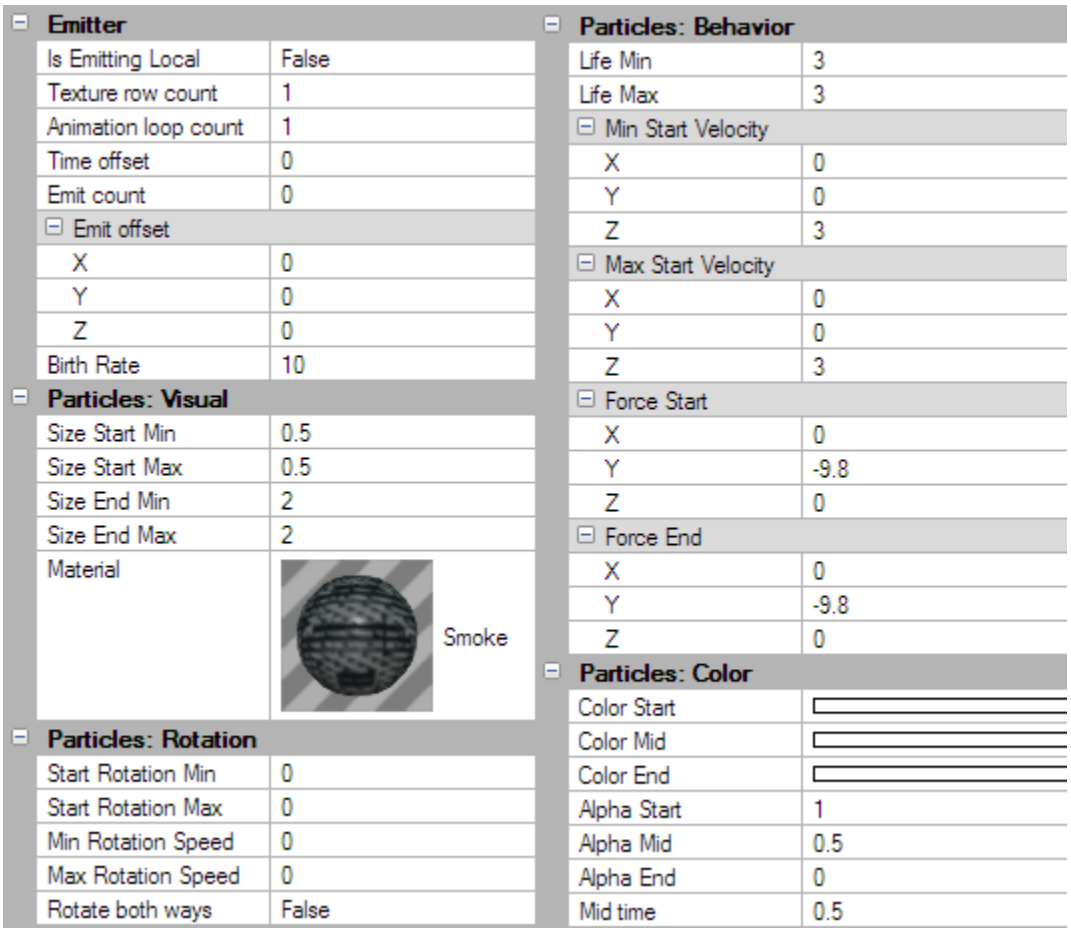

Figure 7.3: The particle emitter interface as seen in the Fibix editor 6.

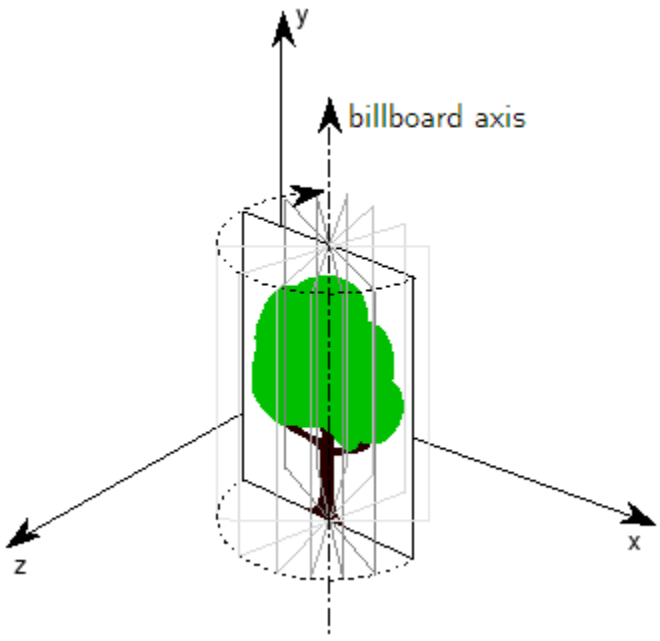

Figure 7.4: An axis-aligned billboard rotates around its axis to face the camera. Reproduced from [9]

a result, if the camera-to-billboard vector and the billboard's axis are parallel, the billboard cannot be seen.

There is a simple true/false switch for an emitter, specifying whether it should emit point or axis-aligned particles. In the case of axis-aligned emitter, the particles' axis can be specified explicitly or inferred from an individual particle's velocity. In the latter case, the particle is also stretched along the direction of its velocity by its speed. This is useful for particles such as sparks.

Animated texture

The last important feature is the ability to animate a particle's texture during its lifetime. This is done using a grid texture and specifying the number of windows it contains in the emitter. The resulting principle is similar to using an animated .gif image as a texture. See Figure 7.5 for an example of an explosion texture (4x4 animation frames).

7.4 Implementation

As was mentioned in Chapter 6, the particle effects system is the only part of our work that was done in the Fibix engine. This also means that it's the only code we have written in C++ (instead of C#). This is why there are different naming conventions: the names of classes begin with the prefix *C_* and names of structs begin with the prefix *S_*.

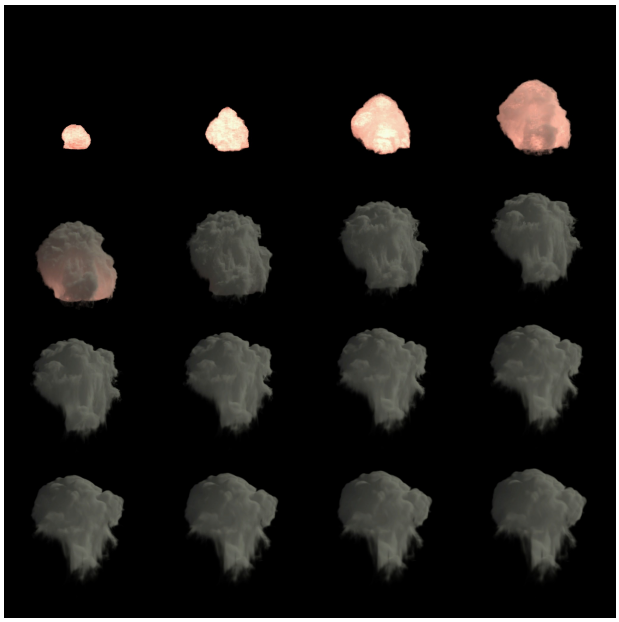

Figure 7.5: The texture used for one of the emitters of the explosion particle effect in the game. The frames of the animation go row by row, from top to bottom, and inside a row from left to right.

Most of the functionality of the particle effects system is contained in two classes: *C_RenderParticleEmitter* and *C_ParticlesModule*. The former represents an actual emitter in the game world – it has a position, orientation and all the properties described above. The latter serves as a manager for all emitters that are present in a scene.

Note: RenderParticleEmitter.h, RenderParticleEmitter.cpp, ParticlesModule.h and ParticlesModule.cpp files are packaged with the source code of MotorDead. They are, however, part of the Fibix Engine, so they will not be compilable.

7.4.1 Particles Module

At the C++ level, Fibix provides a modular way of adding new subsystems to the engine. The way to do this is create a class inheriting from *C_ExtendedModuleBase*, override the methods that we need and register an instance of such module. For the purpose of particle effects, we created the *C_ParticlesModule* class that does exactly this. It keeps track of all particle emitters, manages the GPU data used for particle billboards and pools all of the particles themselves.

The most important method of this module is *Update()*, which is called every frame by the Fibix's module system. In this method, the module iterates through all emitters and calls their *Update()* methods. Then, it sorts the particles in every emitter according to distance from the camera, so that they are rendered in the correct order[1].

[1]Transparent geometry needs to rendered back-to-front in order to give realistic appearance [8].

The particles module is also responsible for creating and updating the GPU data used for drawing the particles. For all existing emitters, we only use one vertex and index buffer. Each emitter contains the offset into the index buffer so that the particles are drawn correctly. The index buffer is created once and stays the same, because every particle is a separate quad (four vertices, two triangles).

Since particles are created, die and move independently in every frame of the game, the vertex buffer is dynamic. This means that unlike for static geometry, the particle vertex data is updated and sent to the GPU in every frame. This is done in *FillBuffers()* method which is called from *Update()*. The vertex buffer is filled with the particle's four updated vertices. Every vertex contains the particle's position, billboard axis, vertex color and texture coordinates. The texture coordinates that are sent to the GPU include the logic for animating the texture's frames (see *C_ParticlesModule::FillBuffers()*).

Because of performance reasons, the module keeps a pre-allocated pool of particles. When an emitter needs to emit a new particle, it asks for a reference to an unused *S_Particle* structure (which is an index into the pool array) from the module (*AllocateParticle* method) and returns an index when a particle dies (*DeallocateParticle* method). The pool has a fixed size given by a constant N_{max}. If there are no free particles available (all N_{max} are being used), the *AllocateParticle* method returns false, and no particle is created.

7.4.2 Emitter

C_RenderParticleEmitter inherits from *C_RenderObject* which is a graphical entity with a model. This gives the emitter spatial properties, such as position and orientation. The *C_RenderParticleEmitter* itself defines members for all the properties we have described in Section 7.3. Of course, an emitter needs to keep track of the particles it emitted, and does so in a dynamic array (vector) of indices. These are indices to the particles pool (discussed in the previous paragraph) – an index is merely a reference to a particle.

Virtually all of the particle's non-visual logic is done in the emitter's *Update()* method. The emitter checks whether new particles should be emitted and does so if required. It asks for a new *S_Particle* struct from the *C_ParticlesModule* and initializes it according to the emitter's properties. The actual particle update logic takes place in the *UpdateParticles()* method, where the particles' positions, velocities etc. are integrated using the Euler method. Also, the interpolation of other particles' properties takes place here (e.g. alpha fade-out, size change). Moreover, the emitter checks for any dead particles (their remaining life is less or equal to zero) and deletes them. Deleting a particle means swapping its reference (its index) with the last one in the vector of indices and then trimming the vector by one. This index has to be returned to the *C_ParticlesModule* because it is now unused and ready for a new particle.

An important member that *C_RenderParticleEmitter* inherits from *C_RenderObject* is *LODSet*, which serves for referencing graphical data as described in the third paragraph of Section 7.4.1. Among other things, it holds two values that are interesting for us: the starting offset of the index in the buffer and the number of triangles belonging to this emitter. We use four vertices and two triangles per particle (see Figure 7.6). That means the offset to the index buffer for a given emitter is $6N$, where N is the number of particles that we already filled in this call of the

FillBuffers() method. Then, the number of triangles for the current emitter is $2M$ where M is the number of particles belonging to this emitter.

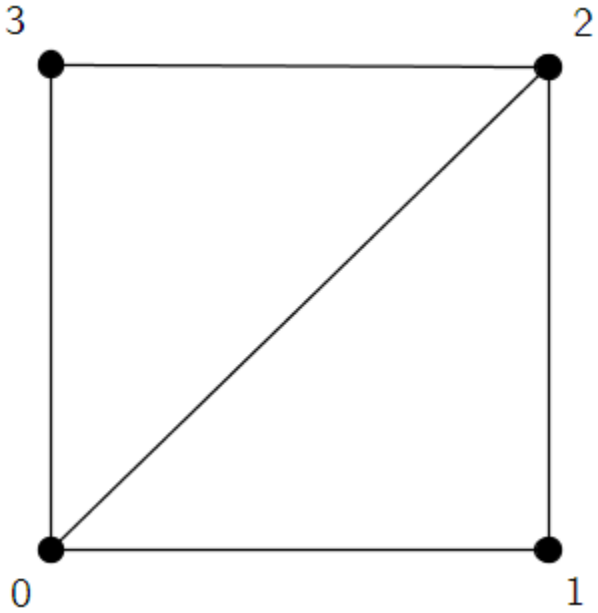

Figure 7.6: Four vertices and two triangle faces constitute a particle's graphical representation.

7.5 Usage and the C# wrapper

In this section, we will be talking about the C# code that uses the C++ classes described above via a wrapper in C++/CLI. A high-level perspective is that a single particle effect is composed of several emitters. Once an artist designs an effect in the Fibix editor, he exports it and we can load it inside the game. For this, the *ParticleEffectsManager* service is used, by calling its *CreateEffectInstance* method, which takes as an argument the name (identifier) of an effect and returns an instance of the *ParticleEffect* class. For the list of all particle effects that were used in the game, see Table C.1 in the Appendices.

Apart from creating particle effect instances, the *ParticleEffectsManager* class is also responsible for cleaning up finished particle effects. In every frame, the manager checks all living instances of effects whether they are finished and whether all of their particles are dead. If so, the instance is deleted from the game world and its memory cleaned up.

There are two kinds of particle effects by design: one-time (e.g. an explosion) and continuous (e.g. exhaust fumes). For the former, it can be automatically determined when it is finished. In the latter case, another object is responsible for stopping the effect (e.g. a rocket would stop emitting its trail after it hits something). A *ParticleEffect* class contains the *AutoStop* property to determine whether it is one-time (*true*) or continuous (*false*). The value of this property is

inferred when creating an effect from its emitter, but its value can be changed in the case when we would like to reuse a one-time effect (e.g. a muzzle flash on a machine gun).

A *ParticleEffect* is really only an array of wrapped *C_RenderParticleEmitter*s. After an instance is created, the effect is automatically inserted into the rendering scene (see Chapter 5). It contains the following properties and methods:

WorldMatrix
: Change the effect's position and orientation by assigning to this value.

Start()
: Starts or restarts the particle effect.

Stop()
: Stops the particle effect for good, marking it ready for automatic deletion after its remaining particles die.

PauseEmitting()
: Pauses the emitting of particles from all the emitters in the effect.

ResumeEmitting()
: Resumes the emitting of all the emitters in the effect.

AutoStop
: Determines whether the effect should be automatically stopped (and hence deleted) after all of its particles have died. Discussed in the third paragraph of Section 7.5.

Intensity
: Multiplies the alpha modifier (Section 7.3) for all of its emitters by the desired value clamped to the interval $[0.5, 1]$.

SpeedProvider
: By assigning an object that implements the *ISpeedProvider* interface to this property, the emitters' emit rate and life become dependent on the speed of this object. The faster it moves, the higher the emit rate we use (so that that there aren't gaps between particles) and the lower life the particles have (so that we keep the effect's particle count the same). This approach has proven quite successful for effects pinned to vehicles, such as engine smoke when the car is damaged.

Chapter 8

Physics simulation

8.1 Physics simulation basics

In this chapter, the general principles of integrating and using the physics simulation in MotorDead are explained. The vehicle simulation, which is a substantial portion of the game, is described in Chapter 11.

Using physics simulation in the computer games has become a standard in recent years. It is a powerful tool. Use of a physical simulation brings a great advantage in that the behaviour and movement of the objects in the game look very natural. Moreover it is still possible to turn off the simulation for the certain objects and tune them as needed in a non-physical way. Therefore the physics simulation is also used in MotorDead as the basis for modelling of the vehicles, weapons and other objects in the game.

Two most common tasks of the physics simulation in games are collision detection and rigid body dynamics. Collision detection is used to detect the overlaps between the simulated objects. Rigid body dynamics manages the movement of the objects including resolution of collisions among objects. Collision detection is one of the inputs of rigid body dynamics. It is used to determine the objects that are colliding (i.e. in contact). Rigid body dynamics then resolves these collisions.

As the name suggests rigid body dynamics only deals with the rigid objects. The shape of such an object remain the same during its whole life. On the other hand the soft body dynamics resolves not only the collisions, but also the deformations of the objects. In MotorDead only rigid bodies are used, as this approach is much more faster.

A rigid body is an object that has defined shape and inertia properties - mass and moment of inertia. Mass can be viewed as resistance to linear acceleration. Linear acceleration is a result of forces that act on the body in its center of mass. Moment of inertia is resistance to angular acceleration which is a result of forces acting in the points different from the center of mass. Dynamic properties of a rigid body (position, rotation, linear and angular velocity) are updated by rigid body dynamics. For more information about the rigid bodies in general, see book Game Physics Engine Development [10] and about the rigid bodies in Bullet physics, see Tutorial [3]. It

is highly recommended to get familiar with the concept of rigid body dynamics before modifying the source code dealing with the physics simulation.

8.2 Units convention

In MotorDead the following units convention is used. All units are taken from the SI table (i.e. International system of units, the metric system). The derived quantities are also given in metric system (e.g. the dimensions are given in metres, the time in seconds, therefore the velocity is given in ms^{-1}). The quantities and their units are summarized in Table 8.1.

Physical quantity	Unit
Mass	kg
Size (dimension, distance)	m
Time	s
Force	N
Torque	$N \cdot m^{-1}$
Velocity	$m \cdot s^{-1}$

Table 8.1: Units of the most common physical quantities.

8.3 Bullet physics library

As mentioned in Section 8.1 the MotorDead uses rigid body dynamics as the basis of the physics simulation. There are many libraries that can be used for this task. The decision to use Bullet physics was motivated by multiple reasons:

- Bullet is well known, actively developed and it is stable and fast.
- A wrapper (Bulletsharp) exists that allows using Bullet from .Net code.
- Both Bullet and Bulletsharp have user friendly licenses that don't require a project using the libraries to be open source (see details in Section 2.1.3).

In MotorDead, the collision detection and the rigid body dynamics parts from Bullet are used. Bullet also contains soft body dynamics but this functionality is not used in the game.

The Bullet physics library is used through the Bulletsharp wrapper. The library is built outside the main project and added as external reference. Please note that in the whole document the class names from the Bulletsharp wrapper are used, although they are presented as a part of Bullet. This convention is used because it better corresponds with the source code of MotorDead.

8.4 RigidBody

Bullet's most used class is the *RigidBody* class that represents one simulated rigid body. Rigid body is also the most common physical form of the physical entities. Therefore, it is also present as a protected field *_rigidBody* in the base class *PhysicsEntity*. The value of this field may be *null* if the physical form of the entity is different from rigid body. For instance, weapons attached to the vehicle don't contain rigid bodies - they don't collide with other objects.

Often it is necessary to determine the physical entity represented by a certain rigid body. Bullet's solution of this task is the *UserObject* property of the *RigidBody* class. *UserObject* has type *object*, so any managed object can be passed as value. In MotorDead a convention is used that each *PhysicsEntity* sets reference to itself as *UserObject* of its *_rigidBody*. Therefore for each rigid body value of *UserObject* can be safely cast from *object* to *PhysicsEntity*.

The environment in which the rigid bodies exist is represented by another Bullet's class - *DynamicsWorld*. All rigid bodies are added to the *DynamicsWorld*. It is responsible for the collision detection and the rigid body dynamics.

8.5 Physics manager

Note that everything that is said about the *SimulationWorld* class is also equally valid for the *GameWorld* class. The *GameWorld* class is a descendant of the *SimulationWorld* class. For more details about the *SimulationWorld* and *GameWorld* classes, see Chapter 5.

The *PhysicsManager* class deals with physical properties of the physical entities that are in the *SimulationWorld* class. The *PhysicsManager* class holds an instance of *DynamicsWorld* that represents the physics representation of the game world (i.e. *DynamicsWorld* is the physics scene of the game). The instance of the *DynamicsWorld* class is initialized in the constructor of the *PhysicsManager* class using the settings from the Tutorial [2].

Each entity that is added to the *SimulationWorld* class is also added to the *PhysicsManager* instance. This is done by calling the method *AddToPhysicsManager* on the added entity. This method can do any operation that is needed. The default implementation (in the *PhysicsEntity* class) adds the entity to the *PhysicsManager* class and it also adds entity's *_rigidBody* to the *DynamicsWorld* of the *PhysicsManager*. Other operation is, for example, adding ghost objects for detecting collisions (see Section subsec:Rocket Launcher).

From this point the entity's rigid body is updated in the *DynamicsWorld* class. Similar procedure is done for the removal of the physical entity from game world with the *RemoveFromPhysicsManager* method of the *PhysicsEntity* class. The relationship between the *SimulationWorld* and *PhysicsManager* classes is showed in Figure 8.1.

8.6 Substep callback

The *PhysicsManager* class is updated in the *Update* method of the *SimulationWorld* class. The *PhysicsManager* updates the *DynamicsWorld* instance by calling its *StepSimulation* method.

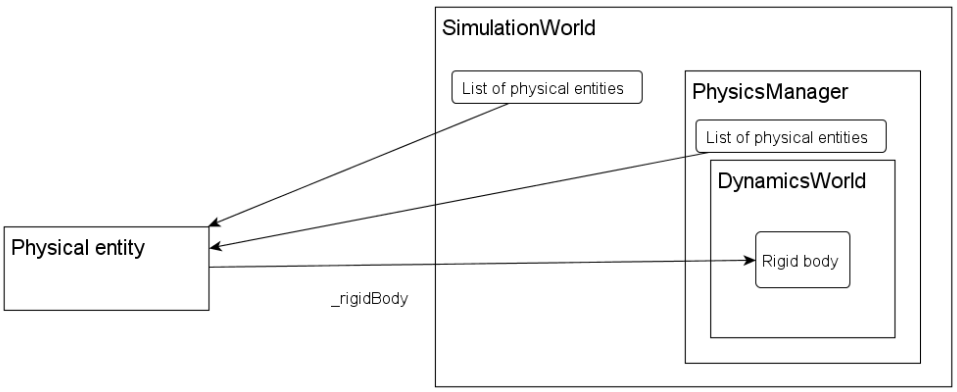

Figure 8.1: The context of the *PhysicsManager* class. Note that a physical entity is referenced by both the *SimulationWorld* and the *PhysicsManager* class.

The physics simulation runs with a constant update time. This is necessary to keep the physics simulation stable. The Bullet physics library has a good support for this functionality. The *StepSimulation* method automatically splits an update call into smaller internal substeps if the desired update time is too long. Note that, it is often necessary to do extra operations in every simulation substep (i.e. at the constat rate of the physics updates). The word “extra” means game specific operations that are not done by Bullet.

The extra operations in the physics simulation substep are done by using Bullet’s substep callback mechanism. The *PhysicsManager.UpdateCallback* method is registered in the *DynamicsWorld* class using its *SetInternalTickCallback* method. The *UpdateCallback* method is then called automatically every time the *DynamicsWorld* does a simulation substep. The callback is registered as "pre-tick" callback, meaning that it is called before every simulation substep.

In the *UpdateCallback* method of the *PhysicsManager* three operations are carried out:

- Update the physical entities added to the physics manager by calling *PhysicsUpdate* method on each of them. In the *PhysicsUpdate* the physical behaviour of the entities is implemented.
- Determine the high level information about collisions between physical entities. Details can be found in Section 8.7.
- Update "substep callback updatables". For more information, see Section 8.8.

For more information about update of the *DynamicsWorld* class and the *StepSimulation* method, see Tutorial [5]. For more information about substep callbacks, refer to Tutorial [4].

8.7 Entity collisions

The collisions among the rigid bodies that represent the physical entities, are resolved automatically. As a result, the movement of the rigid bodies respects the collisions. However, it is necessary to create high level information about the collisions at the level of physical entities.

The method for determining the information about collisions among the rigid bodies is taken from the Tutorial [1]. Result of this method is list of all contact points currently present in the *DynamicsWorld* class. A contact point is formed between each colliding pair of the rigid bodies. However, it is possible that there are more contact points between two rigid bodies (e.g. one of the bodies is concave). These contact points are transformed into the high level information, by creating collision events.

In MotorDead the information about the entity collisions includes four different types of events. All of them are implemented as virtual methods of *PhysicsEntity* class. *PhysicsManager* calls these methods to inform the entities about the collisions. It is called on the both entities that are in contact. The events are summarized in Table 8.2.

Method name	Description
CollisionStarted	The pair of entities is in contact but it wasn't in the previous physics update.
CollisionJustStarted	The pair of entities is in contact but it wasn't in the previous physics update. Moreover the contact for this pair of entities has been already found during the current physics update. The *CollisionStarted* method has been called for other contact point.
CollisionContinues	The pair of entities is in contact and it also was in contact in the previous physics update.
CollisionEnded	The pair of entities is not contact but it was in the previous physics update.

Table 8.2: Entity collision methods

The detailed information about each collision is present in the parameters of the methods mentioned in the Table 8.2. Specifically, the parameter of type *CollisionArgs* can be used to determine various contact details.

8.7.1 Collision driven effects

The collision detection at the physical entities level is also used for triggering additional effects - sounds and particle effects (for general information about sound and particle effects, see Chapters 15 and 7 respectively). The triggering mechanism of both effects is implemented in the *PhysicsManager* class.

A particle effect is played for the contact point for which the *CollisionStarted* methods are called. The materials of the contacting entities are used (for details, see description of the entity materials in Section 5.7). Each material has a defined hardness. The values of hardness are compared and a particle effects is played only for the physical entity with a softer material (or for both of them if they are equally hard).

For example consider the collision between a wooden crate and a gravel surface. The expected result is that the contact will raise a cloud of dust. Therefore the hardness of the wooden material is set to a higher value than the hardness of the gravel. On the other hand a contact between the same wooden crate and a concrete surface should produce only small bits and pieces of wood to come off the crate. Therefore the hardness of the concrete has higher value than the hardness of the wood.

The particle effect is not played if the material of the entity doesn't produce particles. Although the triggering is only approximate, the results are satisfactory for the computer game.

A similar approach is used also for playing collision sound. The difference is that for playing sounds are also used the contact points that cause the *CollisionJustStarted* event. For each pair of colliding entities the collision impulses are accumulated and the resulting value is used to choose the sample that is played (more details are in Section 15.4).

Contact points that produce *CollisionContinues* calls are not used to trigger sounds. These collision should more often create scraping sounds that are more difficult to control then the simple collision sounds. Scraping sounds are triggerd only for the continuous contacts that involve vehicles (see Section 15.6).

8.8 Substep updatables

The substep updatables provide a mechanism that allows any game component to be updated during the substep callback (and therefore with the same constant update time as physics simulation). An updatable can be added to the *PhysicsManager* class using its *AddSubStepCallbackUpdatable* method and removed using the *RemoveSubStepCallbackUpdatable* method. Each updatable (added to the *PhysicsManager* class) is updated by calling the *Update* method of the *IUpdatable* interface. An Example of a substep updatable is the *SkidMarksManager* class, a class that manages skid marks left by the tyres of the vehicles (see Section 12.10).

8.9 ContactAdded callback

ContactAdded is a static event of the Bullet's class *ManifoldPoint*. It is raised whenever Bullet detects a new contact point between the rigid bodies in the physics scene. The properties of the contact are given by the parameters of the event. Moreover, the values of the properties can be changed. For instance, the direction of the contact normal can be set arbitrarily. This is one way of defining and tuning the custom behaviour of the physics. There are three handlers for the *ContactAdded* event used in MotorDead:

1. Single sided triangle mesh callback - described in the Subsection 8.9.1.

2. Vehicle friction callback - changes the friction in the collisions involving vehicles, described in Section 11.15.

3. Vehicle's wheels collisions - deals with the collisions between a wheel and other objects (mostly the static geometry), described in Section 11.15.

All of the handlers of the *ContactAdded* event are called for each of the contacts. It is up to the handlers to choose the contacts that should be changed.

8.9.1 Single sided triangle mesh callback

The static geometry of the track (i.e. the road and the surrounding buildings) is represented by the triangle mesh collision shape. A mesh represents only the surface of an object, therefore the collision detection can easily miss a collision between a mesh and a small, fast object. Small and fast object is a typical scenario in MotorDead, as the collision shapes of a vehicle is formed by multiple smaller collision shapes (for more details, refer to a description of the vehicle's collision shape in Section 11.13).

By default, the triangle mesh collision shape produces collisions from the both sides of the mesh. A collision between a vehicle and a static triangle mesh collision shape may lead to the situation (depicted in Figure 8.2) where one small shape of the vehicle is on side of the mesh, while another shape (of the same vehicle) is on the other side. The collision for the first shape has been missed due to the discrete nature of the collision detection. In such a situation the vehicle becomes stuck in the static geometry, because the impulses from the collisions point in opposite directions. To prevent the vehicle from being stuck, only collisions from one side of the triangle mesh must be considered. This is done by altering the direction of the contact normal, so that the normal points in the same direction, regardless of the side (of the mesh) on which the collision shape of the colliding object is.

Unfortunately, the callback only solves the issue for contacts with the relatively big objects. If an object, formed by a triangle mesh, is very thin, the two sides of the object create the same situation as a double sided mesh. This issue is difficult to solve and only occurs very rarely. Therefore, it is left to the artist to create tracks without the thin objects.

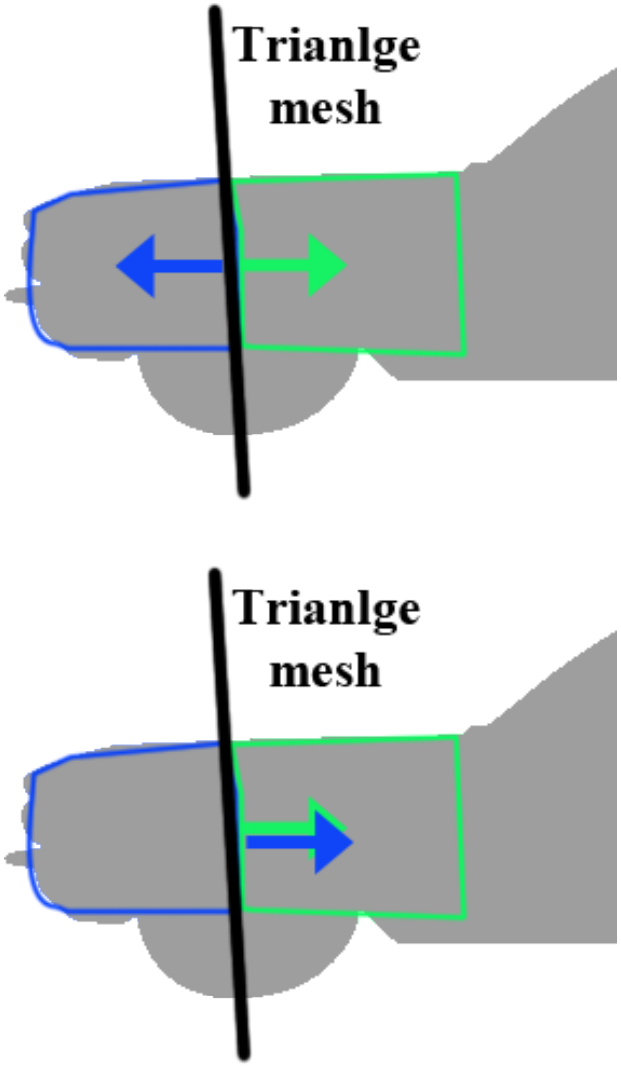

Figure 8.2: The top image shows two partial collision shapes (highlighted with blue and green) of the vehicle stuck in the double sided triangle mesh collision shape (black). Note the contact normals that have opposite directions. The lower image shows the same situation with single sided triangle mesh - the normals have same direction.

Chapter 9

Input

The primary input methods of MotorDead are keyboard input and mouse input. The mouse is used to control the game GUI - navigating the menu screens, clicking a button, etc. The keyboard is used for the actual gameplay - acceleration, steering, shooting the weapons, etc. There is also a very basic support for using an Xbox 360 Controller instead of the keyboard.

9.1 Getting the input

For getting the actual input from the devices, we use the XNA's *Microsoft.Xna.Input* namespace. It contains the *KeyboardState*, *MouseState* and *GamePadState* types. We will call these three states 'input state'. In every frame, our *GameInput* class updates the input state by calling the XNA's *GetState()* methods of the *Keyboard*, *Mouse* and *GamePad* static classes.

Note these are states, which means XNA does not give us any events. Therefore, we store the input state from the last frame, in order to know whether, for instance, a new mouse click has happened – it did if the *LeftButton* property will have a *Pressed* value in the current frame mouse state and a value of *Released* in the previous frame mouse state. This specific piece of logic is implemented in the *GameInput.IsNewLeftMouseKeyPressed()* method.

9.2 Key mapping

It is common in keyboard-controlled games to give the player an option to change the input mapping of the keyboard keys. For this purpose, we implemented a key mapping system. The semantic commands such as 'steer left', 'brake' or 'fire weapon' are defined in the *GameKeys* enumeration. The mapping of the *Microsoft.Xna.Framework.Input.Keys* enum to the *GameKeys* enum is stored in the *KeyMapper* class and can be changed in the options screen by the player (see *OptionsMenuScreen* class). Note that the mapping of the gamepad keys cannot be changed at the moment.

There is neither an option nor a need to map mouse buttons to game controls, since the mouse is only used for navigating the game menus.

Chapter 10

GUI

This chapter describes the structure and features of the GUI library used in the MotorDead, as well as the modifiactions we have done.

10.1 Foundation

Our GUI library is founded on XNA Simple Gui (DGui) [7], a simple XNA library which provided basic functionality and inner logic for controls, such as buttons, labels, etc. Drawing and update methods had to be significantly modified. The main reason for this was that the library originally used XNA renderer for drawing, especially Texture2D, which supports manual runtime texture creation (painting) - a feature that DGui made use of. We had to separate DGui from XNA (drawing, but not all utilities) in order to use our renderer, which only supports loaded textures.

10.1.1 DGui license

DGui was released under LGPL v2.1 [1]. For MotorDead, which uses (dynamically links) the modified library, the consequences are:

- The modified library must be released under LGPL v2.1.
- The source code of the modified library must be provided with each copy of the library.
- Modified files must carry prominent notices of changing the files and the date of any change.

10.1.2 Including DGui

The license of DGui requires it to be an independent library, therefore our modified DGui contains interfaces for rendering textures (*IGuiTextureManager*, *ITexture2D*), fonts (*ISpriteFont*,

[1] http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

IFontManager, see Section 6.7), sounds (*I2DSoundPlayer*), coordinate system (*ICoordinateManager*, see Section 10.6) and input (*IInput*). These interfaces are implemented in our project and use Fibix (See Chapter 6).

10.2 Controls and tools provided

The library provides several GUI controls (Panels), such as:

- Panel; ancestor of all controls, no functionality
- Button; classic and toggleable ("sticky")
- Label
- Checkbox; with text
- Scrollbar
- Listbox
- Combobox (dropdown)
- Progressbar
- Textbox; single- and multi-lined (Text field and Text area)
- Image Panel; holds a single image, supports rotating the image
- Form; a form (Panel in WinForms) that can contain other controls
- Layout tool; for organizing controls into a grid (simplifies positioning)

In addition to those, taken from DGui, we implemented our own controls

- Carousel; a revolving selector (image flow)
- RadioButtons; a tool for uniting a toggle button group into one control. Only one button in group can be clicked at a time
- Scrollable Form; extension of a form that can have bigger space for controls inside and can be scrolled
- Slider
- Number Counter; a control displaying a number using monospaced digits

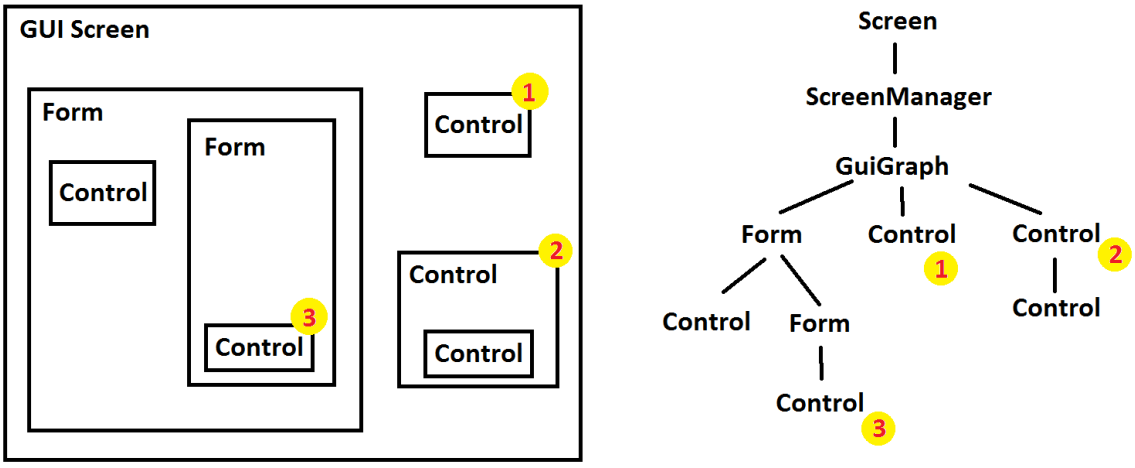

Figure 10.1: Structure of *Controls* in a *Screen* viewed by a user (left) and represented in a tree (right). A simple standalone control (1), a composite standalone control (2) and a simple control in a structure of forms (3). The *ScreenManager* node is described in Section 10.4

10.3 Hierarchy of controls

This section describes the idea of composing simple controls into more complicated strucutres, i.e. composite controls, form structures and screens (See details about screens in Section 4.4), as depicted in Figure 10.1.

10.3.1 GuiGraph

GuiGraph is an oriented tree structure containing all controls in a domain. Each node has a *Parent* and a collection of *Children*. There are three ways a control can be a member of a *GuiGraph*:

- It can stand alone. It is added directly to the *GuiGraph* as a child of the root node.
- It can be a functional member of a different control, called *Composite control* (described in Section 10.3.2, e.g. a checkbox has a button for toggling and a label for text). Then it is a child of that control.
- It can be organized in a structure of Forms. This situation is similar to the one above but Forms are specially designed to contain many unrelated child controls (including other forms).

Either way, each control needs to be a member of the *GuiGraph*.

10.3.2 Composite controls

Some controls have complicated functionality and DGui achieves it through joining simple controls into larger units - *Composite controls*. A simple example is the *Progressbar*, which has one

child control - the progress indicator (the structure is depicted in Figure 10.2). The progressbar changes the size of its progress indicator to present the required progress.

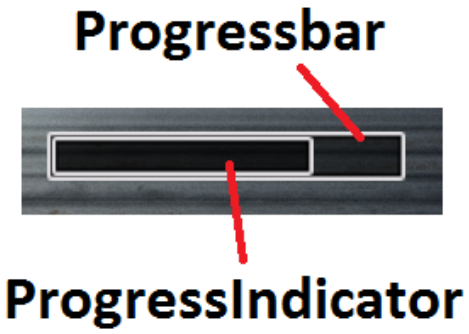

Figure 10.2: Screenshot of a Progressbar, where the value is 75%. Both Progressbar and its *ProgressIndicator* are controls, but together they provide the required functionality. More about appearance in Section 10.5.

10.3.3 Forms

Forms are controls designed for containing other controls. Unlike in .NET's WinForms, where forms are invidiual windows, here they are merely tools for the manipulation of sets of controls within the GUI graph - logical grouping, collective positioning, etc. All forms are drawn with renderer, inside the game window, just like regular controls.

10.3.4 Screens

The content to display is organized into Screens (described in Section 4.4). There are two types of screens: GUI type and World type. World type screens contain a scene (background scene or gameplay) and are rendered first (this topic is described in Chapter 4). Screens of the GUI type contain GUI content like forms or single controls. These screens are used for menus, message boxes and HUD.

The game uses a stack of screens (ScreenStack), which allows easy navigation between menus (a newly opened submenu lies atop of all previously opened menus, disappears when closed).

10.4 Update and draw

The most important methods, when it comes to GUI, are Update and Draw. They are both called from the main loop by a descendant of *Microsoft.Xna.Framework.Game*. The calls propagate

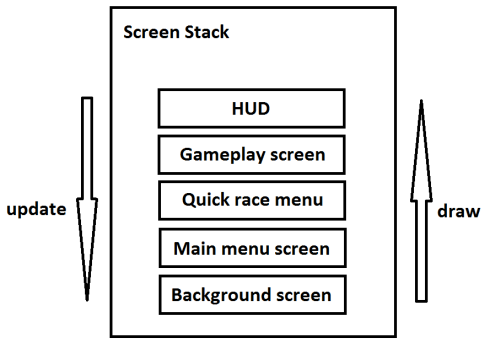

Figure 10.3: Scheme of a ScreenStack. More about Update and Draw in section 10.4.

through the game to its *ScreenStack* (See Section 4.4), which calls them on separate screens, which call them on the *ScreenManager* (which is a structure for managing the *GuiGraph*). All these classes have methods called *Update* and *Draw*.

In each *ScreenManager*, both methods recursively (breadth first) pass through the *GuiGraph*, which is a tree structure containing controls (See Section 10.3.1). The breadth first pass determines the order in which the controls are updated and drawn.

Draw is called in every frame. Update might (depending on performance) be called more often, but at least once per frame. Each control may override methods of *Panel* (the base class for all controls) to implement its own behavior.

Update(GameTime)

In this method, input is processed by each control, events are invoked (e.g. clicking), changes are made (often depending on GameTime, for example cursor blinking). More details in Section 10.4.1.

Draw(GameTime)

In this method, the controls are drawn. This method is called once in every frame. Although this method is virtual, it is only overridden in Panel, where it calls *DrawControl()*. *DrawControl()* is overridden in several classes and implements special beahvior of a control (concerning drawing). The changes are in *Label* (only text, no textures) and *ImagePanel* (only one texture, possibly rotated). Otherwise, the *Skin* (More on that in the section 10.5) is drawn.

Drawing text is realized by calling *Draw* on control's *ISpriteFont*. The implementation of drawing texts is described in section Fonts 6.7.

Drawing textures

GUI textures are managed by *GuiTextureManager*. The manager provides textures by name and loads them if they are not loaded yet. Supported formats are $.jpg, .bmp, .dds, .tga$. Textures

for controls should be united under a *Skin* (See Section 10.5), but it is possible to draw a texture manually by calling *Draw* on the *ITexture2D* wrapper object. More about drawing single textures in Section 10.7.

10.4.1 Hover and focus

There are properties in the *ScreenManager* - *FocusedControl* and *HoveredControl*. These variables are considered global in the scope of the screen and denote the last active and topmost hovered control (respectively).

In each update, *HoveredControl* holds the logically topmost (highest update index) control, which the mouse is hovering over. This property is used to resolve the situation where more controls overlap (for example child control or two badly positioned buttons). Although each control is updated individually, they all ask for *HoveredControl* first to make sure no other control acts like hovered.

After each update, the *FocusedControl* is refreshed. If the current *HoveredControl* was clicked (the mouse was hovering over and the left mouse button was pressed), the control is set to be the new *FocusedControl*. Otherwise, the *FocusedControl* remains to hold the last active control and persists to the next update. Each control behaves individually and independently in its Update method. However, they might ask for the *FocusedControl* to find out whether they are supposed to do something special (e.g. only a focused button is supposed to process a click).

*Note: This system is a leftover of the original library. The downside is the fact that focused control is determined at the **end** of each update and is used for the next call of Update(). A more elegant apparatus might be implemented, but this works.*

10.4.2 Foreground controls

The tree structure of normal GUI controls (meaning all but those special controls, described in this section) is a subtree of the *GuiGraph* and its root is called the *ControlsNode*. However, it is not the root of the entire tree. There is a special node in the *GuiGraph*, the *ForegroundControlsNode*. The structure is still a tree, but the *ForegroundControlsNode* is a "right brother" of *ControlsNode* and is (because of the breadth-first left-to-right pass through the *GuiGraph*) therefore updated and drawn **after** all the normal GUI controls. *ForegroundControlsNode* allows the GUI to have a special layer of controls, which are drawn last and are therefore topmost. This is used in *Combobox* (See Figure 10.4), where the dropdown would otherwise be under later added controls.

10.4.3 Input

This section describes how the GUI works with the information from input devices (keyboard, mouse, Xbox 360 Controller). DGui has the interface *IInput*, which has two properties - XNA's *KeyboardState* and *MouseState*. More about input in Chapter 9.

Each control gets the entire input structure and only uses information that is necessary to decide the control's actions (buttons being pressed etc.).

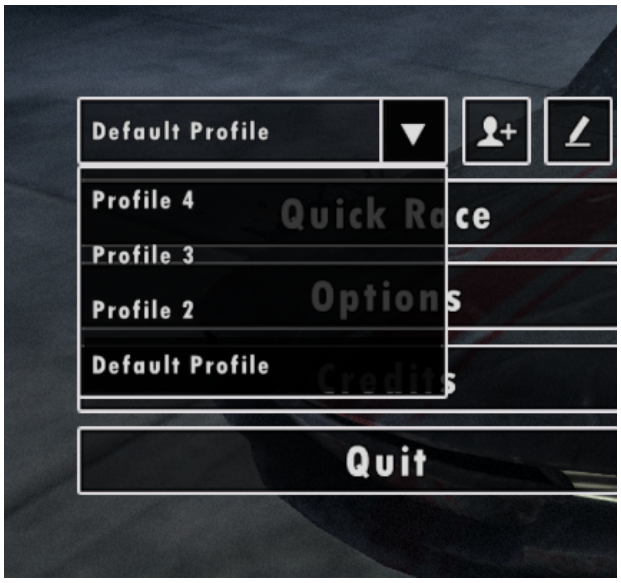

Figure 10.4: Screenshot of a dropdown control of a *Combobox*, which is a child of the *ForegroundControlsNode* and is therefore drawn **after** all normal controls.

10.5 Skin

Every control has a visual appearance. It consists of a *Skin* (which is a texture composition) and optionally other controls, which are painted later (on top of this control, see Section 10.3.2 about Composite controls). Skin manages several textures to compose the right impression of a control (body, borders, corners, etc.). Each implementation of *Skin* provides specific functionality. The point of this visualization mechanism is to prevent simply stretching one texture, which is ugly.

All controls are considered rectangular. Therefore, their shape is described with *Vector2 Size*. The skin stretches its textures on the screen to fulfill the size at the (absolute - see Section 10.6.1) position of the control. Some controls need to look different: to look non-rectangular, child controls are used, or the *Skin* can be made invisible (e.g. label has a text, but intentionally no background).

Note: Making the Skin invisible does not have the same effect as making the entire control invisible, which prevents drawing the control itself ***and*** *its descendants.*

Skin also supports three states - default, hovered over and clicked. Specific behavior depends on chosen implementation.

10.5.1 3x3 grid

This skin is designed for button-type controls. It has borders, corners and inner body. Border width and height are customizable. This skin uses four textures - center, (top) horizontal border, (left) vertical border and (top left) corner, as seen in Figure 10.5. In the *SkinState.Default* state,

these textures (Figure 10.5) are displayed. When it comes to hover or clicking, this implementation has two variations. They both have *ITexture2D* members *InsideHovered* and *InsideClicked*.

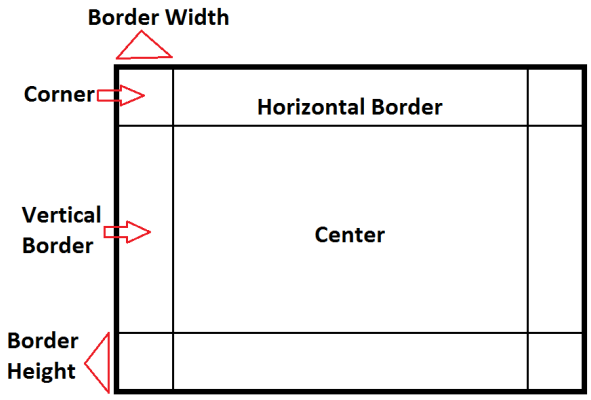

Figure 10.5: Structure of a 3x3 grid skin. Bottom cells are filled with horizontally flipped textures, right cells are filled with vertically flipped textures. Bottom right corner texture is flipped both ways.

When hovered (clicked),

- *SkinTritextural3X3Grid* draws the hovered (clicked) texture **instead** of the default texture.
- *SkinOverlay3x3Grid* draws the hovered (clicked) texture **over** the default texture, with transparency (RGBA tint color - alpha = 100 (200) by default).

10.5.2 Repetitive skin

This skin is designed for stretched controls and those which should look segmented (for example Slider). This skin is either horizontally or vertically aligned.

The *SkinRepetitive* has properties to set thicknesses of both textures (regardless of alignment; meaning width for horizontal alignment, height for vertical alignment).

Note: When the repetitive skin is created vertically aligned, it loads textures with file names with "Vert" suffix. This is to let the 2D designer provide different textures for vertical and horizontal repetitive skins.

10.6 Coordinate system

We decided to abandon the pixel coordinate grid (as used in DGui) and use normalized (float) values. The values run between $[0..1]$ for the shorter side and $[0..ratio]$ for the longer side of the screen. This approach allows positioning regardless of screen resolution and ratio and prevents unintentional stretching of controls.

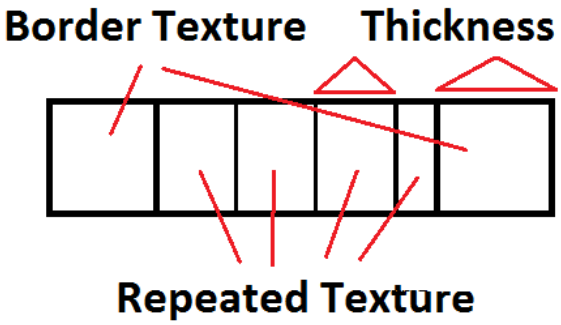

Figure 10.6: Structure of a repetitive skin for a horizontally aligned control. The *Repeated Texture* is repeated from left to right, until it would begin farther to the right than where the right border texture begins. In the case of vertical alignment, the repeating goes from top to bottom. The right (bottom) border texture is vertically (horizontally) flipped.

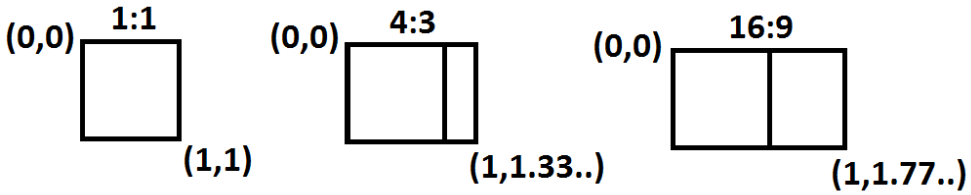

Figure 10.7: Depiction of the idea of normalized coordinates

All controls are rectangular and are positioned by their upper left corner. The size and position of controls are the same units.

10.6.1 Positioning

Each control has a *Vector2 Position*, which denotes the position relative to the control's parent (in the *GuiGraph*, described in section 10.3.1). Before each *Draw* call, the actual position, denoted by *AbsoluteTransform*, is calculated recursively.

The current maximal coordinate is always available (*_screenManager.CoordinateManager*'s properties *ScreenSizeNormalized* and *ScreenCentre*) to allow anchoring (and sizing) controls to right/bottom edges. For example:

```
button.Size = new Vector2(0.2f, 0.05f);
   // fixed size (relative only to the smaller side of window,
   // independent on ratio)
```

```
button.Position =
   _screenManager.CoordinateManager.ScreenCentre
       // the center of the screen
     - button.Size / 2f ;
       // centered by center of the button
```

Form has the method *GetAlignedPosition* to provide positioning within it.

```
button.Position =
   _form.GetAlignedPosition(
   HorizontalAlignment.Right, VerticalAlignment.Bottom,
   button.Size,.1f,.1f);
```

To be able to use the method *GetAlignedPosition* to its full potential, *Layout* has the property *SizeExpected* which estimates the size of the layout grid before it is filled with controls.

10.7 DrawTexture

This method, in the wrapper identified as *DrawTexture2D* in the *C_Viewport* class (described in Section 6.2.3), draws a single texture to the viewport. It is used by GUI controls to draw their *Skin* as described in Section 10.5. There are two points about the *DrawTexture* method that are noteworthy in the documentation.

TransformMatrix

Fibix supports transformation argument of a *Matrix3F* matrix type. This structure contains information about the location and the rotation of the texture.

Coordinates

The coordinate system of our application is explained in Section 10.6. However, the renderer uses (for its own reasons) a different system - the grid is normalized (both coordinates run between $[0..1]$) and the `y` (height) coordinate origins at the **bottom** of the screen. Visual comparation of both systems is shown in the Figure 10.8.

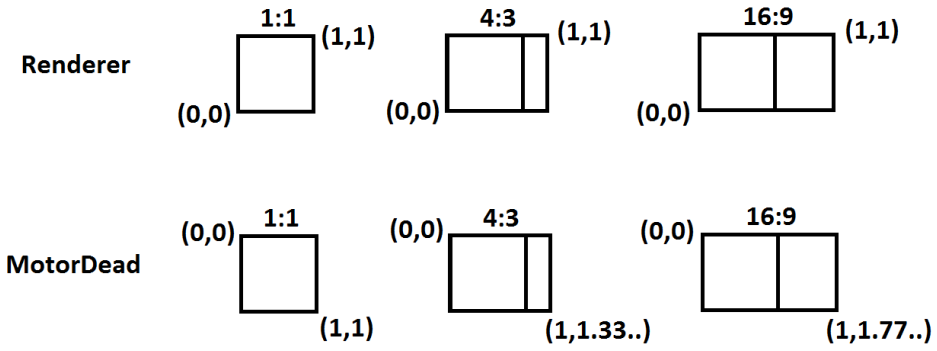

Figure 10.8: Comparation of 2D coordinate systems of the Fibix renderer and MotorDead GUI

Chapter 11

Vehicle simulation

11.1 Introduction

The principles and methods used to simulate vehicles in MotorDead are described in this chapter. The game logic part of the *Vehicle* class is described in Section 12.11.

A physical entity representing a vehicle is implemented by the *Vehicle* class. (derived from *PhysicsEntity*). It contains the implementation of the vehicle's physics and gameplay properties. Since the vehicles are the center point of MotorDead, the *Vehicle* class is very complex. It is separated into three source files using the .Net mechanism of partial classes.

Methods used to simulate the behaviour of vehicles are physically inspired but there are many details that are neglected, simplified or deliberately treated in a physically unrealistic way. This is done to make the behaviour of the vehicles predictable, easier to control and provide the user with a more pleasant gaming experience.

Note that, although Bullet physics library (Bullet is used for the collision detection and rigid body dynamics in the physics simulation) contains the implementation of the vehicles, in MotorDead we use a custom simulation of vehicles. Having custom vehicle simulation gives more control over tuning of the vehicles behaviour. The vehicle simulation source code can be located in the MotorDead source code.

11.2 Ray-cast vehicle

A vehicle in the simulated world is represented by its chassis which is one rigid body. The task in the simulation is to model the forces that are exerted on this body. These forces include:

- Suspension forces from the suspension springs and dampers
- Friction forces from the tyres (these include the torque from the engine and the brakes)
- Aerodynamic drag force

Note that *Vehicle* class contains the field *_chassis*. This field references the same rigid body as the *_rigidBody* field. The *_chassis* field is used for convenience in the physics related code.

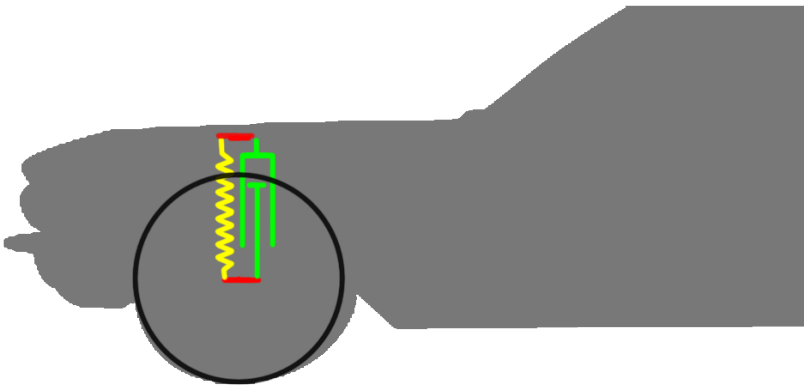

Figure 11.1: Mounting of the wheel to the chassis using the spring (yellow) and the damper (green).

A ray-cast vehicle is a concept often used in the situations when the speed of the vehicles is relatively high and the small details of the (track's) surface can be neglected. Instead of modelling the wheels as rigid bodies that are connected to the chassis with the spring joint, each wheel and suspension is modelled using a ray-cast. The forces are computed from the properties of the ray (e.g. distance from the ray's origin to the hit point). Ray-casting and computation of the suspension and friction forces are implemented in the *Wheel* class.

In the physics scene a ray is cast from the point where the suspension is mounted to the chassis in a downward direction (see Figure 11.2). The ray-casting procedure returns the point where the ray intersects the scene. Naturally the result might be that the ray doesn't intersect the scene. If the ray hits the scene the forces from the wheel are computed. Otherwise the wheel has no contact with a surface underneath (it is "hanging in the air") and the forces are zero .

11.3 Suspension forces

The suspension forces are exerted by the mechanism that connects a wheel and the vehicle's chassis. This mechanism (depicted in Figure 11.1) consists of two parts: a spring that smooths out the the roughness of the surface and a damper that limits the "bumpiness" of the spring.

As mentioned in Section 11.2, a ray is cast from the wheel's mounting position. From the distance between suspension mounting position and the scene intersection point we can determine current suspension's length (depicted in Figure 11.3). The length of the suspension defines the force the spring exerts on the chassis.

The suspension has three defined lengths - minimum, rest and maximum length. The minimum length is the length of the fully compressed suspension, when the spring produces the

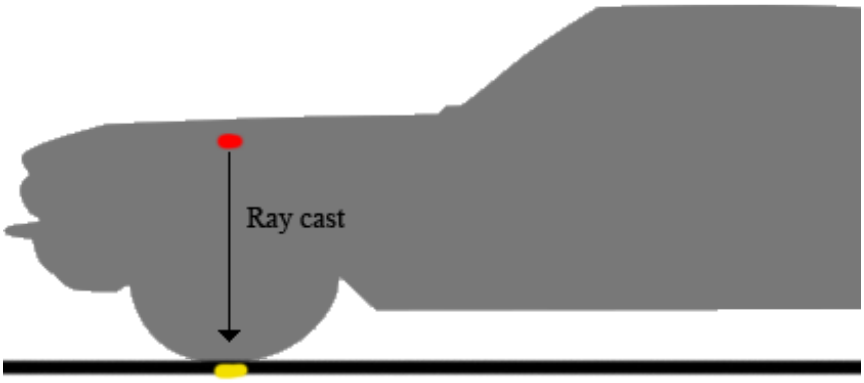

Figure 11.2: Ray cast from the suspension's mounting point.

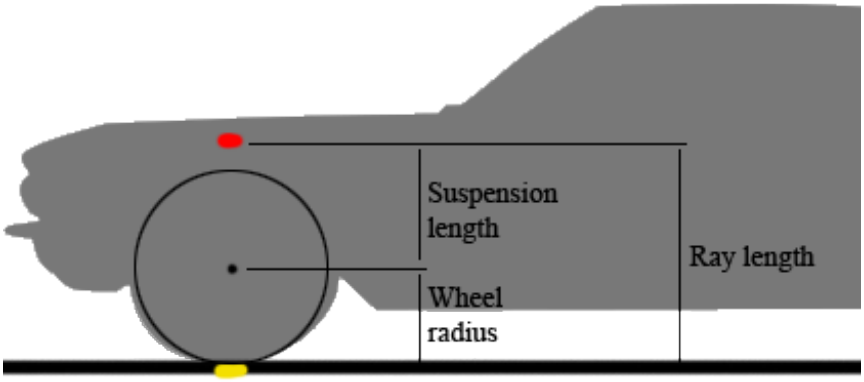

Figure 11.3: Length of the suspension is determined from the distance to the surface.

maximal force. The suspension can't be compressed to a smaller length than the minimum length. That means, even when the ray-cast returns ray length suggesting a shorter length, the suspension's length is clamped to the minimum length. The Rest length is the length when the spring doesn't produce any force. The maximum length, as the name suggest, is the maximum length of the spring. It defines how low will be the wheel hanging when the car jumps and is high above the ground.

The suspension force is computed only when the suspension's length is smaller than the rest length. The formula used to compute magnitude of this force is given in Equation 11.1. It is known as Hooke's law:

$$F_{spring} = stiffness \cdot (restLength - currentLength) \tag{11.1}$$

where $stiffness$ is a coefficient of the suspension that can be tuned. At first, it was set high enough for the suspension to support the chassis at half the distance between the minimum length and the rest length. The experiments showed that even higher values produce better result in high speeds. A meaningful stiffness could be approximated using Equation 11.2. The formula means that each spring supports quarter of the vehicle's mass when compressed to two tenths of its length. However, the value given by Equation 11.2 is meant only as solid starting point for the tuning of the suspension.

$$stiffness = \frac{0.25 \cdot 10 \cdot vehicleMass}{(restLength - minLength) \cdot 0.2} \tag{11.2}$$

The position of the wheel is given in the 3D model of the vehicle (see dummy point explanation in Section 6.5.1 and modelling conventions in Appendix F). This position is interpreted as the wheel's position when the length of the suspension is equal to the rest length. Therefore the suspension mounting point is determined by moving this position upwards by a distance equal to the rest length. This means that a common offset can be added to (or subtracted from) the three suspension lengths. It follows that the minimum length could be fixed to 0 and only the rest and the maximum length are adjusted.

Another important property of the suspension is damping. It models the damper that is attached to the suspension's spring and it reduces the "bumpiness" from the suspension. Damping force is computed using Equation 11.3.

$$F_{damping} = -dampingCoeff \cdot suspensionVelocity \tag{11.3}$$

The higher the damping coefficient the more "bumpiness" is reduced and vehicle is more stable in turns. The velocity of the suspension (i.e. how fast is the length of the suspension changing) is computed using the velocity of the vehicle's chassis in the local point (this includes both linear and angular velocity of the chassis). The position of the ray hit in chassis's local coordinates is used as the local point. Resulting velocity is projected on the suspension direction (i.e. the direction of the ray) and the value of this projection is used in Equation 11.3. For more details about the velocity in the local point, see method *GetVelocityInLocalPoint* of the *RigidBody* class.

The resulting suspension force is computed using Equation 11.4.

$$F_{suspension} = F_{spring} + F_{damping} \tag{11.4}$$

11.4 Friction forces

The friction forces are caused by the friction between the tyres and the surface. For each wheel, there are two separate forces (see Figure 11.4):

- Sideways friction force - responsible for the forces in the wheel's sideways direction (this include the steering)

- Forward friction force - responsible for for the forces in the wheel's forward direction (acceleration and brakes)

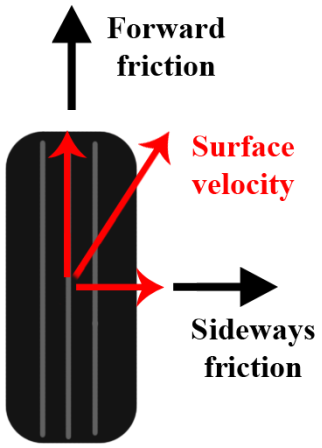

Figure 11.4: Friction forces of the vehicle's wheel - top view.

The sideways friction force compensates the movement of the wheel in the sideways direction. It counteracts the speed of the wheel against the surface underneath in the sideways direction. This speed is computed by taking the chassis's velocity in the local point with the ray hit location as the local point and projecting it on to the wheel's sideways direction. For more details about the velocity in the local point, see the *GetVelocityInLocalPoint* method of the *RigidBody* class.

The forward friction force counteracts the velocity caused by either the torque from the engine or from the brakes. The torque is set to the wheels and it accelerates the rotational movement of the wheel (see Figure 11.5). This acceleration (a_{torque} is used to compute the velocity that should be counteracted by the friction: $v_{forward} = a_{torque} \cdot deltaT$. The torque from the engine and brakes is set to the *Wheel* class by the *Vehicle* class.

The two forces are treated independently but the method used to compute the friction forces is the same. It can be found in [10]. Since the friction forces are treated separately, only the method for the collision resolution is used (although it is adapted for friction).

For a given impulse applied on a rigid body at a given point, the resulting change in the body's velocity can be computed using the mass and inertia of the rigid body. The friction computation method is based on the idea of inverting the task. The change in velocity and the point of application are known and the needed impulse is computed. The change in velocity is the velocity that should be counteracted by the friction and the point of application is the

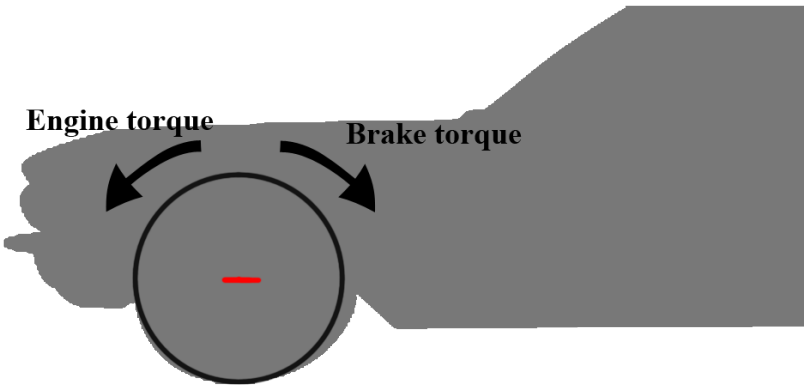

Figure 11.5: Friction forces of the vehicle's wheel

location of the ray hit converted to the vehicle's local coordinates. The computed impulse I is then transformed to the force F using the equation $F = \frac{I}{deltaT}$.

The original method from [10] computes a collision impulse (i.e. the impulse in the direction of the contact normal, that is needed to remove the velocity in the normal direction). The magnitude of the collision impulse depends on the velocity of the rigid body and is virtually unlimited. Adapting to the friction is done by limiting the magnitude of the impulse (and therefore the friction force) according to the force in normal direction and the friction coefficient (i.e. standard Coulomb friction, see Equation 11.5). Note that since the friction forces are computed independently, the friction coefficients for forward and sideways direction are also independent. Use of the independent friction coefficients allows finer tuning of the vehicle's handling.

There are two special cases in computation of the forward friction. The first occurs in the situation when the handbrake is pulled and the wheel doesn't rotate around the axle (it is locked). In this situation, the whole surface velocity in the forward direction is used in the friction computation, not only the velocity change caused by the torque.

The second case is the forward friction coefficient under braking. When the wheel is braking (the value of the *BrakesOn* property of the *Wheel* class is set to *true*), the forward friction coefficient is multiplied by the *BrakingFrictionMultiplier* value. This mechanism is used when the vehicle has installed a brakes upgrade. It ensures that wheel has more friction and the vehicle stops faster. Otherwise the more braking power would be wasted by the lack of friction. Adding such a multiplier is completely non-physical. However, it corresponds with expected behaviour of the upgraded brakes - the vehicle stops faster, regardless of the other upgrades and the surface under the tyres.

$$F_{friction} = F_{normal} \cdot frictionCoefficient \tag{11.5}$$

The value of $frictionCoefficient$ is combined from the value given in the settings of the vehicle (see Section 11.16) and the friction of the entity hit by the ray. If the hit entity is the track the *GetMaterialID* method is used to determine the friction (for more details about the entity materials, see Section 5.7).

$$F_{normal} = 10 * chassisMass * chassisMassDistribution \tag{11.6}$$

The force in normal direction is computed from the vehicle's mass using Equation 11.6. The coefficient $chassisMassDistribution$ is set to 0.25 for each of the vehicle's wheel, but it can be adjusted if necessary. Note that in a strictly physical simulation the suspension force should be used as the normal force. Using of the constant normal force causes the vehicle to be more predictable to control.

11.5 Friction issues

The biggest issue of the friction computation is the fact that the wheels are treated independently. In reality the friction of the wheels is coupled through the chassis.

In case of the forward friction this is not a problem. It might not be physically correct but it doesn't have any negative impact on the gameplay.

However, in case of the sideways friction the independence of wheels causes certain issues. The sideways friction force counteracts the chassis' velocity in the sideways direction. As a result of wheels' independence, the velocity might the counteracted multiple times. This "overshot" friction force gives the vehicle unrealistic velocity in the opposite direction to the counteracted velocity. In the next physics update the velocity might again cause the overshot friction force. This leads to unstable shaking of the vehicle as is showed in Figure 11.6.

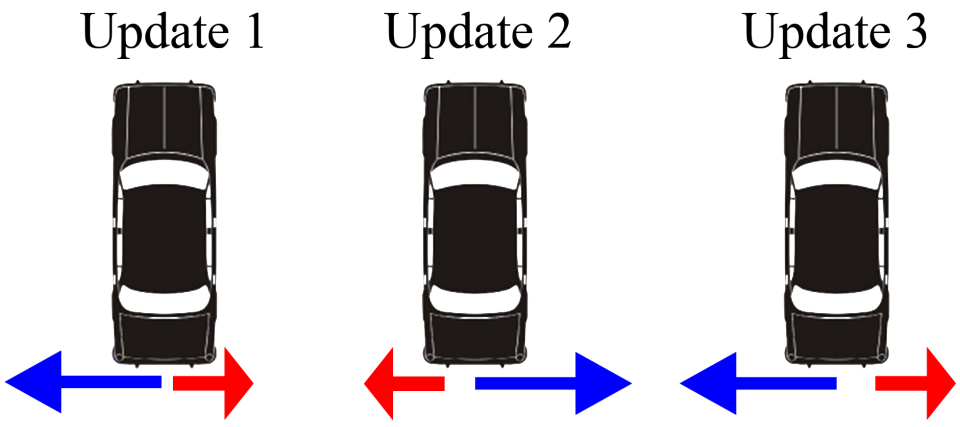

Figure 11.6: Overshooting of the side friction force in 3 following updates, the speed is illustrated by red arrows, the force is blue.

Part of the solution to the problem of overshooting the friction force is using an inertia tensor in the friction computation that is different from the actual tensor of the chassis. The value of the sideways friction inertia tensor is computed in the same way as the chassis's tensor but with the quarter of the vehicle's mass. The chassis appears lighter and the resulting force is smaller. The inertia tensors used in the friction computation can be set to the wheel using *ChassisInvInertiaTensorSideFriction* and *ChassisInvInertiaTensorForwardFriction* properties. For more details about the vehicle's inertia tensor, see Section 11.16.

Second part of the solution is that the torque (of the chassis) that results from the friction forces is limited so it doesn't point in the direction opposite to the steering direction. For example if the vehicle is turning left, the torque can't rotate the vehicle to the right. If the torque rotates to right, it is set to zero.

11.6 Handicap and damage

The vehicle (as entity) has a defined value of health. The health is reduced by collisions and weapon hits. The performance of the vehicle is impaired with the reducing health. This is implemented by reducing the forward friction force when the throttle is pressed (i.e reducing only the force resulting from the engine torque, not from the brakes). The friction force is reduced by the multiplier that depends on the value of vehicle's health (for detail see the *UpdateHealthStatus* method of the *Vehicle* class).

The *Vehicle* class also implements a simple handicap system. Handicap is a multiplier used to either impair or improve the performance of the vehicle. The value of the multiplier is used to multiply the wheels' forward friction forces. In contrast to the health multiplier, the handicap is also used to influence the forces when vehicle brakes (the forward friction force is multipled also when vehicle brakes). Moreover, the handicap multiplier is used to modify the friction coefficient of the vehicle's tyres (see the *ApplyHandicap* method).

Value of the handicap multiplier is set by the *Handicap* property and it is used by the AI system to implement rubber banding. Rubber banding is a very common technique used to keep the opponents near the user (in the terms of race progress). When the opponents are in front of the user, their handicaps are set to slow them down and allow the user to catch up. On the other hand, when the user manages to get away from the opponents, the handicap is set to improve their performance (see details in Section 13.3.2).

11.7 Ray length

In the strictly physical simulation, the lenght of the ray (that is cast for a wheel) should be set to $wheelRadius + suspensionMaxLenght$. It is the greatest distance when the wheel is in contact with the underlying surface. The suspension and friction forces are computed only when the ray hits the scene, otherwise they are set to zero.

At first this strict approach was used, but it made the vehicle difficult to control on a rough and bumpy surface. The bumps cause that often some of the wheels lose contact with surface. While

it is physically correct, sudden loss of friction cause vehicle the to spin. Therefore, a longer ray length is used to give the vehicle more stability: $wheelRadius + suspensionMaxLength + antiSlideMargin$, where $antiSlideMargin$ is a constant set to smal value (currently it is 0.2).

Now the forces are computed even though the wheel is in the air. The stability is greatly improved and since the value of $antiSlideMargin$ is low there are no visual artefacts (i.e. it is not visible that the vehicle turns even though the wheels are in the air).

11.8 Aerodynamic force

There is implementation support for the aerodynamic drag force in the *Vehicle* class. It is modelled using Equation 11.7. The drag force can be tuned by setting the value of the *AeroDragCoefficient* property. The *AeroDragCoefficient* is expected to be equal to $\frac{1}{2}Ac_x$.

$$F_{aero} = \frac{1}{2}Ac_x\rho_{air}v^2 \tag{11.7}$$

Note that the aerodynamic force has turned out to be unimportant and therefore the applied value is always zero. However, the implementation is left in the code for the possible future use.

11.9 Sliding power

Steering of the vehicle is done implicitly by rotating the front wheels around the suspension's up direction. It changes the forward and sideways directions of the wheel and therefore also the directions of the friction forces. This implicit steering mechanism that comes directly from the physics simulation. In addition, an artificial steering force is added. It can give the vehicle more over-steer behaviour. The direction of the force is determined by current direction of steering (i.e. left or right). Maximal magnitude of the steering force is given in the vehicle's EDF file by the option "Sliding power" (for more details, see Section 11.16). However, the actual magnitude is influenced by the vehicle's speed. Handbrake also raises the force. This implements the behaviour expected by the player, that the handbrake helps in the slides. The implementation of the sliding force is located in the *UpdateSliding* method.

11.10 Center of mass position

Position of the vehicle's center of mass is defined using the positions of wheels. The positions of the wheels are given in the vehicle's 3D model (see the modelling conventions in Appendix F). The centroid of these points is computed and "COM offset" is added to get the position of the center of mass. This operation is illustrated in Figure 11.7. COM offset can be changed to tune the handling properties of the vehicle (see Section 11.16).

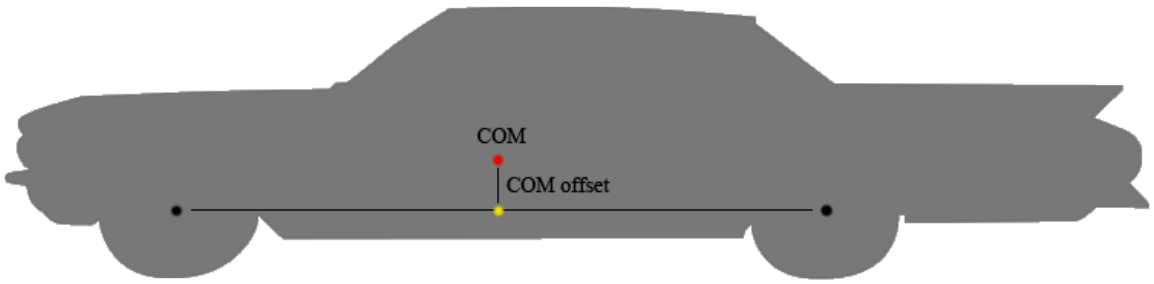

Figure 11.7: Center of mass position (computed using wheels' positions and COM offset).

11.11 Engine torque

An engine is the source of the power that accelerates the vehicle. Each vehicle must have engine assigned to the *_engine* field (using the *Engine* property). Engines are implemented by the class *Engine*. Properties of the engine are defined by the torque curve, which gives the amount of torque for given revs (revolutions per minute). Current revs are set to the engine as feedback from vehicle's wheels. Angular velocity of the wheels is transformed using the vehicle's transmission to the engine's revs (see Section 11.12). The engine is then updated using the *Update* method which returns current torque.

The feedback revs are set to the *Engine* class using the *SetRevsByAngVel* method. The revs value given in the *SetRevsByAngVel* method is not used directly, it is filtered in the *Update* method. The filter prevents from rev changes higher than given threshold, which effectively creates a low-pass filter. The filter is used to simulate the inertia of the engine (i.e. the fact that engine can't change its revs immediately).

The torque from the engine is set to the driven wheels. The torque is distributed evenly to all the driven wheels. The property which wheels are driven is controlled by the vehicle's setting in the EDF file by the option "Drive" (see Section 11.16). There are three different configuration: RWD, FWD and AWD (rear, front and all wheel drive). Unfortunately, however, the differences between the configurations in our vehicle simulation are not very significant.

11.12 Transmission

The *Transmission* class implements a gearbox that assigned to the *Vehicle* class. The gearbox is used to convert the angular velocity of the driven wheels to the angular velocity at the output of the engine (the *TransformAngVelFromWheel* method) and the torque from the engine to the torque at the driven wheels (the *TransformTorqueFromEngine* method).

The attributes of the gearbox are given by the ratios of the gears and the final gear ratio (the multiplier used for all the gears). The number of the gears is given by the number of the gear ratios given in the EDF file that defines the particular gearbox entity.

Note that the *Transmission* class represents a manual gearbox. It doesn't change gear by itself. The mechanism of automatic gear changes is implemented in the *Vehicle* class, in the *UpdateGearbox* method.

11.13 Vehicle deformations

Although only rigid body dynamics is used in MotorDead, the vehicles are visually deformed. This is done by blending two shapes with the same topology. One shape represents the fully damaged state of the vehicle and the other the fully new state. The blending is done by Fibix and it can be controlled by setting the value of the blending parameter. Note that the damage is purely visual and the collision shape of the vehicle doesn't change.

For localized damage, the model of the vehicle is divided into 26 separate hitzones. Otherwise the damage caused by the collision in the front of the vehicle would cause visual damage to the whole vehicle. Therefore the blending parameter is not a single value, but 26 independent values - one for each hitzone. The hitzones can be seen in Figure 11.8 together with their color codes. The color codes are used to divide the vertices of the model into the hitzones and they are given using the vertex colors of the model.

Similar division is also used for the collision shape of the vehicle. (i.e. the shape of the vehicle's rigid body). The rigid body of the vehicle is divided into the 26 smaller child shapes. Each of these shapes must be convex. The child shapes are added to one *CompoundShape* that is used as the vehicle's collision shape. The vertices in the model for collision shape use the same color codes as the model for graphical appearance (see Figure 11.8).

When the vehicle collides, the index of the colliding child shape is retrieved from Bullet. The index indicates the hitzone that is hit. The strength of the collision (i.e. the applied impulse) is used to reduce the health in the hitzone and the health is used to set the blending parameter for the hitzone. The management of the hitzones and the construction of the vehicle's collision shape is implemented in the *Hitzones* class.

The child shapes of the vehicle's collision shape are showed in Figure 11.9. Note that cylinder shapes are added for the wheels in the position when the suspension is fully compressed. This corresponds with the situation when the suspension is fully compressed and it acts as the rigid connection. The cylinder shape will collide with the track and the results is a rigid body collision between the vehicle and the track.

11.14 Vehicle's inertia

Bullet's standard method of setting rigid body's inertia tensor is by using the collision shape (the *CollisionShape* class). It is able to compute the inertia tensor using the *CalculateLocalInertia* method. However, in MotorDead, a different approach is used. The inertia is not computed from

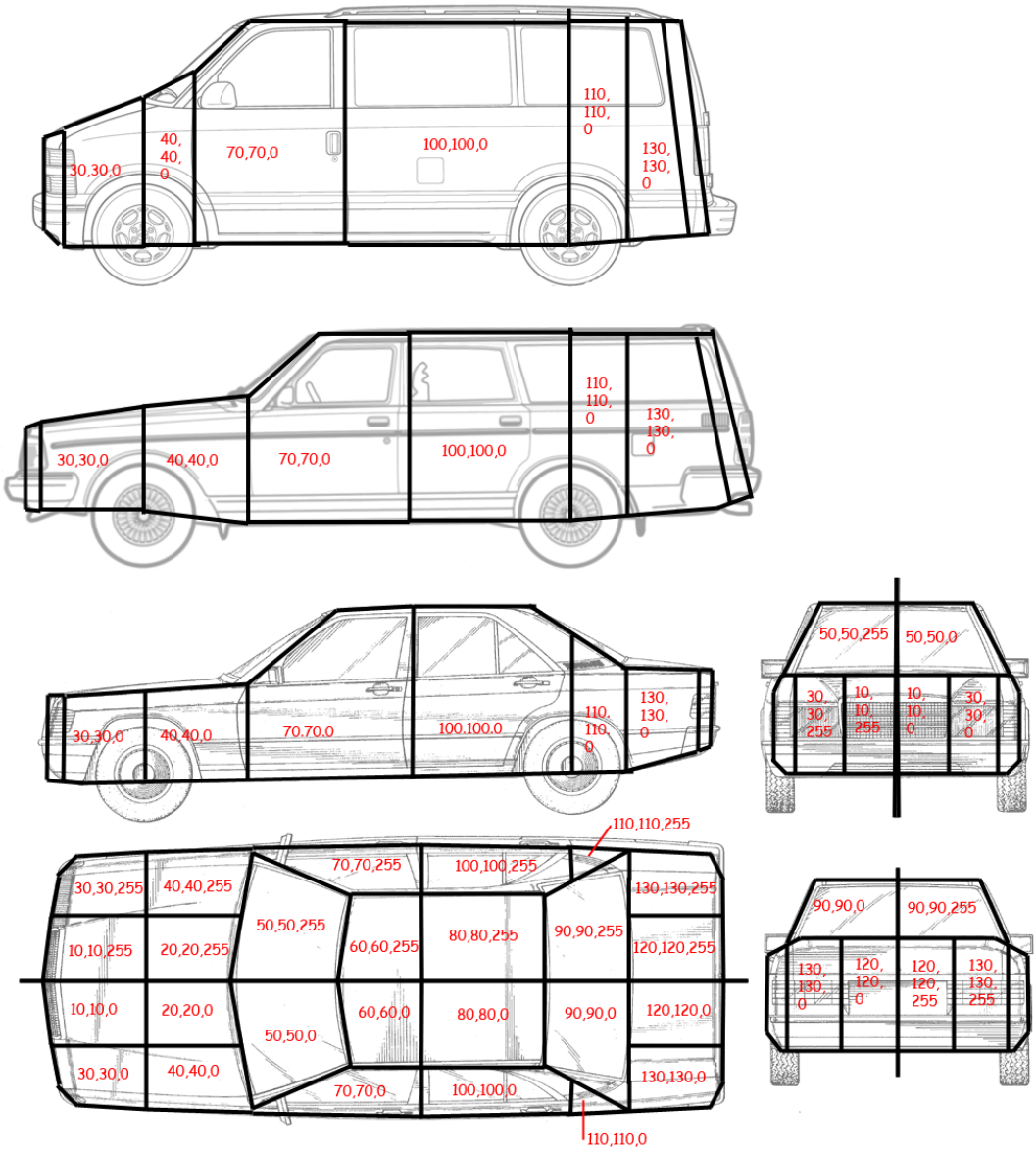

Figure 11.8: Hitzones of the vehicle and their color codes.

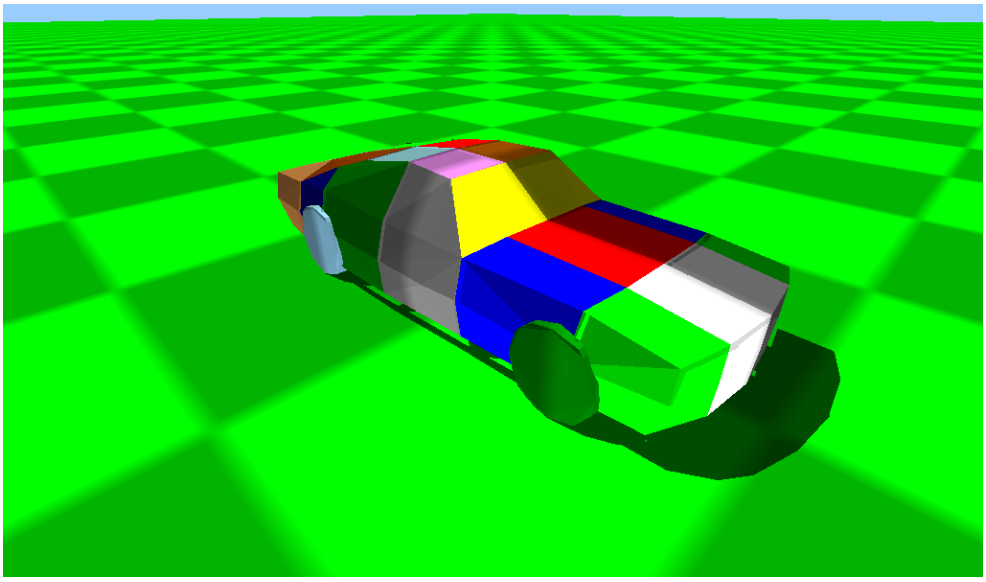

Figure 11.9: Collision shape of the vehicle.

the vehicle's collision shape but from the box shape (the box shape is depicted in Figure 11.10). The base of the box shape is given by the wheel positions and the height is given as an option in EDF file (see Section 11.16). This approach allows to have the collision shape and the inertia tensor of a vehicle independent (i.e. inertia tensor is not dictated by the shape). Therefore, it gives more control over the tuning of vehicle's handling and is expected to be more predictable.

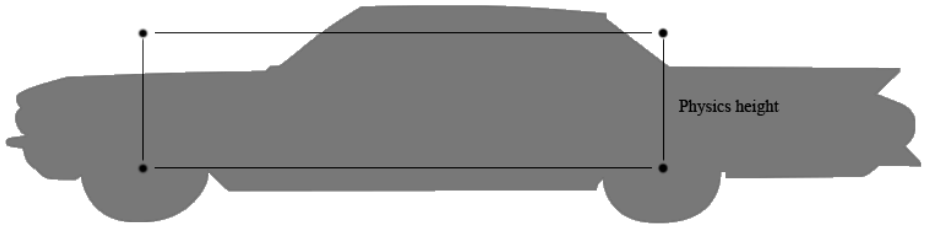

Figure 11.10: Collision shape used to compute the inertia tensor of the vehicle. The base of the box is given by the positions of the wheels and the height is an option in the vehicle's EDF file.

11.15 Vehicle related ContactAdded callbacks

An important part of the vehicle simulation is also the pair of *ContactAdded* event handlers (for more information about *ContactAdded* event, see Section 8.9). The callbacks are implemented

in the *PhysicsManager* class. Although the *PhysicsManager* is a more general purpose class, such a specific functionality can be located there. The the goal of MotorDead is not to create a general game engine and the vehicles are the center point of the game.

One callback (the *CustomVehicleFrictionCallback* method) changes the friction for the collisions involving vehicles. In reality the chassis of the vehicle is a soft body and it is deformed in the collisions. The deformations absorb part of the vehicle's kinetic energy. When using rigid bodies the kinetic energy can be reduced using friction. The method used to determine friction used in the collision is, more or less, based on a trial and error approach. The friction of the vehicle collision depends on:

- Vehicle's speed - the faster the vehicle goes the higher is the friction. A collision in higher velocity causes a greater deformation than a low speed collision. A greater deformation means that more of the kinetic energy is absorbed. Therefore the friction is set to a higher value for the collisions in higher speed.

- The angle between the normal at the contact point and the vehicle's velocity direction - more perpendicular contacts have lower friction (for example the vehicle is only sliding along the wall).

- Local position of the contact - contacts bellow the vehicle's center of mass have no friction. This gives better behaviour after jumps (the vehicle doesn't stop suddenly.)

A second callback deals with a collision when the colliding child shape is one of the cylinder shapes that represents the wheels (see Figure 11.9). In these situations, the contact normal is changed so it points in the same directions as the suspension's up direction (i.e. the opposite direction of the ray-casts). The collision impulse then acts in the suspension's direction and the result is closer to the real world behaviour of a suspension.

11.16 Vehicle settings

Vehicles are defined by the EDF files (see Chapter 5). The options from the file that defines a vehicle are summarized in Table 11.1. The description of the options from the point of view of a game designer can be found in Appendix E.

11.17 Vehicle upgrades

There are following types of vehicle upgrades in MotorDead:

- Tyres

- Brakes

- Suspension

- Armor
- Nitro
- Engine

For each upgrade type there are three stages (or levels) and they are common for all vehicles. This means that armor upgrade at stage 3 is the same upgrade for vehicle A as for vehicle B. For each type it is also possible to have no upgrade installed.

Most of the vehicle's physical properties are given in the EDF file of the vehicle (the settings are listed in Table 11.1). The values in the file are used as default and can be modified by the upgrades. If there is no upgrade that modifies certain value, this value is used directly as given in the EDF file. The upgrades are applied in a following way:

- tyres upgrade - contains multipliers for the forward and sideways friction coefficients. Default friction coefficients of the tyres are multiplied with the multipliers to get the values used in the simulation. Values of both multipliers should be greater than 1. However, worsening of the handling might be possible by applying values lower than 1. It is implemented by the *TyresUpgrade* class.
- Brakes upgrade - very similar to the tyres upgrade. Contains a multiplier for default power of the brakes. The value of the multiplier is also used as the value of the *BrakingFrictionMultiplier* property of the *Wheel* class (it raises the friction of tyres under braking). The brakes upgrade is implemented by the *BrakesUpgrade* class.
- Suspension upgrade - contains s multiplier for stiffening the suspension by multiplying the default damping and stiffness values. To improve stability there is also a multiplier that modifies vehicle's center of mass offset. The upgrade is implemented by the *SuspensionUpgrade* class.

Upgrades that doesn't modify the vehicle settings are:

- Armor - reduces damage of the vehicle. The damage from all collisions and weapon hits are reduced by the armor's coefficient which should be lower than 1. The upgrade is implemented by the *Armor* class.
- Nitro - gives temporary boost by multiplying the engine force that is exerted on the chassis. Specifically, the forward friction force from the wheels is multiplied when the nitro is activated and the throttle is pressed. Properties of the nitro are "Boost" - value of the force multiplier and "Amount" - time for which the nitro can be used (how long it takes to consume a full tank of the nitro).The nitro system is implemented by the *Nitro* class.
- Engine upgrade - upgrade of an engine is, in fact, a whole new engine installed in the vehicle. Therefore it doesn't modify the default value of the torque curve, but each engine contains its own torque curve. It is implemented by the *Engine* class.

The upgrades are also game entities and therefore they are defined by the EDF files. These files are located in “data\Entities\VehicleUpgrades” folder. The number of the stages of upgrades is limited to three. The following naming convention for identifiers of the upgrades is used: “<upgradeType><Stage>” (e.g. “Nitro3”, “Suspension2”).

For engines, the situation is a little bit different, since even when there is no upgrade installed, the vehicle still needs to have an engine to run. This situation is handled by default engine of stage 0 with name “Engine0”.

11.18 Vehicle builder

A vehicle that is used in the game (either in the race session or in the menus) is defined by the particular combination of the vehicle, upgrades, weapons etc. For representation of such a combination the *VehicleEntry* class is used. The *VehicleEntry* class is also used to serialize (and deserialize) the combination to the file. A vehicle entry contains following items:

- Vehicle identifier - identifier of the vehicle.
- Engine identifier - identifier of the engine that is installed in the vehicle.
- Gearbox identifier - identifier of the gearbox that installed in the vehicle. Currently there is only a single one gearbox used in the game.
- Brakes upgrade identifier - defines the installed brakes upgrade. The value may be *null* if there is no upgrade installed.
- Suspension upgrade identifier - defines the installed suspension upgrade. The value may be *null* if there is no upgrade installed.
- tyres upgrade identifier - defines the installed tyres upgrade. The value may be *null* if there is no upgrade installed.
- Armor identifier - defines the installed armor. The value may be *null* if there is no armor installed.
- Nitro identifier - defines the installed nitro system. The value may be *null* if there is no nitro installed.
- Offensive weapon identifier - defines the offensive weapon attached to the vehicle. The value may be *null* if there is no weapon attached.
- Defensive weapon identifier - defines the defensive weapon attached to the vehicle. The value may be *null* if there is no weapon attached.
- Health value - value of the vehicles health.
- Hitzones health values - values of the health in the vehicle’s hitzones.

Option name	Description
Model	Name of the model that is used for graphical appearance of the vehicle. See modelling conventions in Appendix F for more detail.
Collision model	Name of the model that is defines the collision shape of the vehicle.
Wheel model	Name of the model used for the vehicle's wheels.
Center of mass	Offset used to tune center of mass position (see Section 11.10 for details).
Mass	Mass of the vehicle
Physics height	Height of the vehicle's inertia shape (see Section 11.14 for details).
Suspension min length	Minimal length of the suspension (see Section 11.3).
Suspension rest length	Length of the suspension when no force is exerted by the spring (see Section 11.3).
Suspension max length	Maximal length of the suspension (see Section 11.3).
Stiffness	Stiffness of the suspension's spring (see Section 11.3).
Damping	Damping of the suspension's damper (see Section 11.3).
Forward friction	Coefficient of the tyres' friction in forward direction.
Side friction	Coefficient of the tyres' friction in sideways direction.
Rear side friction multiplier	Multiplier of the side friction coefficient used for rear tyres. Settings side friction of the rear tyres to higher value gives more stability in high speeds.
Wheel radius	Radius of the wheels.
Brake power	Power of the brakes.
Sliding power	Artificial steering torque that cause oversteering (see Section 11.9).
Drive	Controls which wheels transfer the torque.
Thumbnail	Name of the image displayed in the menus.

Table 11.1: Vehicle settings (present as options in the EDF file).

The *VehicleEntry* only defines a specific vehicle. The vehicle still must be created. The creating of the vehicles is done by the *VehicleBuilder* class. In the *Build* method it takes an instance of *VehicleEntry* class and builds the vehicle according to the entry. Building of the vehicle includes these steps:

1. Create the vehicle and the other parts (upgrades, weapons, etc.) of the vehicle according to the identifier in the *VehicleEntry* instance. The parts are created using the *EntityFactory* class (see Section 5.6).

2. Modify the vehicle settings according to the upgrades.

3. Assign the upgrades that do not modify the vehicle settings (armor, nitro and engine upgrade).

4. Attach weapons.

5. Set the value of vehicle's health and the values of health in the hitzones.

Chapter 12

Game logic

12.1 Environment

The *Environment* class represents the environment of the simulated world. The rigid body of the environment is used to represent the static geometry of the world.

When an environment is created it loads the scene dump given by the option "ModelFile" (in the EDF file defining the created environment). The geometry of the scene is used to create the collision shape of the environment. The *BvhTriangleMeshShape* class is used for the collision shape (it is created by the *CreateEnvironmentCollisionShape* method). Note that loading the scene dump also changes the background of the rendering scene (i.e. the scene contaning 3D models of the simulated objects, for more details, see Chapter 6). Therefore, there should at most one environment in the simulated world. The *Environment* class is used in the main menu background scene.

Important part of the environment is the set of applied rendering postprocesses. The postprocesses play significant role in the perception and the mood of the scene. The name of the file containing the set of postprocesses for the environment is given by the option "PostProcesses". The postprocesses are set to the renderer when the environment is created.

The *Environment* class has been designed as the lightweight base class of the *Track* class. It contains only the basic functionality needed for the environment whereas the *Track* add the features needed in the race sessions (mostly for the AI).

Note that there is a legacy code in the *Environment* class. Originally, the game entities should have been included to the *GameWorld* class with an environment (or a track) using the dummy points. The idea is that the scene dump of the environment contains the dummy points that define the included entities. The name of a dummy point is used to create game entity that is then placed at the position of the dummy point. The dummy points can be edited in Fibix Editor which is the main tool for preparing the game data. However, after the custom plugin to Fibix Editor was created, this system became obsolete. In the plugin the game entities can be added directly. For more details about the plugin, see Chapter 16.

12.2 Track

During a race session the environment of the simulated world is represented by the *Track* class. A track is essential part of any race session. Therefore there is track's identifier present in the *RaceSessionDefinition* class which defines a race session. The *GameWorld* class contains the *Track* property for accessing the track of the ongoing race session. Note that the *GameWorld* class is used during the race sessions (for more details, see Section 5.4.

The *Track* class is derived from *Environment* class. The *Track* class represents the static geometry of the simulated world in the same way as the *Environment* class. Therefore when a track is created it loads a scene dump that represents the graphics geometry and the set of postprocesses used for the track (note that by loading a scene dump with the track's geometry the background of the rendering scene changes too, see details in Chapter 6). The *Track* class extends the *Environment* class by adding the following features:

- AI data - data for AI navigation.
- Entities data - data for creating game entities and including them to the *GameWorld* class with the track.
- Explicit collision geometry - the collision geometry can be different (i.e. simpler) from the graphics geometry.
- Multiple entity material support - implementation of the *GetMaterialID* method that returns correct material according to the semantic IDs set in Fibix Editor.

12.2.1 Track's collision geometry

The collision geometry of a track is different than the graphical geometry. Because of performance reasons, the collision geometry should be much simpler than the graphical geometry. The scene dump that is used for the collision geometry is given by the "CollisionModel" option. Note that this scene is not supposed to be visible. It is not loaded to the rendered scene but to a newly created empty graphical scene. After the collision shaped is created, the scene used for loading is destroyed.

A loaded scene dump contains a list of render entities. A render entity is an object in the rendering scene (e.g. light, reflection source, object with a geometry, etc.). Only the rendering entities with a visible geometry are relevant for creating collision geometry. Such entities are called render objects (implemented by the *C_RenderObject* class). For more details about the render entities, see Chapter 6.

From the render objects of the scene dump the collision shape of the track is created. The approach is the same as for the *Environment* class and it is implemented in the *CreateEnvironmentCollisionShape* method. For each render object an instance of *IndexedMesh* class is created. The instances of the *IndexedMesh* class are then added to the *BvhTriangleMeshShape* class.

12.3 Race progress

We needed to be able to assign to every vehicle a value describing how far in the race it has progressed. If we compared the competing vehicles by these values, we would get the positions in which the vehicles are placed at any moment.

We decided to use the waypoints that serve for the AI's navigation and for checkpoints. We can estimate the progress of a vehicle for the whole race by assigning each waypoint a value of how far in the lap from the beginning it is. In code, this value is contained in the *DistanceFromStart* property of the *Waypoint* class. Then, $progress = numberOfCompletedLaps \cdot lapLength + closestWaypointDistanceFromStart$.

Since the waypoint graph may contain forks, computing the *DistanceFromStart* is done by breadth-first search of the graph. The goal is to compute the shortest distance from the starting waypoint to every other waypoint. For the implementation, see the *ComputeShortestPaths* method of the *Waypoints* class.

12.3.1 Multiple material support

In MotorDead the game entities have assigned entity materials that define the properties of the entities (see Section 5.7). A track represents a large static collision shape, therefore it can't have only a single entity material. For example an asphalt surface of the road has different properties than the grass area next to it.

The collision shape of the track is created from the render objects contained in the collision scene dump (see Section 12.2.1). One render object defines one part of the collision shape and it also defines the entity material for the part. The entity material is given by the semantic ID of the render object's rendering material. Therefore when a track is created it prepares a structure that stores a semantic ID value for each part of the collision shape.

It might seem as a simple array is enough to be used for storing the entity material IDs for the parts of the track's collision shape. However, the situation is more complex. A render object can have multiple materials assigned using a material group. The material group contains multiple materials together with the offsets telling on which triangles are the materials applied.

For example, let a render object be formed by 30 triangles and its material group contains two materials with offsets 0 and 20. The first material is used on the triangles #0 - #19 and the second material on the triangles #20 - #29.

Therefore the structure for storing the entity material IDs for the track contains a list of the semantic IDs and offsets for each part of the collision shape. The structure is implemented by the *_childMaterials* field and it is used in the *GetMaterialID* method.

12.4 Offensive weapons

The offensive weapons are entities that can be attached to a vehicle and are used to damage the opponents' vehicles. A typical offensive weapon, when fired, creates new items (i.e. game entities) and adds them to the simulated world (the same in which the weapon and the vehicle is).

The items are fired with a given velocity and cadence. Once an item is added to the simulated world it is independent and it can do any operation desired to damage the vehicles in the world. Although this description might not fit all imaginable offensive weapons, it is general enough for the most common ones.

The base class for the offensive weapons is the *OffensiveWeapon* class. It implements the firing mechanism. When a new item should be fired the *CreateFiredItem* method is called. The *CreateFiredItem* method is abstract and it is implemented in the subclasses that implement specific weapons. The subclasses can also use the *GetMuzzleWorldMatrix* and *GetMuzzleDirectionWC* methods to easily set the world transform and the velocity to the item created in the *CreateFiredItem* method.

The firing mechanism also includes counting time since the last item has been fired (for firing with a given cadence) and keeping the value of ammo. The time counting is done in the *PhysicsUpdate* method. The ammo value is managed by the *Ammo* class.

When an item is fired the additional effects are activated: a firing sound is played and a particle effect is created. The names of the sound and the particle effect are given as the parameters of the constructor of the *OffensiveWeapon* class.

12.4.1 Ray-cast projectile

The *RayCastedProjectile* class is the base class for representing projectiles. A projectile is an object that has given starting velocity, mass, gravitational force and range. A projectile moves according to these properties until it hits another object in the simulated world or it exceeds its range. In both cases the the life of projectile ends.

The *RayCastedProjectile* class is derived from the *PhysicsEntity* and it uses its rigid body to represent the projectiles. However, it is not possible to use the collisions of the rigid body for detecting the collisions of the projectiles. Projectiles are very small and they travel at a great velocity. The discrete collision detection can easily miss a collision between such object and another object with a thin collision shape (see Figure 12.1).

To address the issue of the missing collisions, the rigid body of the projectile is not used for detecting collisions. It is only used to simulate the movement of the projectile (the collision response for the body is turned off - the collision impulses are not applied). In each physics update the position of the rigid body is saved, so it can be used in the next physics update. The collisions are detected by ray-casting. A ray is cast from the last-update position to the current position of the rigid body. If the ray hits any object the abstract *Hit* method is called. The *Hit* method provides a way of implementing projectile's hit behaviour (e.g. explosion). In the hit implementation, the projectile must be removed from the simulated world.

The travelled distance of the projectile is updated in the physics update by adding the distance between the last-update position of the rigid body and its current position. Once the travelled distance exceeds the projectile's range the *RangeExceeded* method is called.

Note that the rays can be cast in the physics scene or in the rendering scene. This possibility has been added because some projectiles need to determine the precise position of the hit (see Section 12.4.2). They use the rendering scene. If the approximate hit position is satisfying the projectile should cast rays in the physics scene, which is faster.

Figure 12.1: Ray-cast projectile in two following update steps. The left picture shows the position of the projectile in first update step. The velocity of the projectile is illustrated by the black arrow. The right picture show the position in the second update step. Because of the high velocity the projectile is completely moved to the other side of the other collision object (the black line) and the collision is not detected.

12.4.2 Machine gun

The machine gun weapon is implemented by the *MachineGun* class. In the *CreateFiredMethod* it creates instance of the *Bullet* class. The *Bullet* class represents a bullet. In the *Hit* method, if the hit entity implements the *IDestroyable* interface bullet reduce the health of the hit entity. If the hit entity is a vehicle, the bullet raises the *WeaponHit* event. This event is used to claim bonus money for the hit (see Section 12.6).

The *Bullet* class uses ray-casting in the rendering scene (see Section 12.4.1). It is necessary for setting the precise positions of the additional effects that are activated when in the *Hit* method. With using the approximate hit positions (from the physics scene) the visual result of the effects is not satisfying.

One of the additional effect is a particle effect that visualizes the hit. The entity material of the hit entity is used to determine whether the particle effect should be created. Also the name of the particle effect is given by the entity material, should the effect be created. For more details about entity materials, see Section 5.7.

The second additional effect is the bullet hole effect. The bullet hole is represented by a decal. Decal is a type of render entity that is projected on the nearby geometry. The entity material of the hit entity is used to determine whether a bullet hole should be created. If it is so, the material also gives the name of the model that contains the decal. The model is instanced from the *ModelFactory* and the decal is added to the model of the hit entity using the *AddRenderEntity* class of the *Model* class.

Particularly, adding the bullet holes requires the hit position to be very precise. The decals does not contain geometry. Rather, they are projected on the nearby geometry. Moreover, in Fibix, there is no support for specifying which geometry is included in the projection. The decal is projected on all of the nearby geometry. This causes a certain issue, when the decal is added to the the track (i.e. static geometry). A model of a dynamic entity can get close to the decal and the decal is also projected on the geometry of the dynamic model, where it shouldn't be (see Figure 12.4.2). This issue is addressed by keeping the projection space (i.e. the AABB where the decal is projected) of the bullet hole decals tight.

12.4.3 Rocket and grenade launcher

The *RocketLauncher* and *GrenadeLauncher* classes fire instances of the *Rocket* and *Grenade* classes respectively. Both the *Rocket* and the *Grenade* classes are derived from the *Explosive* class. The *Explosive* class represents the projectiles that explode when they hit another object or they exceed the range.

The explosion is a physical entity itself. It is implemented by the *Explosion* class. When the *Explosiove* class explodes it creates a new instance of the *Explosion* class and adds it to simulated world. The *Explosion* class uses an instance of the *GhostObject* class. Ghost object is a special collision object that keeps the list of all collision objects (i.e. rigid bodies) it is currently colliding with. The ghost object used in the *Explosion* class to determine the rigid bodies that are in the range of the explosion.

In the *AddToPhysicsManager* method the *Explosion* class adds its ghost object into the physics scene. In the *PhysicsUpdate* method it iterates through the rigid bodies colliding with the ghost object. If the colliding rigid body represents entity that implements the *IDestroyable* interface the health of the entity is reduced (the entity is determined using the *UserObject* property of the *RigidBody* class).

If the colliding rigid body is not static the *Explosion* class applies an impulse on it. The direction of the impulse is given by direction from the position of the ghost object to the position of the rigid body. The point of application for the impulse is randomly generated. It makes the explosion more lively and gives the affected bodies also a rotational movement. Opposed to purely linear movement achieved by applying the impulse in the center of mass of the rigid body.

The magnitude of the impulse is proportional to the mass of the affected rigid body. A constant magnitude for all of the rigid bodies can't be used. It would be either too small and it would not move the heavier rigid bodies, or it would be too high and the light rigid bodies would be given an unrealistically high speed.

The *Explosive* class does not use the *Explosion* class directly. It uses the *WeaponExplosion* class derived from the *Explosion* class. *WeaponExplosion* is the inner class of the *Explosive* class. The reasons is that the *WeaponExplosion* implements the *IWeapon* interface and it can claim bonus money if the explosion hits a vehicle (see Section 12.6).

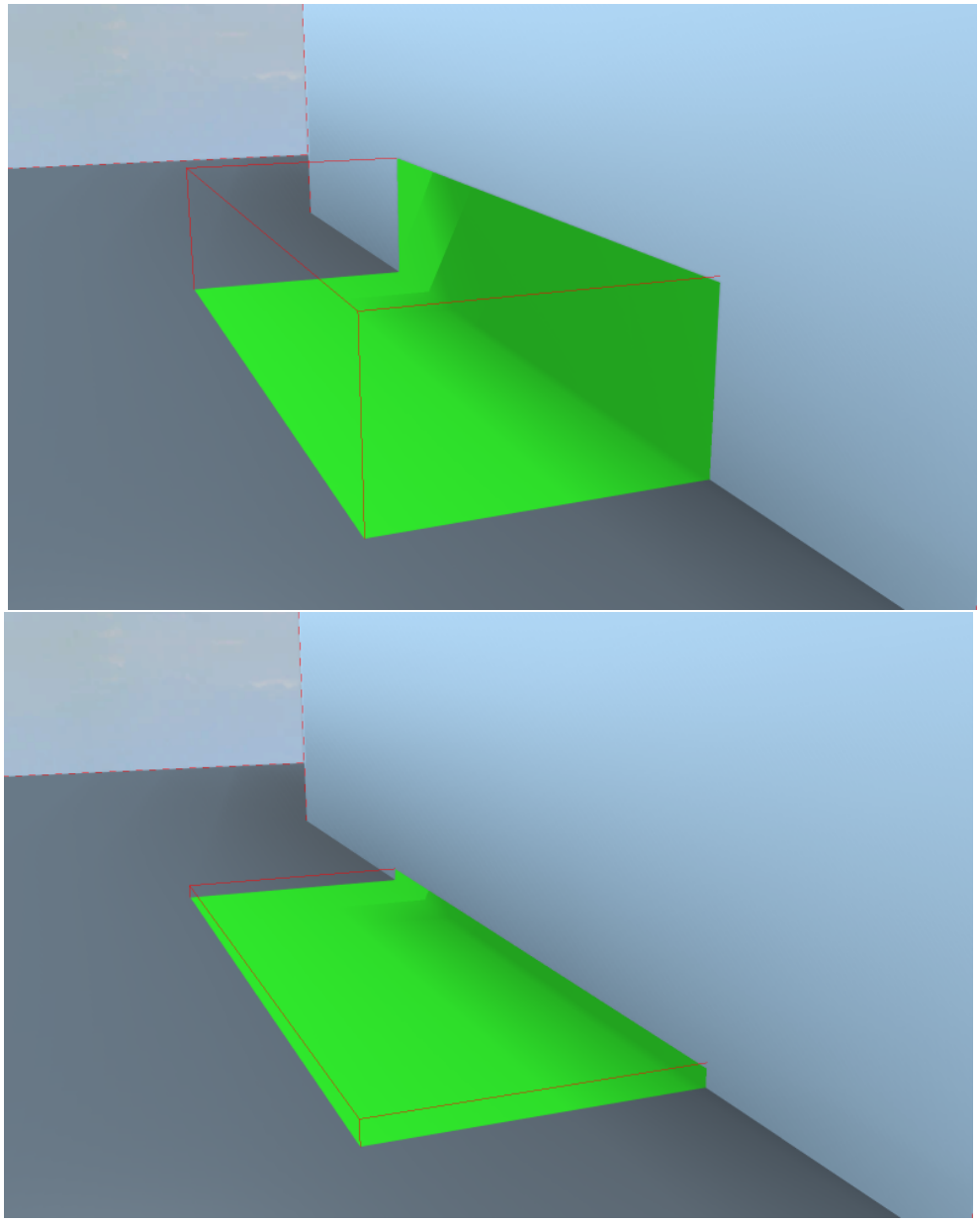

Figure 12.2: The top picture shows the decal (green) with big projection box. The box causes that the decal is projected also on the wall (light grey). The bottom picture shows the same objects but with smaller projection box of the decal. The projection of the decal on the wall is barely visible. The projection box is visualized using the red AABB of the decal.

12.5 Defensive weapons

The defensive weapons do not cause damage directly. They impair the handling of the hit vehicle or the driving conditions. As a result it is more likely that the driver of the vehicle will make a mistake and crash. Similarly to the offensive weapons, the defensive weapons are also attached to the vehicles and when fired they create new entities that are added to the simulated world.

12.5.1 Traps

The *Trap* class is the base class for the trap-like entities produced by the defensive weapons. The *Trap* class uses static rigid body with no collision response (i.e. the collision impulses are not applied). Such a rigid body works as a non moving sensor for detecting colliding entities.

When a vehicle collides with a trap, the trap starts affecting the vehicle (the *StartAffectingVehicle* method is called). The vehicle is added to an internal list of affected vehicles. In the *PhysicsUpdate* method the trap iterates through the affected vehicles and for each one of them it calls the *UpdateVehicleHandling* method. The *StartAffectingVehicle* and the *UpdateVehicleHandling* method are abstract and they must be implemented in the derived classes.

The *Trap* class also solves the problem arising when a vehicle collides with a trap and its already being affected by another trap of the same type. The *Trap* class has the static *_trapForVehicle* field. It is a dictionary, where the keys are the vehicles and the values are the traps that currently affect the vehicle.

When the vehicle collides with the trap and it is currently being affected by another trap, the trap which affects the vehicle is determined using the *_trapForVehicle* field and the *StartAffectingVehicle(Vehicle vehicle, T originalTrap)* method is called. The implementation of the *StartAffectingVehicle(Vehicle vehicle, T originalTrap)* method should remove the vehicle from *originalTrap* and only then start impairing the handling. If the colliding vehicle is not affected by any trap the *StartAffectingVehicle(Vehicle vehicle)* method is called.

The *Trap* class is generic to distinguish between the different types of traps, but still use the same implementation. The example of derived class is the *OilSpill* class (see Section 12.5.3).

The trap has a limited life time. The duration of the trap's life is given in constructor. When the trap dies, its model is removed from the rendering scene and it no longer starts affecting colliding vehicles. However, trap stays alive to finish affecting vehicle that are currently being influenced. The trap is removed from the simulated world when it finishes with the last vehicle.

12.5.2 Trap setter

The *TrapSetter* is a base class for defensive weapons that creates items derived from the *Trap* class (it is similar to the *OffensiveWeapon* class that is described in Section 12.4). The *TrapSetter* class implements the firing mechanism. When a new trap should be fired, the *Fire* method is called. In the *Fire* method a ray is cast to find the world transform for the trap (the trap should be lying on the ground). The ray is cast from the approximate position of the trap setter in downward direction. The approximate position is used to place the trap in the expected position - behind the vehicle, regardless of the mounting point where the weapon is attached to the vehicle. The ray

is colliding only with static collision shapes because it is expected that the trap will be placed on the non-moving object (it is mostly the entity represented by *Track* class.)

If the ray collides with a static geometry the world transform for the trap is computed. The position of the hit is used as the position of the trap. The normal at the hit point is used to compute the orientation of the trap. A random rotation around the hit point normal is added, which gives a very natural feel. Without the random rotation the traps are placed too uniformly. The rotation matrix is created using the *MathExtensions.GetNormalOrientedMatrix* method.

After the world transform for the trap is computed the abstract *CreateTrap* method is called (the world transform is passed as a parameter). The *CreateTrap* methods produces a new trap that is added to the simulated world.

12.5.3 Oil spills

Oil spills are traps that temporarily reduce the friction of the tyres of the influenced vehicle. An oil spill is implemented by the *OilSpill* class (derived from the *Trap* class). The friction is reduced by changing the sideways friction coefficients of the vehicle's wheels. The values of coefficients are interpolated between the zero at the start of the effect and the original values at the end of the influence. The original values are stored in the *_originalFrontTyresSideFriction* and *_originalRearTyresSideFriction* fields. Oil spills are created by the *OilSpiller* class.

12.5.4 Caltrops

Caltrops represent the trap that causes a puncture. However, in MotorDead there is no way of sustainable damage of the tyres. There the effect of caltrops is only that they slow down the vehicle that collides with the trap.

Caltrops trap is implemented by the *Spikes* class. It simply applies braking force to all affected vehicles until the speed of the vehicle is bellow certain threshold. Instance of the *Spikes* class are created by the *SpikesSetter* class.

12.5.5 Smoke screen

Smoke screen is different from the trap-like defensive weapons. It doesn't affect the handling of the vehicle but the driving conditions. The weapon is implemented by the *SmokeScreenCanister* class. It creates instance of the *SmokeScreen* class. One instance of the *SmokeScreen* class represents one smoke cloud.

The *SmokeScreen* class uses its rigid body to detect colliding vehicle. The rigid body has no collision response. When a vehicle enters the smoke cloud (the *CollisionStarted* method) the value of its *SmokeScreenThickness* property is increased by one. When a vehicle leaves the smoke cloud (the *CollisionEnded* method) the value is decreased by one. This approach automatically resolves situation when one vehicle is inside multiple clouds at once.

The value of the *SmokeScreenThickness* property is used by the AI that controls the vehicle (see Section 13.4.3). The driving conditions of the player are impaired only by the particle effect created by the *SmokeScreenCanister* class together with the instance of the *SmokeScreen* class.

12.6 Bonus money

During a race session, the bonus money is awarded to a driver who, using a weapon or with a vehicle collision, causes damage to the vehicle of another driver. The money is counted in the *GameWorld* class that uses an instance of the *RaceSessionCashCounter* class. The *GameWorld* class counts the money that is claimed by the game entities in the world.

The *Vehicle* class claims the bonus money for the collision via the *VehicleCollision* event. In the *GameWorld* class these events are handled by the *VehicleCollisionHandler* method, which adds the money to the driver of the vehicle that has raised the event.

Bonus money for weapon hits are claimed by all entities that implement the *IWeaponHit* interface (e.g. bullets, rockets, oil spill, etc.). Whenever an entity implementing the *IWeaponHit* interface is added to the *GameWorld* class, the *GameWorld* registers a handler on the *WeaponHit* event of the entity. The events from the *IWeaponHit* interface are handled by the *WeaponHitHandler* method. The *WeaponHitHandler* adds the money to the driver that had fired the weapon and it raises the *WeaponHit* event of the *GameWorld* class. This event is used by the AI system (see Section 13.5.3).

The overall bonus money is part of the race result. The bonus money of the player is added to the cash stored in the profile of the player. Bonus money might become more important in the future, if the career mode is implemented.

12.7 Debris

Debris in MotorDead are the dynamic entities that collide with other static and dynamic objects. Debris are passive in a way that no forces are exerted on them, except for the gravity and the impulses from collisions. Typical examples of debris are various boxes and barrels that are placed along the track. Having such objects in the game adds a lot of "life".

One piece of debris is implemented by the class *Debris*. It is defined by a model that is used in the rendering scene, a collision model that is used in the physics simulation and the physical properties of mass and friction (they are used to define the rigid body of the debris). The *Debris* class also allows setting of the rigid body from outside of the class using the *SetRigidBody* method. The *SetRigidBody* method is used by the environment triggers (see Section 12.9).

Note that the debris are placed on a track using the FightRace plugin (for more details about the plugin, see Chapter 16). In the game the debris are created from the data serialized in the plugin. The debris data are part of the definition of the track.

12.8 Static collision objects

Static collision object represent an invisible and static rigid body added to simulation world (i.e invisible barriers). It is defined only by the collision shape. The static collision object is implemented by the *StaticCollisionEntity* class.

The static collision objects are also used to prevent the vehicle from getting to unwanted places on the track (such as the water in "The docks" track). A static collision object may be set as a "resetting object". The *ResetsVehicle* property of the *StaticCollisionEntity* class is used to turn on the resetting of the vehicle. When the value of the *ResetsVehicle* property is true, each vehicle the collides with the object is reset using the *ResetVehiclePosition* method of the *GameWorld* class (see the *CollisionStarted* method of the*StaticCollisionEntity* class).

12.9 Triggers

Trigger is a physical entity that acts as a sensor that is triggered by a collision with a vehicle. The trigger has two states - active and inactive. When a vehicle collides with the trigger in the active state, the trigger does a specific action (the action depends on the type of the trigger) and it switches itself into the inactive state. When a vehicle collides with the trigger in the inactive states, the collision is ignored. The trigger has a given time representing how long it stays in the inactive state. After this time elapses, the trigger automatically switches from the inactive into the active state.

The *VehicleTrigger* class is the base class for all triggers. It implements the collision detection and the state switching. The *VehicleTrigger* class uses a rigid body with no collision response for detecting collisions with vehicle. When a vehicle collides with the trigger in the active state the *VehicleHit* method is called. It is overridden in the child classes and it implements the specific trigger action. The *VehicleTrigger* class also plays a sound when a vehicle collides with a trigger.

Trigger type	**State**	**Entity name**
Ammo trigger	Active	AmmoTriggerActive
Ammo trigger	Inactive	AmmoTriggerNonactive
Repair trigger	Active	NitroRepairTriggerActive
Repair trigger	Inactive	NitroRepairTriggerNonactive
Speed-up trigger	Active	SpeedUpTriggerActive
Speed-up trigger	Inactive	SpeedUpTriggerNonactive
Environment trigger	Active	EnviroTriggerActive
Environment trigger	Inactive	EnviroTriggerNonactive
All triggers	Both states	InvisibleTrigger

Table 12.1: List of the trigger materials. To get a certain material from the scene, an entity with given name is found (in the scene) and the material of this entity is used. Note that invisible material is common for all trigger types and for both states.

The implementation of switching between the active and inactive state includes changing the appearance of the the trigger. The trigger is visualized using a decal, that uses two materials. One for the active state and the other one for the inactive state. The materials are kept in the protected *_activeMaterial* and *_activeMaterial* fields and can be set in the derived classes.

The trigger can also be invisible. This option is switched by the *Invisible* property of the *VehicleTrigger* class. When the *Invisible* property is set to *true* the material used for the both active and inactive state is invisible and the same one.

The materials for the trigger are managed by the static class *TriggerMaterialManager*. It is initialized from the *FightRaceGame* class. In the initialization the *TriggerMaterialManager* class loads a scene dump that contains all the materials for triggers. The *TriggerMaterialManager* class holds the materials from the scene according to the names of render entity that has the material assigned (i.e. to get a certain material, an entity with given name is found in the scene and material of this entity is used). The names of the are listed in Table 12.1.

There are four types of triggers in MotorDead:

Speed up trigger
: Boost the speed of the colliding vehicle. It is implemented by the *SpeedUpTrigger* class. Note that the speed up trigger is always active. This behaviour is implemented by setting the duration of the inactive state to zero.

Ammo trigger
: Recharges ammo of the weapons of the colliding vehicle. It is implemented by the *AmmoTrigger* class.

Repair trigger
: Fully repairs the colliding vehicle. It is implemented by the *RepairTrigger* class.

Environment trigger
: Environment trigger references a debris object. When the environment trigger is triggered it changes the rigid body of the debris object from static do dynamic. By doing so, the debris object starts moving. This trigger is intended for various surprise traps in the tracks. The environment trigger is activate only once, then it's stays in the inactive state.

  The environment trigger is implemented by the *EnvironmentTrigger* class. The instance of *Debris* class can be assigned to the trigger using the *LinkedDebris* property. In the setter of the *LinkedDebris* property the rigid body of the given debris object is stored and a new static one with the same collision shape is created. The static body is then assigned to the debris object with the *SetRigidBody* method of the *Debris* class. Since the rigid body isn't added to any *DynamicsWorld*, the *LinkedDebris* property must be used before the trigger and the debris are added to the *SimulationWorld* or the *GameWorld* class.

  When the environment trigger is triggered during the race, it changes the body of the linked debris object from the static body back to the original (dynamic) one. At this point, however, both the trigger and the debris are added in the *SimulationWorld*. Therefore the debris is removed from the *PhysicsManager* of the *SimulationWorld* before the body is changed and then is added back.

12.10 Skid marks

A skid mark is the mark a tyre makes when a vehicle wheel stops rolling and slides or spins on the surface of the road. Skid marks are a common part of motorsport and naturally they are present in all racing games. Skid marks are also implemented in MotorDead. The whole system consists from three parts:

- The *SkidMarksManager* class - central object that manages the skid marks visible in the scene. The skid marks have to added to the *SkidMarksManager* to be visible.
- The *Wheel* class - produces the skid marks according to the current condition of the wheel (i.e. spinning, sliding, handbrake, etc.).
- The *SkidMarksCreator* class - provides the connection between the *Wheel* class and the *SkidMarksManager* class.

12.10.1 Skid marks manager

The *SkidMarksManager* class is a central manager of the skid marks that are created during a race session. The *GameWorld* class holds an instance of the *SkidMarksManager* class. It is accessible through the *SkidMarksManager* property of the *GameWorld* class.

The skid marks (of one material) are represented by a pair of render objects. The geometry of these render objects is created dynamically in the game. When a skid mark is created, the geometry of the objects is changed.

The skid marks are not added at once. A wheel slide that produces the skid marks may last for multiple consecutive physics updates (the *Wheel* class produces skid marks in the physics update, see Section 12.10.4). Therefore, the skid marks are created and added to the *SkidMarksManager* class by parts. Each time a new part of skid mark is added, the *SkidMarksManager* class adds a new **quad** to the render objects that represents the skid marks in the rendering scene.

The render objects used to represent the skid marks are prepared in advance, in the constructor of the *SkidMarksManager* class. They use pre-allocated vertex and index buffer. Note that the index buffer can be filled right away in the initialization. The parts of the skid marks are always added as quads - the topology of the object representing skid marks is constant and known in advance. The vertex buffer is filled with the added parts of the skid marks.

The render objects representing the skid marks are not used directly in the *SkidMarksManager* class, but through the *SkidMarkSingleMaterialObject* class. The *SkidMarkSingleMaterialObject* class holds one render object and manages its vertex and index buffer. A new part of the skid mark can be added using the *AddSkidMark* method of the *SkidMarkSingleMaterialObject* class. The *SkidMarkSingleMaterialObject* class also contains the *Lock* and *Unlock* methods for locking and unlocking of the vertex buffer.

The size of the vertex buffer in the *SkidMarkSingleMaterialObject* class is limited. The size is set when the vertex buffer is created and it stays constant. It follows that the buffer might get full. Therefore there are two instances of *SkidMarkSingleMaterialObject* class for each of the skid marks materials. When the first object gets full, the second one is reset and new skid marks

are written to the vertex buffer of the second object. When the second object gets full, the first one is reset and filled from that moment. Reset of the *SkidMarkSingleMaterialObject* class is done by its *ResetLock* method. The *ResetLock* methods locks the vertex buffer in a way that the GPU discards the old buffer and creates new one. The *ResetLock* method also sets the number of the triangles, for which the material is applied, to zero.

The approach of using two render objects in turns (and resetting them in each turn) has a disadvantage. The skid marks represented by the object disappear when the objects is reset. It can be improved with using more objects, which would make the disappearing less apparent. The best solution would be to fade the object away with increasing transparency before the reset. However, this is not implemented in MotorDead. The two render object shares the same material. Therefore setting the transparency changes both of them. This would be solved with the use of different (although visually same) materials. Unfortunately, Fibix currently doesn't support the duplication of materials at engine level.

The pair of the *SkidMarkSingleMaterialObject* instances is managed by the *SkidMarksSingleMaterialManager* class. Note that both the *SkidMarkSingleMaterialObject* and the *SkidMarksSingleMaterialManager* classes are inner classes (also called nested classes) in the *SkidMarksManager* class. They are used for better encapsulation of the data.

In real world skid marks look different on the asphalt then on the gravel. In MotorDead there is support for multiple materials of the skid marks. The *SkidMarksManager* class contains one instance of the *SkidMarksSingleMaterialManager* class for each entity material on which skid marks can be created (for more details about game entity materials, see Section 5.7). The instances of the *SkidMarksSingleMaterialManager* class are prepared in the constructor of the *SkidMarksManager* manager class. The instances of the *SkidMarksSingleMaterialManager* class are showed in Figure 12.3.

The creating of the skid marks uses information from the physics simulation. Therefore, the skid marks are created in *Wheel* class during the physics update. To bypass any complicated buffering system the *SkidMarksManager* class is also updated with the physics simulation. To do so it implements the *IUpdatable* interface and it is added as substep updatable to the *PhysicsManager* class (for more details about the substep updatables, see Section 8.8).

12.10.2 Adding new skid marks

Adding of the new parts of the skid marks (new quads) to the render objects is not a straightforward operation. Firstly, the vertex buffer must be locked in order to write the vertex data to it. Secondly, the lock operation costs some time. Therefore the *SkidMarksManager* class must keep the number of the lock operations down.

In the *Update* method of the *SkidMarksManager* class all vertex buffers are locked. Then the vertex data are written for all newly created parts of the skid marks and afterwards the vertex buffers are unlocked. This approach may lock and unlock the buffers even when no vertices are written. On the other hand, each vertex buffer is locked only once, even if several quads are added to the buffer. As a result the time spent locking the buffers is independent of the number of added skid marks - the performance of the game is not impaired in the situations when several vehicles produce the skid marks at once.

Figure 12.3: The *SkidMarksManager* class that contains three instances of the *SkidMarksSingleMaterialManager* class - one instance for each skid marks material. Note the pair of the *SkidMarksSingleMaterialObject* instances for each material.

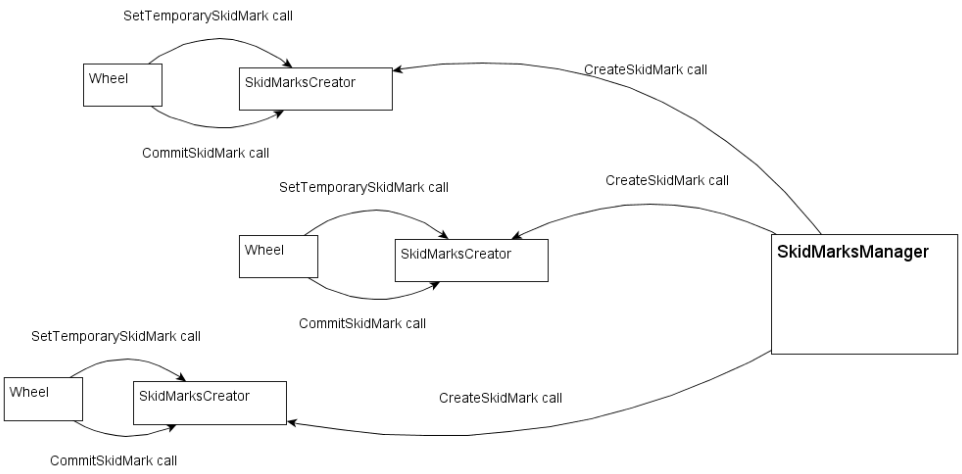

Figure 12.4: Skid marks system. The *Wheel* class uses the *SkidMarksCreator* class to add new skid marks. The *SkidMarksManager* class queries the instances of the *SkidMarksManager* class for newly added skid marks.

The *SkidMarksCreator* class (for details, see Section 12.10.3) is used for adding new skid marks produced by the *Wheel* class to the *SkidMarksManager* class. An instance of the *SkidMarksCreator* class can be added to the *SkidMarksManager* class using the *AddCreator* method. A creator can be removed using the *RemoveCreator* method.

The *SkidMarksCreator* class keeps a list of creators. In the *Update* method of the *SkidMarksManager* class, after the buffers are locked, the creators are queried for newly added skid marks. The *SkidMarksCreator* class is queried using the *CreateSkidMark* method.

12.10.3 Skid marks creator

A skid marks creator is used as the connection between the *Wheel* class and the *SkidMarksManager* class. Originally, the *Wheel* class added skid marks directly to the *SkidMarksManager*. However, this approach caused certain issue.

The issue is caused by the fact, that the last skid mark quad that is added to the scene becomes visible in the following frame, when the vehicle is already moved to a new position. As a result, there is always a gap between the wheel and the skid mark. Such a gap is unacceptable. Therefore the skid marks are created at the position of the wheel that is predicted for the next physics update. The prediction is made using the velocity of the wheel and the time step of the update. Using the time step of the physics update is not correct (the frame update time step might be different from the physics update time step - see Section 5.1). However, using the predicted position is an approximation in the first place. Therefore, using the time step duration of the physics update is not a big issue.

Placing the skid marks at the predicted future position solves the problem with a gap between the wheel and the skid mark left by the wheel. Unfortunately, it also brings new issues. When the

wheel produces skid marks and it approaches a sudden drop on the road (e.g. a jump). The skid mark is placed at the predicted position. If the drop in the geometry is too close to the current position of the wheel, part of the skid mark might end up in the air (see Figure 12.5).

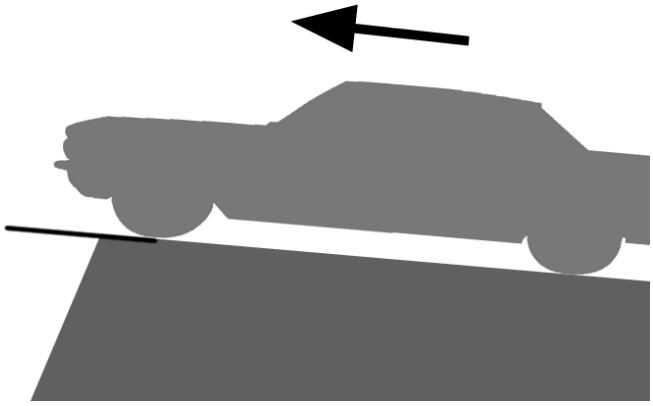

Figure 12.5: Skid mark is created at the predicted position. If there is a sudden drop in the geometry, the skid mark ends up in the air.

Second issue is the case when the skid marks intersect with the geometry of the track. It can be illustrated with the example of a vehicle landing after a jump in Figure 12.6. The skid marks are created at the predicted positions and as a result they might intersect with the geometry of the track. The issue is that the offset between the skid marks and the geometry of the track might be very small and it causes a Z-fight.

The both issues are tackled with the *SkidMarksCreator* class. The *Wheel* class uses an instance of the *SkidMarksCreator* class. The *SkidMarksCreator* class contains a small render object that displays only one quad. This quad is used to display the most recently added part of the skid mark. The most recent part of the skid mark displayed by the *SkidMarksCreator* class is set by the *SetTemporarySkidMark* method.

The quad is only temporary. If the *Wheel* class confirms it by calling the *CommitSkidMark* method with first parameter set to *true*, the quad is passed to the *SkidMarksManager* class and it becomes permanent. However, the *Wheel* class can also discard the temporary quad by calling the *CommitSkidMark* method with first parameter set to *false*. This option is used in situations that are showed in Figure 12.5. The *Wheels* class calls the *CommitSkidMark* method in the following update after it has called the *SetTemporarySkidMark* method. The temporary skid mark is commited only when the wheel has contact wich the surface. This prevents from creating the skid marks in the air.

The parameters of the *CommitSkidMark* are also used to correct the position of the confirmed (i.e. committed) part of the skid mark. Correcting the position is necessary to prevent a Z-fight between the skid marks and the geometry of the track (see Figure 12.6). The position is corrected

using the current hit position of the wheel's ray-cast (for more information about ray-casting in the *Wheel* class, see Chapter 11).

The *SkidMarksCreator* class also prepares the raw vertex data from the parameters given by the *Wheel* class. This includes connecting the neighbouring quads - with both the positions of vertices and the values of alpha in the vertices.

It is apparent that the above mentioned issues are particular results of the fact, that the physics is simulated in the discrete steps and the contact between the tyre and the surface is not continuous (as it is in reality). However, the general solution is out of the scope of the game.

12.10.4 Wheel - producing skid marks

The *Wheel* class is responsible for creating skid marks. In the strict physics simulation it is straightforward to determine when the skid mark should be created. The skid marks are created when the friction force is not high enough to counteract the wheel's speed and the wheel slides or spins. However, the appearance of the skid marks changes with the condition of the wheel. For example when the vehicle is in a controlled slide in the turn, the skid marks are thin and less visible. On the other hand, when the vehicle brakes with locked wheel, it leaves wide and dark marks on the road.

Controlling the attributes of the skid marks (the width of the skid mark and the alpha of the skid marks' material) from the physical simulation is a tough task. Therefore, the method that produces the skid marks used in the *Wheel* class is ad-hoc and tuned to satisfying results.

The skid marks are produced only when the speed of the wheel compared to the surface is greater than the threshold given by the *SKID_MARK_MIN_SPEED_THRESHOLD* constant (currently set to $5m \cdot s^{-1}$). This limit prevents from producing a large amount of skid marks at low speeds. The skid marks are produced from three different reasons:

- The wheel is locked (i.e. handbrake) - the value of the *Locked* property must be set to *true*. In this case the value of alpha is set to one (the skid marks are as apparent as the material allows).

- The wheel is spinning - the friction force in the forward direction is too small to counteract the rotational speed cause by the torque applied to the wheel. The value of alpha is set using the formula $\alpha = 2 * spinningPower$. The $spinningPower$ term is value used to approximate the spinning. It is computed from the values of the friction force that would be needed to counteract the speed in forward direction and the value of the maximal friction force from the Coulomb friction (see Section 11.4). The value of $spinningPower$ is computed using Equation 12.1.

$$spinningPower = \frac{needed\ friction\ force}{maximal\ friction\ force} - 1 \tag{12.1}$$

  The skid marks from spinning are created only when the speed of the wheel is lower than the value of the *SKID_MARK_SPINNING_MAX_SPEED_THRESHOLD* constant. This limit is completely ad-hoc, but it improves the overall visual appearance of the skid marks.

It prevents the wheel from producing skid marks whenever a torque is applied. In reality a vehicle with powerful engine may produce skid marks almost regardless of its velocity. However, with a throttle pedal it is easier to control the torque applied to the wheels, than it is with a keyboard.

- The wheel is sliding - this case is used when the value of $sidewaysSpeedRatio$ is greater than the threshold given by the *SKID_MARK_SIDEWAYS_RATIO_THRESHOLD*. The value of $sidewaysSpeedRatio$ is computed using Equation 12.2. The meaning of this case is "the wheel is going too much sideways".

$$sidewaysSpeedRatio = \frac{sideways\ component\ of\ the\ wheel's\ velocity}{magnitude\ of\ the\ wheel's\ velocity} \tag{12.2}$$

In all three case the width of the skid marks is controlled by the value of $sidewaysSpeedRatio$, which is given in Equation 12.2. The width is computed by the *GetSkidMarksWidth* method. The material of the skid marks is determined from the game entity material given by the surface under the wheel (i.e. the entity hit by the wheel's ray-cast, for more information about ray-casting in the *Wheel* class, see Chapter 11).

12.11 Vehicle - gameplay

This section describes the vehicles from the point of game logic. The physics principles of the vehicles simulation are described in Chapter 11.

There are two types of sounds in MotorDead: looping (designed to be played in a never ending loop) and non-looping (played once). More details about the sounds in MotorDead can be found in Chapter 15. The sound of the vehicle is implemented in *VehicleSound* class. The *VehicleSound* class plays both the non-lopping sounds and the looping sounds of the vehicle. The non-looping sounds are used for event action (e.g. changing gears, nitro sound). The looping sounds are used for continuous sounds produced by the vehicles (e.g. engine sound, rolling sound from the wheel, etc).

The vehicle creates particles. The particle produced by the vehicle include:

- Exhaust fumes - a particle effect is attached to all exhaust positions given by the dummy points of the vehicles models (see modelling conventions in Appendix F).

- Damage particles - a particle effect is attached to all positions given by the dummy points of the vehicles models (see modelling conventions in Appendix F). There are two different damage particle effects. The light damage particle effects are activated when the health of the vehicle falls below the *LIGHT_DAMAGE_HEALTH_THRESHOLD* threshold. The heavy damage particle effect are activated (and the light damage particle effects are de-activated) when the health of the vehicle falls below the *HEAVY_DAMAGE_HEALTH_-THRESHOLD*.

- Continuous collision particles - particle effect activated when vehicle is scraping along other object. Such effect is implemented in the *VehicleParticleEffectManager* class. The *Vehicle* class reports collisions and updates the *VehicleParticleEffectManager* by calling the *Update* and *Collision* method respectively. Note that the (non-continuous) collision particle effects are created by the *PhysicsManager* class (details can be found in Section 8.7.1.

- Wheel particle - the particles produced by the contact between the wheel and the surface under it. There are two types of the wheel particles: rolling wheel particles (produced by the wheel rolling on the surface) and sliding wheel particles (produced when the wheel slides or spins on the surface). The fact whether the particles are created and, if so, the name of the particle effect that is used depends on the entity material under the wheel (see description of the entity materials in Section 5.7). The *Wheel* class uses an instance of the *WheelParticleEffects* class to produce particles. Note that the wheel sliding particles are triggered using the value *CreatesSkidMarks* property of the *Wheel* class. This approach ensures that the skid marks and the sliding particles are produced at the same time.

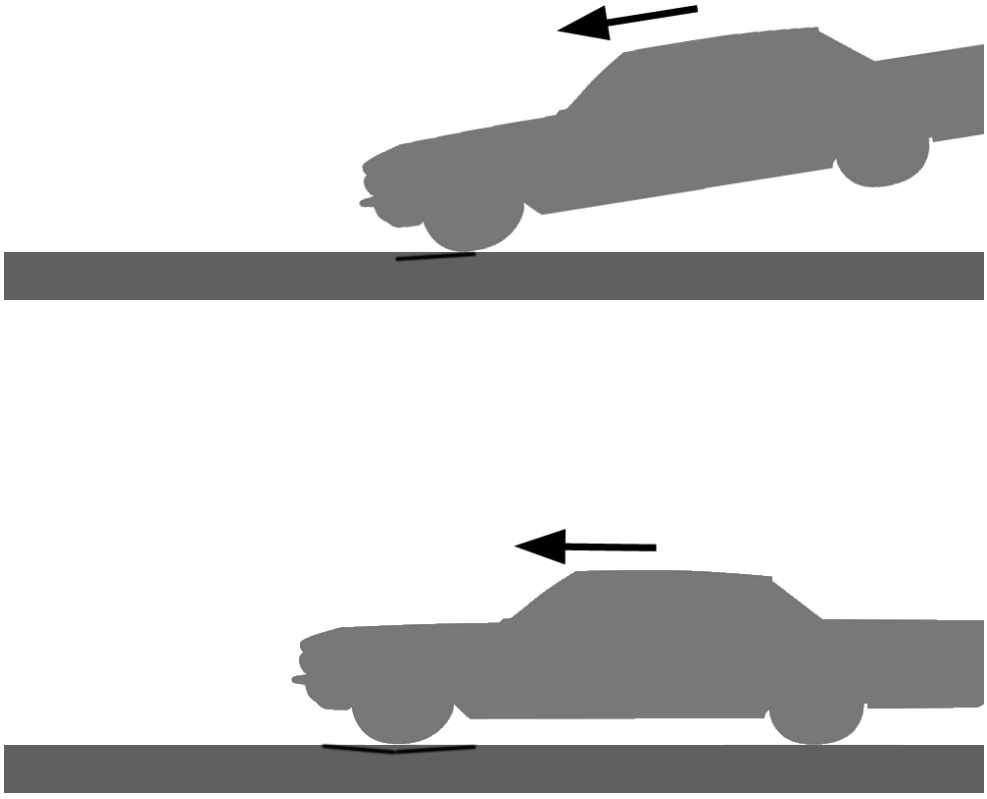

Figure 12.6: Two consecutive updates of a vehicle landing after a jump. Skid marks created in both updates using the predicted position of the wheel. Notice the small offset between the skid mark and the geometry of the track, which causes a Z-fight. The position is therefore corrected in the commit operation to prevent the Z-fight.

Chapter 13

Artificial Intelligence

13.1 Introduction

MotorDead is a single player racing game with the possibility to have up to 6 opponents on the track. That means we have to implement some kind of artificial intelligence (AI) to control other vehicles. We decided to do so without any external library or framework. At the beginning we created an independent project called *AISandBox*, which was used as a proof for our AI concepts. For simplicity we used a projection of 3D world to 2D. We can see this projection in Figure 13.1. After successful tests on a small part of the Masaryk circuit, we started to implement our concepts in MotorDead project directly. In this chapter, you will be introduced to the concepts we used for AI.

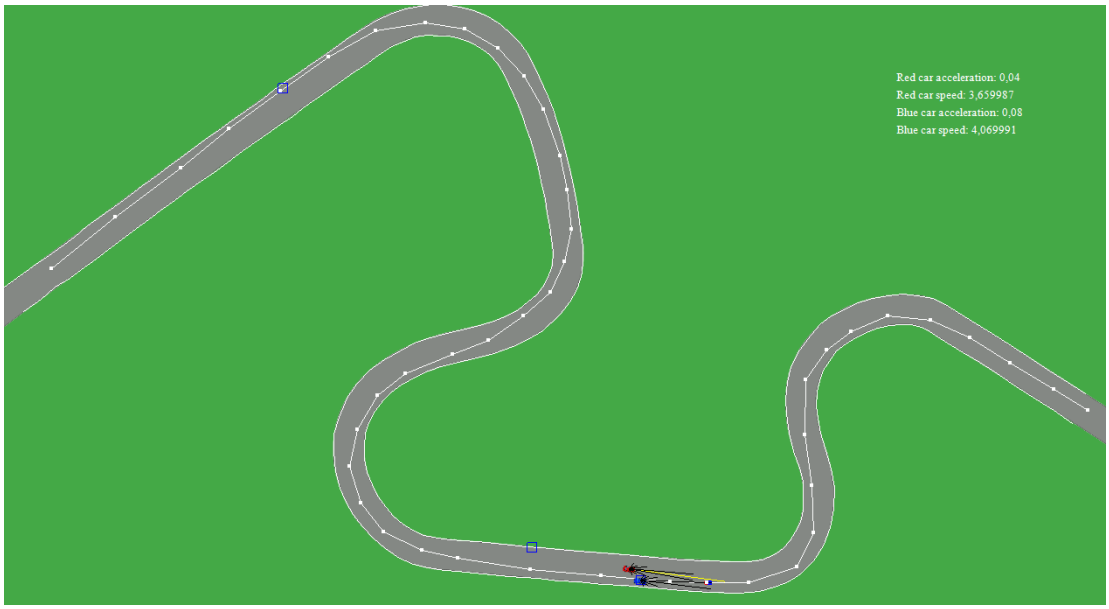

Figure 13.1: Game world 2D projection in AISandBox project.

13.2 AI environment

Game world as we see it in the race is too complex for AI purposes so we had to create another representation of the world which is much simpler but still has enough information for controlling the vehicle. For this representation we used a pair of special entities — track borders and waypoints. The world representation with these entities we can see in Figure 13.2. Their creation is handled in the Fibix editor and its described in Section E.3.1.

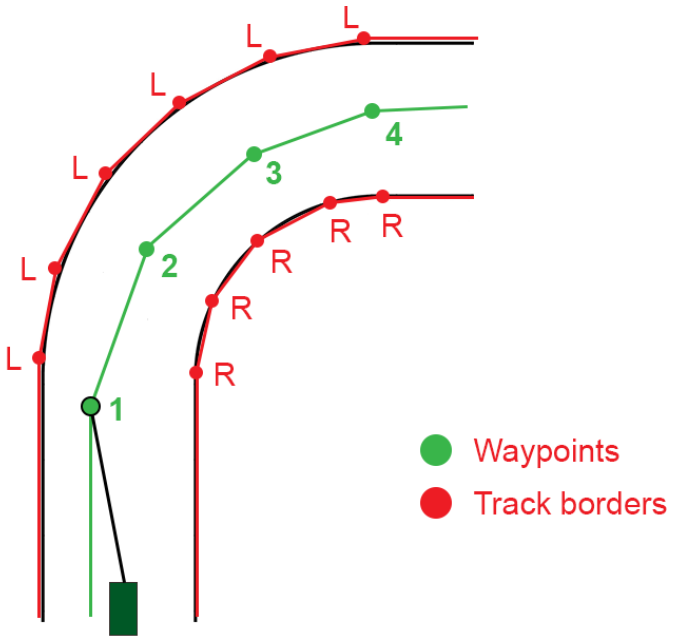

Figure 13.2: Definition of game world for AI.

Track borders

Track borders are entities created as a definition of the race track for AI drivers. Because of this representation an AI driver can easily determine if his vehicle is on the track or not. The whole track border is created from points implemented as class *TrackBorder* which are connected with the line. Each instance of class *TrackBorder* has exactly one previous and one next track border point to which it is connected. This object also holds information about the track side — left or right. This information is important for correct calculation of collisions between the vehicle and the track borders. The advantage of this design is that we can use the same set of track borders for both the normal and the reversed track. The only task we have to do during the loading is to swap the track border sides.

Waypoints

Controlling with track borders only is possible, but for a better control of the vehicle, especially on the straight track, we created the object waypoints which help us to control

the vehicle more precisely. It is implemented in the *Waypoint* class in the *AI* namespace. Every waypoint must have at least one next waypoint to which it is connected. If the waypoint has more than one next waypoints it creates so-called *fork*. The fork creates an alternative routes for the AI player. To create a valid list of waypoints, each alternative route has to join the main route. Connected waypoints form a circle which represents one lap of the track. Individual connections between waypoints are used as a guide lines which the vehicle should follow. More about how the vehicle follows these guide lines is in Section 13.4.1. Waypoints have two specific properties that hold information about track:

- Jump - determines if waypoint lies near any place where a vehicle can jump.
- Start - determines the first waypoint of the track.

Beside track borders in case of reverse mode we have to define another set of waypoints. This is because the existing reversed path is not always driveable. For example, if we jump from second floor of the building in normal direction is not possible to get from ground to second floor in reverse mode.

13.3 Vehicle control

Controlling the vehicle is the key feature needed for implementation of the proper AI driver. In MotorDead we are trying to simulate player's input because of we are limited to only a few actions that could be performed (for example accelerate, brake, steer right, fire front weapon, etc.). All these actions are handled by the *AIController* class, which represents the core element of AI. Each AI driver has one instance of this class which holds all necessary information needed for his vehicle to be controlled. Control actions are generated each *AIController's* update which consists of many steps. These steps are depicted in Figure 13.3.

In the game we could have more than one AI opponent. For this reason we created *AIManager* class for managing all AI drivers. For each race we have only one instance of this class and it holds all AI drivers in list which is created before the race starts. The number of AI drivers is based on player's settings specified in *Quick race* menu. *AIManager* has two main functions. It contains all common methods and structures for *AIControllers* and manages update of each *AIController* instance.

13.3.1 Steering

Steering is used for vehicles to follow the desired destination. Each turn with driver's steering wheel generates steering which turn the vehicle's front wheels to a desired position.

This section will introduce us to the concept of steering for controlling AI vehicle. Each steering represents steer side and angle of steering wheel. We hold these information in *Steering* class. For determining steer side there are only tree possible values (left, right and forward) and they are equivalent to player's input from keyboard.

During one update there are many generated instances of steering which have to be collected by *AIController*. At the end of the update *AIController* will aggregate all these instances and will

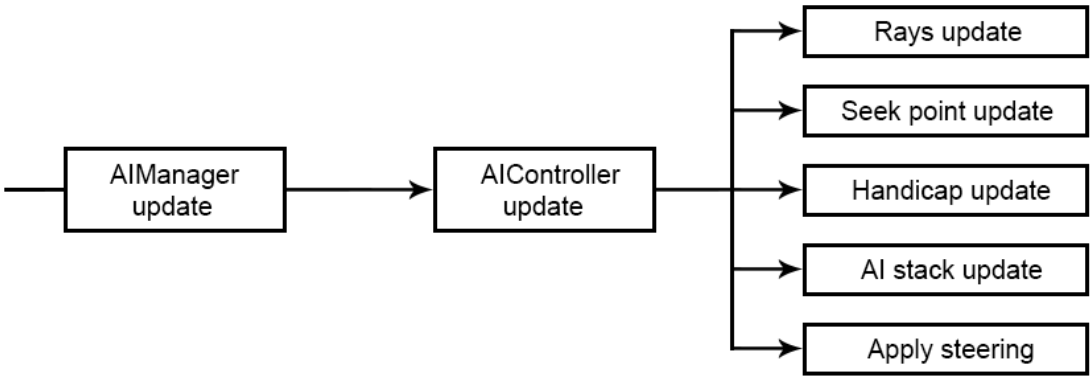

Figure 13.3: Decomposed update of AIController.

produce one steering representing the next move of the AI vehicle. Calculation of final steering is done by simple adding and subtracting angle values. If the steering's steer side is left we subtract the angle value, otherwise we add this angle value (for forward steer side it's zero). If the value after adding and subtracting all steering angles is greater than zero, the final steer side will be to the right. On the other side if it's less than zero final steer side will be to the left. In case the result is zero, the steering wheel is set to the forward direction.

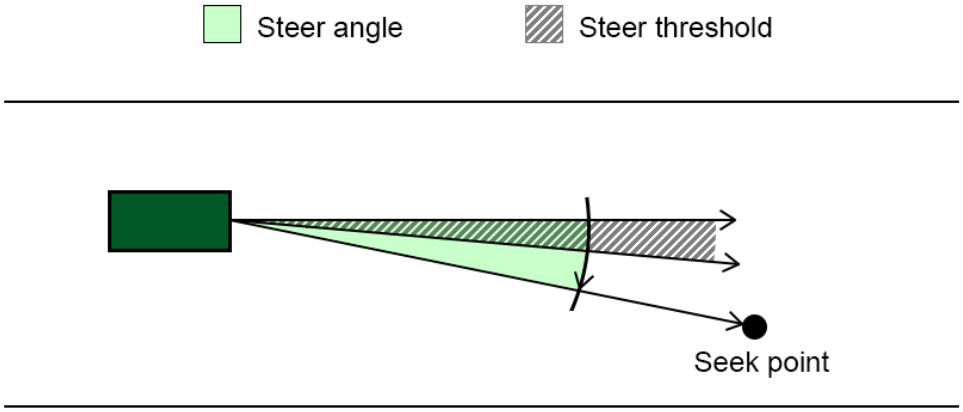

Figure 13.4: Creation of steering action to seek point with defined threshold.

During the implementation our goal was to create steering which will look realistic. Our concept is based on following the driving line defined waypoints. If the vehicle just follows this line, every change in steering will be sharp and not good-looking. To prevent this, we add some features to this following. For more smooth cornering we specify the vehicle's minimal distance to the following point (so-called *steer point*, described in Section 13.4.1) on the driving line. In case the vehicle is following a more distant seek point, the movement of this seek point

is projected as a smaller steering angle. This feature gave us the feeling that the driving line is created with splines and not just straight lines. Second and also very important feature is the usage of a steering threshold shown in Figure 13.4. If we use each final steering generated in every update AI will be trying to correct his direction too often. Even on straight road, the vehicle will be changing direction too fast because after the each used steering, we will never get an exactly zero result for angle to next position of the following point. Because of this, possible situation threshold determines which steering should be used and which not. If the steering's angle is too small (below the threshold value) it won't be used.

13.3.2 Handicap

MotorDead is racing game where action is one of the key parts. Because of that we created the concept of handicap for AI drivers. This concept is well-known in many racing games. It makes the game more interesting for the human player by keeping all drivers together around player's vehicle. As well as in other racing games, the handicap concept in Motordead is based on cheating of AI drivers. To determine how much AI driver should use cheating, we use a simple distances between AI drivers and player. For example, if AI driver is too far behind the player, that AI driver's vehicle will gain more performance and driveability which leads to catching up the player's vehicle. On the other side, if AI driver is too far in front of the player, his performance and the driveability of the vehicle will be reduced so the player will be catching him up easier.

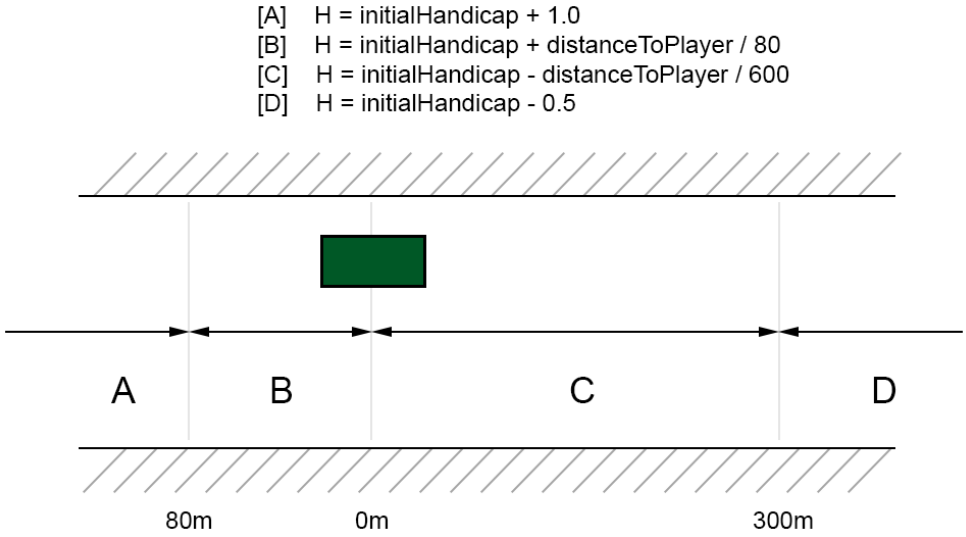

Figure 13.5: Handicap calculation in different distances from player's vehicle.

The calculation of a positive or a negative handicap is depicted in Figure 13.5. As we can see, this calculation gives the different values of the handicap only to a specific distance from the human player's vehicle. This calculation is provided by the equation in Figure 13.5. Outside

this boundary the handicap is set to a constant value no matter how far is the AI driver from the human player. This constant value equals the value of the handicap for the vehicle which distance to human player is the same as boundary distance. In handicap calculation we used race progress distance rather then actual direct distance. Because of that we had to add one small modification when the AI driver is not in the same lap as the human player. This modification recalculates the handicap value as they were in the same lap and its applied only if the AI vehicle's distance to the human player is smaller than the boundary distance. In other words, if the AI vehicle is near the player's vehicle it's handicap is regularly calculated with mentioned equation.

13.4 Actions

In this section, we will describe all possible actions which can the AI driver use during the race. Each action works like an independent unit so it's very easy to edit or add actions without affecting the others. More about a composition of the actions can be found in Section 13.5.1.

13.4.1 Seek

The action seek implemented in the *Seek* class is one of the most important actions. It's the only action which triggers the acceleration of the vehicle so without this action the AI driver won't move.

The core feature of this action is to determine a steering to desired position which we call *seek point*. The seek point represents a position to which the vehicle should be heading to. At the beginning of the race, seek point position is set to the first follower of the starting waypoint. Update of this position is performed in every *AIController*'s update. Seek point moves on driving line defined by waypoints. Because of that each driver holds the value of a waypoint which he currently follows. With this value we can calculate a vector on which seek point should move. If the distance between the followed waypoint and the seek point drops to a small value the next waypoint in the line will be set as currently following and the seek point will move towards it. In case of the fork we chose the next waypoint randomly. Movement speed of the seek point depends only on a position of the vehicle. If the distance between the vehicle and seek point drops under the value of minimal distance defined in *AIConstants* class, seek point will be moved. This implementation lead us to a system where seek point is moving in front of the vehicle in a constant distance, except when the vehicle is driving backwards. This exception we didn't have to handle because the AI driver is designed to follow the seek point, that means he will go backwards only in special cases like avoiding the obstacle. These cases are acceptable because the distance between the seek point and vehicle will rise only by a few meters at most.

For purposes of attacking other vehicles we had to extend the seek action with a concept called **leader following**. Its function is to create an alternative seek point which will be placed on one of the opponent's vehicle and determine if we should use it instead of the default seek point. The alternative seek point is created only in case if the current vehicle has a valid target (by word target we mean one of the other vehicles in the race). If the AI player's target is set depends on his current behaviour. More about behaviours and choosing the target is in Section

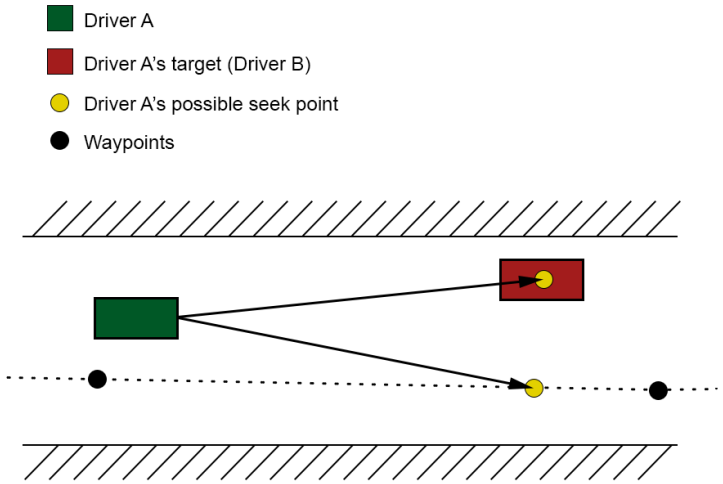

Figure 13.6: Example of possible seek points for one vehicle.

13.5. To determine which seek point we use, we first check few conditions which have to be satisfied. For example, we won't use the seek point on the opponent's vehicle if between our and his vehicle is obstacle or if the opponent's vehicle is off the track.

As we mention before, seek action is the only action which could make the vehicle move. That means, every moving vehicle must have this action enabled. Because of that, the seek action detects specific situation like jumping, rolling over or if the vehicle is stuck. The detection of the first two situations is quite simple because all these information are available from the physics. Detecting if the vehicle is stuck is more complex and we use two tests for that.

In the first test we use two rays shown in Figure 13.7 which help us to detect obstacles in front of the vehicle. If only one of the rays intersects with any obstacle and the vehicle's speed drops under 2 km/h, that vehicle is marked as stuck and a special behaviour for this situation will be added on top of the AI stack from Section 13.5.2.

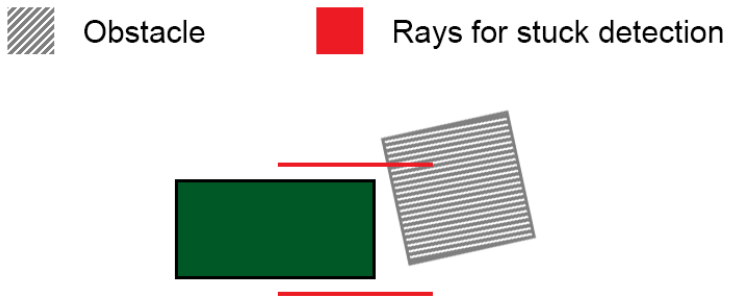

Figure 13.7: Definition of rays which are used for stuck detection.

Rays from the first test are very good for they fast responsiveness, but they have one disadvantage. If the track contains an object that is very small or thin there is a big probability that our rays won't detect this object. For this reason we created a second test which checks the movement of the vehicle. If the vehicle is not moving or its speed is nearly 0 km/h for at least two seconds, its very probable that the vehicle is stuck. Because of that, if we reach this value, the seek action will put on top of the AI stack the same behaviour like in the first test.

13.4.2 Brake

Braking is one of the key features for controlling the AI vehicle. Its implemented as action brake in the *Brake* class. In this implementation we wanted to create a braking system which could be used for many different tracks. To achieve this we created two tracks with lots of differences (width of the track, shape of the curves, track highest speed etc.). On these two tracks we let drive one AI driver who never uses brakes, but we were able to define his maximal speed limit. At the beginning we started with a smaller limit. Then we progressively set this limit to the higher and higher values until we reached critical value when the AI vehicle left the track. When the vehicle left the track we saved it's speed and the angle of the last curve. When we obtained enough data, we created graph with points where each point represents a pair — vehicle's speed and curve's angle. Our next goal was to find a function which will go through this points. For finding this function we used program *Mathematica*, which gave us this result:

$$F(x) = 2.46503 \times 10^{-11}x^6 - 3.01482 \times 10^{-8}x^5 + 0.0000143107x^4 \\ -0.0033001x^3 + 0.379157x^2 - 20.6786x + 569.348$$

We used this function as a basic guideline which tells us when it is necessary to use brakes because it represents maximal speed for each angle of the curve. This concept is depicted in Figure 13.8. The main advantage of this concept is that we can move this function in the vertical direction and create a different driver skills. For example, if we move this function up for just one driver he will be using brakes less often than the others. Because of that in some curves he could be very fast but in some special cases he will have a problem to remain on the track.

13.4.3 Avoid

Avoiding obstacles or overtaking other vehicles belongs to basic driving skills. Our AI driver is not an exception and has these skills too. In this section, we will introduce concepts which we used for the implementation of these skills.

Because of the nature how the AI driver can see the game world described in Section 13.2 we decided to divide this implementation into three parts:

- Avoiding static objects from the scene and handling collisions with track borders.
- Avoiding very slow vehicles, wrecked vehicles, oil spills and spikes.
- Overtaking opponents.

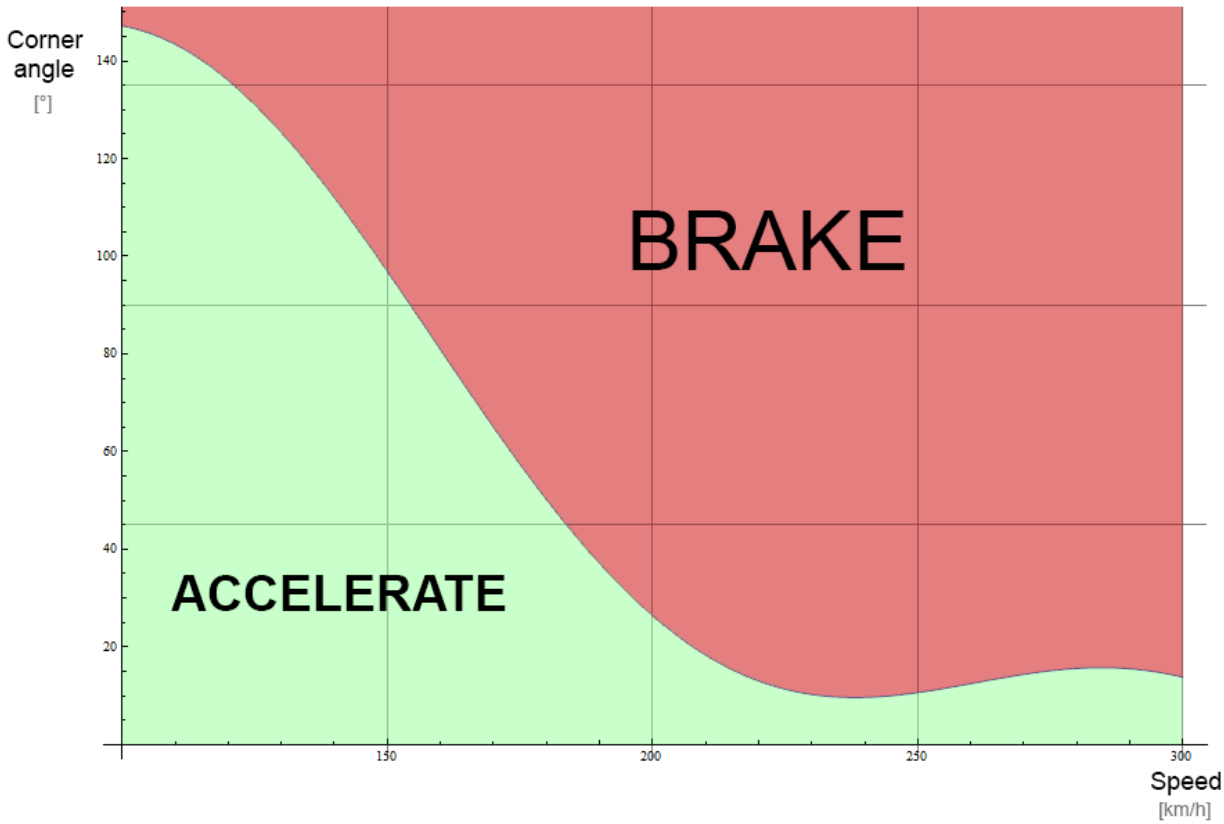

Figure 13.8: Graph representing the function which calculates the braking angle for given speed.

The main reason why we divided this implementation was that AI driver doesn't have the same information for each object in the scene which he has to avoid. All dynamic objects like opponent vehicles, oil spills and spikes AI driver can see only as a point which represents its current position. For detecting all other object he uses so-called AI rays.

AI rays are special rays designed for AI purposes only. They could be referred as AI driver's "eyes" because their main job is to detect any obstacle in the vehicle's way. For a better performance we use only four rays per vehicle. Starting positions of these rays are relatively to the vehicle always at the same place. Endpoints of rays are slightly different during the vehicle's movement because we designed them to adapt for the vehicle's speed and for the vehicle's turning. If the vehicle's speed is increasing the position of the ray's endpoint is set to a greater distance from the vehicle. This feature is used for detecting any obstacle much sooner because with a greater speed we have less time for reactions. In case the vehicle is turning, the endpoint is moving to the same side. How far we move it from the base position is given by the angle of the current turning. Movement of the ray's endpoint to left or right side is used for detecting an obstacle in the bigger scope. For example, if the vehicle is turning left its useless to detect an obstacle in right-front area because the vehicle won't go through that sector. Just because of that,

we move the rays to the left and so we have a better chance to find obstacles which we could hit on the left side. All situations how AI rays can move are shown in Figure 13.9.

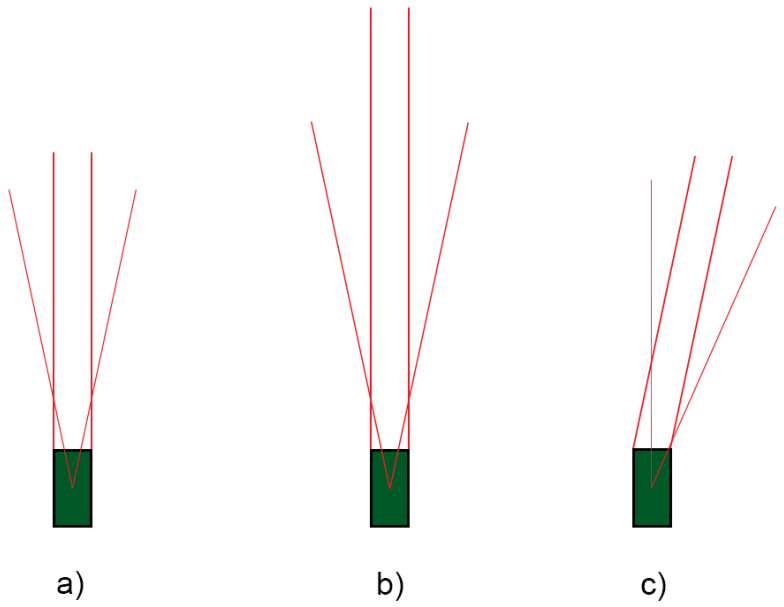

Figure 13.9: Example of three different positions for AI ray's endpoints. a) positions for normal speeds b) positions for higher speeds c) positions for turning right

Evaluation of the AI ray's collision

We have four instances of the AI ray for each vehicle. That means each ray can detect a collision with any object. Each detected collision with an object is considered as a hint which tell us that avoiding this object could be necessary. As we can see in Figure 13.9 AI rays are designed in pairs (left-front/right-front and left-side/right/side). For avoiding redundancy, each pair generates only one steering. For calculation of this steering we had to add some properties for AI rays:

- **Collide** - boolean value which determines if the current ray hit some object
- **Collision distance** - if the current ray collides, this value represents distance between the vehicle and the colliding object
- **Collision angle** - holds value for the steering angle which should be used for avoiding the colliding object. The calculation of this value is based on the collision distance.
- **Steer side** - defines the steering side for the colliding ray.

After each evaluation of the rays we gather this information and then we start comparing values between rays that belong to the same pair. In case the only one ray from this pair is colliding, we take it and create a new steering. In case the both rays are colliding, we take

only that ray collision which have smaller value in property *Collision distance* and create a new steering. All values for the new steering are given by the ray's properties.

Static avoidance

In this section, we will describe avoidance handling for static obstacles. We put only three objects to the category of static obstacles — the track itself, track borders and heavy debris. Each track has one big collision model which contains every static part of the track. This representation is very useful because with AI rays described above we can very easily determine a possible collision with any object which belongs to the track. Specially for our track these objects are representing buildings, train, columns, containers etc.

The second object which we put to the static obstacles category is the track border. The detection of any possible collision with them is done like for the track with AI rays, but with one difference. Track borders don't have any collision model, they are represented only as a line with the starting and the ending point. Because of that, we are using a projection into 2D world where we determine if the AI ray has an intersection with the track border line or not. If there is an intersection, we treat it as a regular collision with an object.

The last object, which belongs to this category is heavy debris. The collisions with heavy debris could have the same effect like collisions with static objects. Because of that, each debris object with a weight greater than 100 kg is considered as heavy and the AI rays are used for avoiding this object.

Dynamic avoidance

In MotorDead game there are only four objects that are handled as a dynamic obstacle — vehicle, oil spill, spikes and smoke screen. To avoiding these objects we are not using AI rays like in the static avoidance. We decided so because some of these objects are too small for this kind of detection and even if they are detected it could be very late when the avoidance is not possible. Because of that, we use only their position in the world to detect possible collision in the future. To save some performance we test only dynamic objects that are in front of the vehicle in a radius of 100 meters. For these objects we calculate a signed angle between the forward direction vector and the vector created from the vehicle's position and the object's position. If this angle is too small, its very probable that we can collide with this object in the future, so we have to handle this situation. Result of this handling is like for AI rays a new steering which should be applied for avoiding this object. The determination of the steering's side and the angle is based on an angle calculated from the signed angle of the vectors mentioned before. If this vector is less than zero we will create a new steering with the steer side set to left, otherwise we will create a new steering with the steer side set to right. The value of the steering's angle is not calculated from the angle of the vectors but we use a constant value which represents the angle needed for proper avoidance of the dynamic object.

In implementation of the calculation described above we have one difference for avoiding the vehicle. Each vehicle unlike the other objects have information about the nearest track

borders. This information allows us to set the steering's side for the avoidance more properly. For example, if the wrecked vehicle is on the left side of the track, proper avoidance side would be to the right. This value we can get from the nearest track borders of the wrecked vehicle very easily. All we have to do, is to compare vehicle's distance to right and left side of the track. As we mentioned before, other obstacles don't have information about how far they are from each side of the track, because of that we choose the steer side only from the value of the signed angle between the obstacle and the vehicle's forward direction vector. This may sometimes cause that if the obstacle is for example on the left side of the track, the vehicle will try to avoid it from the left side.

The dynamic avoidance has one special case — the avoidance of the smoke screen. Size of it's effect is so big that it's nearly impossible to avoid it. For this reason we decided to handle smoke screens differently. If the smoke screen appears in front of the AI driver he will ignore it and in case he enters it he will stop to accelerate and all avoidance mechanism will be skipped. We decided to do this for creating more realistic behaviour of the AI driver.

Overtaking

The overtaking could be taken as a form of avoiding the obstacle which moves nearly the same speed as the overtaking vehicle. The next difference is that the overtaking and the avoiding obstacles don't have the same weights in calculations for the final steering. It is logical that the avoidance is more important then the overtaking, because with a bad avoidance decision we can loose more than with the bad overtaking decision. For this reason when the AI driver knows that he needs to avoid any obstacle the calculation for the overtaking will be skipped. Similar reaction is used when the AI driver is heading towards the jump. Ramp for the jump could be very narrow and if the driver tries to overtake another vehicle he can fall from the ramp. Because of that, every time when the AI driver's current waypoint which he follows is marked as *Jump*, his overtaking calculation is skipped.

13.4.4 Attack

MotorDead is a action racing game which allows usage of weapons. Driver's main goal is to finish the race on the first place, but for that sometimes he needs to destroy other vehicles. Usage of weapons is also important to keep the game interesting for the human player. AI drivers use weapons according to implementation in the *Attack* class. In this implementation we took all weapons defined in the game and created separate methods which determine their usages (*UseMachineGun*, *UseRocketLauncher*, *UseGrenadeLauncher*, *UseOilSpiller*, *UseSpikesSetter*, *UseSmokeScreenCanister*). For easier deciding, if the driver should use the weapon we created the list of properties and values for each weapon type:

Minimal distance

Effects of some weapons have greater range from the position of the impact. As an example we can take the rocket launcher's explosion. Because of that an unwise usage could lead to damaging not only the opponent's vehicle but ours as well. For this reason we defined

for each weapon type a value for a minimal distance between the attacking vehicle and its target.

Maximal distance
All weapons have their own specifications. One of them is a range distance. Its pointless to try attacking opponents which are beyond this range. Because of that, we specified for each weapon type its maximal distance to which the AI driver should try attack the target.

Frequency of use
Actions are updated every game update, so if in one update the AI driver fires from any weapon its very possible that all conditions will be satisfied in next updates as well and he will try firing again as soon as the weapon is ready. This leads to use of all the ammunition in a very short time. Because of that, we defined for each weapon a minimal delay between the two usages.

Maximal angle for aiming
Each weapon creates a different effect near the position of the impact. The radius of this effect can be bigger, that means if we don't hit the vehicle directly its possible that we can damage it anyway. For this reason we specified for each weapon the maximal angle to the target. If our current angle to the target is smaller then this value we can try to shoot with a good possibility of damaging the target's vehicle.

13.4.5 Nitro

Nitro in the racing game is very useful and creates a great advantage in particular situations. That's the reason why our AI drivers handle the usage of the nitro too. There are many differences between just using the nitro and using it correctly. Because of that, we created independent *Nitro* class which determines when is the usage of the nitro needed at most. This decision is made from several conditions which have to be satisfied if the nitro will be used. If we use the nitro randomly, in many cases, like using almost empty nitro or using it before the sharp curve will reduce its effect to almost none. Because of that, our AI uses the nitro only in two cases:

- Track before the vehicle in a specific distance is almost straight.
- Vehicle's speed drops to values below 50 km/h.

For both cases we also check available amount of the nitro. So in spite of the conditions from the cases are satisfied, the nitro will be used only if we have available at least one-third of the full nitro.

13.4.6 Jump

The jumping vehicle is very common in the MotorDead. Because of that we have to prepare AI for special cases which could bring this fact. At the beginning of the implementation our biggest problem was during the landing, when the AI driver tries to turn left or right immediately

when his wheels touch the ground. This behaviour led to many crashes so we decided to create a special action for handling jumps of the vehicle. We implemented this action in the *Jump* class and its only function is to detect the landing after the jump and prevent any turning of the steering wheel during a specific amount of the time.

13.4.7 Return on track

When the vehicle gets off the track its not always the driver's fault. So even if we create a perfect AI driver he can still gets off the track because of the other vehicles in the race. Our track in the game is quite complex and unlike classic racing tracks it has many obstacles, especially off the track. Based on these facts its very common situation that the vehicle could be stuck on many places along by the track. The vision of the world is for the AI driver limited by rays used for avoiding and positions of obstacles and vehicles. Because of that, the AI driver could be stuck a very long time. To prevent this situation we created a specific *ReturnOnTrack* class which handles situations when the vehicle couldn't move forward towards the seek point.

The detection if the vehicle is stuck is done by the Seek action from Section 13.4.1. *Return on track* action has a different role. This action is responsible for the returning of the stuck vehicle on the track. The AI driver always tries to go forward, so if he is stuck its probable that some obstacle is right in front of him so he should try to go backwards first. This is done by *Return on track* action which always holds the brake with released acceleration. For more efficiency, the action uses a steering which tries to rotate the vehicle so the seek point will be in front of it. In Figure 13.10 we can see how we determine the side for these steering (the value of the angle is not important because one update of this action will always produce only one steering).

Most important task for this action is to determine when the vehicle should not be marked as stuck anymore and should try to continue the race. This determination depends on three conditions which have to be satisfied:

- Vehicle's seek point have to be in front of the vehicle.
- In front of the vehicle couldn't be any obstacle. (For this test we use extended rays from Figure 13.7)
- Line between the vehicle's position and the seek point's position must not have any intersection with the world.

If all conditions are satisfied the vehicle is no longer marked as stuck and resumes the race. From the nature of this handling we can see that the vehicle could be stuck in a way where just going backwards will not be enough. For handling this situation we used a concept of resetting the vehicle to the last visited waypoint. For the human player could be this reset distracting so we decided that the AI driver can reset his vehicle only if the human player's vehicle is at least fifty meters from him.

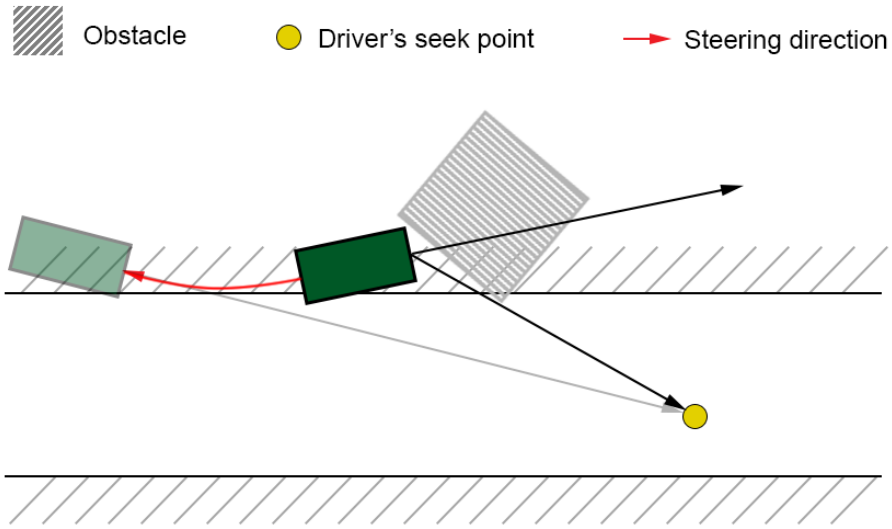

Figure 13.10: Calculation of steering used for vehicle which goes backwards.

13.4.8 Start

Start is a specific part of the race. AI drivers have to wait for a countdown like human player which can't move. For handling these first seconds of the race we created the action start implemented in the *Start* class. Its function is to activate the handbrake during the start which prevents the vehicle from moving. For the player's better feeling of the game we also added a feature that makes the vehicle to accelerate during the countdown what makes better acoustic effect.

13.4.9 Stop

Action stop is the simplest action because it has only one function. This function is to stop accelerating and to use the brake with the handbrake for a better effect. Stop action is used only in two situations which require stopping the vehicle. First is at the end of the race and second is when the vehicle is wrecked.

13.4.10 EndRace

At the end of the race when the vehicle crosses the finish line the AI driver should behave differently. We decided that the AI driver should continue driving for an unspecified amount of the time. This amount we will get randomly from values between 5 and 10 seconds. Immediately when the vehicle crosses the finish line the countdown from this value starts and when this amount of the time expires the vehicle will stop with the action stop described above.

13.5 Behaviours

If we use for creating our AI all algorithms and principles described in previous sections we will end up with many similar drivers. Their reactions for all situations will be always the same and that is not what we want because it doesn't look very realistic. To prevent these similarity we decided to implement some differences and put some randomness to our calculations. To manage differences between opponents we use the database mentioned in Section 13.6. In this database we are able to describe some basic properties for each driver. For example, we can define how much careful he drives, how often he tries to overtake opponents or how often he uses weapons. These attributes are represented with values of *Aggressivity*, *Overtaking* and *Braking*. Their values are from 0 to 10 where smaller value means that driver is less aggressive, he overtakes less and uses more brakes. In next sections, we will describe how we create and how we use these behaviours.

13.5.1 Behaviour creation

Each behaviour is created as a set of actions described in Section 13.4. The nature of the final behaviour depends only on which actions are included to this set. For creating new behaviours we have to extend the base *Behaviour* class, which holds implementation for proper updating of the actions. In MotorDead we end up with seven different behaviours. In the list below we can see the composition of these behaviours:

- **Basic behaviour** - Seek, Avoid, Attack, Brake, Nitro
- **EndRace behaviour** - Seek, Avoid, Brake, EndRace
- **JumpHandling behaviour** - Jump
- **ReturnOnTrack behaviour** - ReturnOnTrack
- **SeekAndDestroy behaviour** - Seek, Avoid, Attack, Brake, Nitro (with possible target)
- **Start behaviour** - Seek, Start
- **Stop behaviour** - Avoid, Brake, Stop

The advantage of this component approach is in possibility to alter existing behaviours with custom actions. It gives us an opportunity to have more implementations for the same action and it's up to us which one we use for the current behaviour's definition. In MotorDead we have for each action only one implementation.

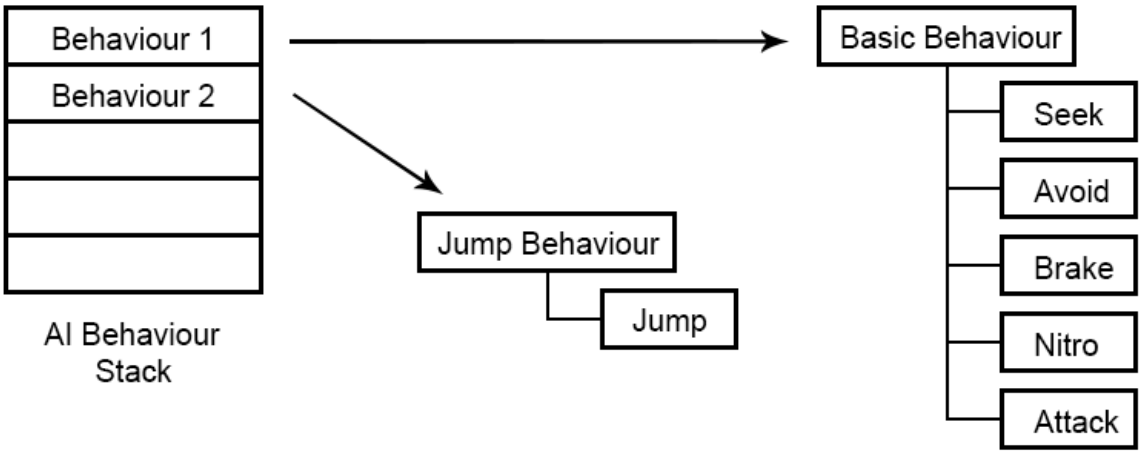

Figure 13.11: Decomposed example of AI behaviour stack with two behaviours.

13.5.2 AI behaviours stack

For purposes of changing behaviours during the race we created the stack of AI behaviours. Every AI driver has his own instance of this stack. In Figure 13.11 we can see a decomposition of the AI behaviour stack example.

Behaviours are divided to long-term and short-term behaviours. Short-term are used for specific occasions during the race like jump handling, stopping the vehicle at the end of the race, returning on the track etc. These behaviours are simple because mainly they are composed only from one action which handles it's current situation. Long-term behaviours are used to represent drivers personality in general. Each long-term behaviour is composed from set of actions. Adding or removing even one action always creates a new behaviour. For example, if we have a driver with the behaviour that is created from all actions and we remove just the *Attack* action, the driver will never use weapons.

Updating the AI stack is handled in the *AIController* class. This update has two parts. In the first part we calculate time from the last long-term update. If this time is greater than a specified period, *AIController* calls the method *UpdateStack* which will update the AI stack with a possibility of replacing the long-term behaviour from top of the stack.

The second part of the update in *AIController* gets any behaviour (long-term or short-term) from top of the AI stack and triggers its *Update* method. This method takes all actions from current behaviour and triggers their *Update* method. This flow of updating the AI stack is depicted in Figure 13.12.

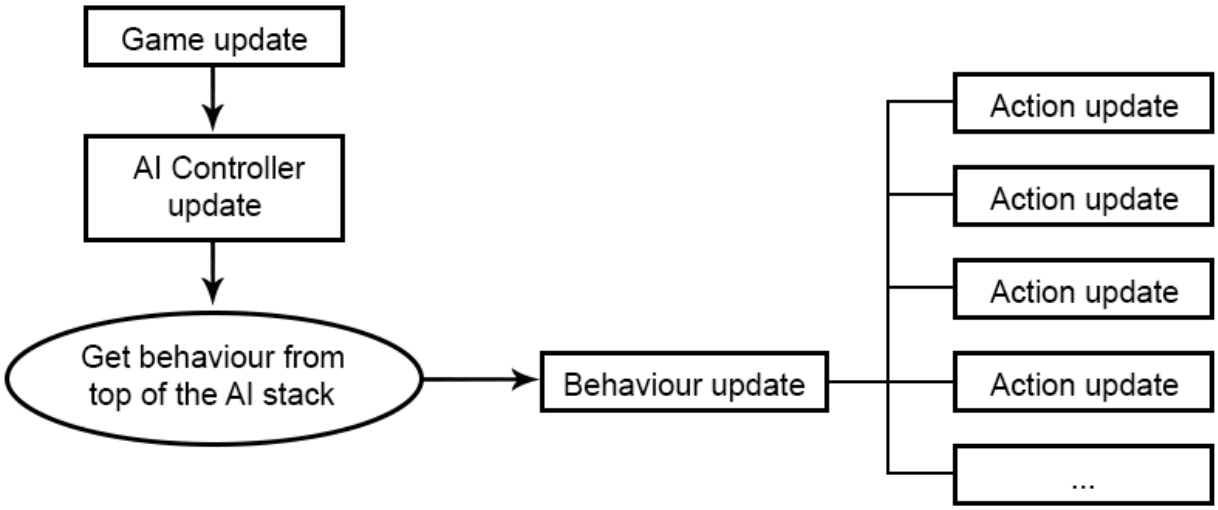

Figure 13.12: Flow which demonstrates one update of AI.

13.5.3 Revenge

Revenge is a very common human behaviour even more in games like MotorDead. For implementation of this behaviour we used a simple system which counts the damage dealt between all drivers. At the beginning of the race all counters are set to 0. If the driver A dealt damage to driver B, the driver A's counter for the driver B is decreased by the value of dealt damage. On the other side, the driver B's counter for the driver A is increased by this value (In Figure 13.13 is shown this case with damage equal to 8 points). These damage counters are stored as a two-dimensional array in the *AIManager* class. It's obvious that the dimension of the array is based on the number of players. For n players we have $n \times n$ array.

Revenge is applicable when the AI driver is about to choose his next target. This part consists of few steps. The AI driver gets values from the table with damage counters. Next he selects a table row assigned to him and finds the greatest number in this row. If the number is found and it's unique his next target will be a driver who represents the table column with this number. In case more than one number with the same value are found, next target will be selected randomly. For making this game more difficult and more action for the human player, we affected this randomness. Because of that the human player is chose as a target with the probability of 50%. Remaining 50% is divided between other drivers.

13.6 Opponents

This section describes how we create opponents for each instance of the race. For storing all opponents which can be chosen for the race we created an XML database. Each opponent in this database has several information which describe his nature:

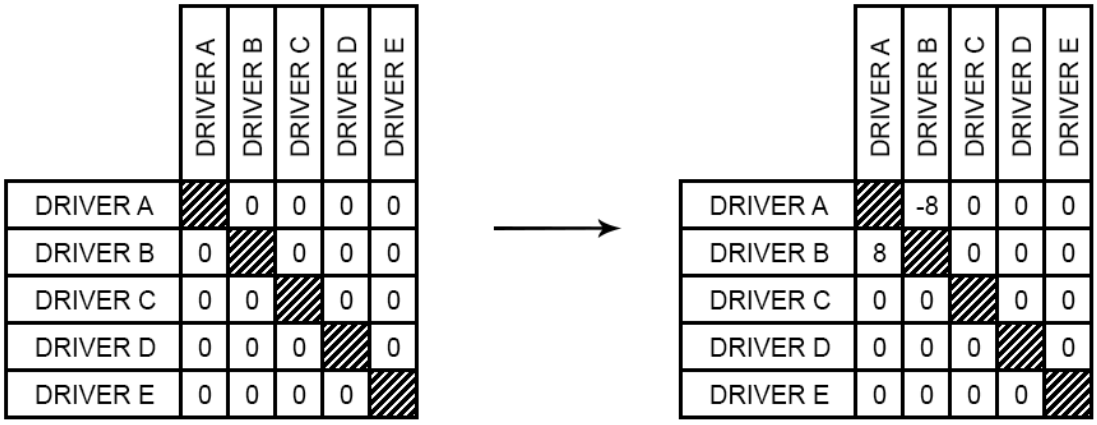

Figure 13.13: Updating the attack counter. In this example, the player A dealt damage to player B.

- **Id**
- **Name**
- **Braking**
- **Overtaking**
- **Aggresivity**
- **VehicleEntry**

Before the race starts we randomly chose as many opponents as the human player chose in the quick race menu screen. For each of them we get all information from the database and store them in the *Opponent* class.

For making opponents more adaptive to current state of the player's vehicle we created system which alters chosen opponent's vehicle entry. This modification is based on current upgrade level of all parts on the human player's vehicle. Each part of the opponent's vehicle gets nearly the same upgrade level as the human player's vehicle. We created a discrete Gaussian distribution where we have 70% chance to get the same level, 15% to get level decreased by one and 15% to get level increased by one (If we have the lowest level of the upgrade there will be no decreasing, same for the highest upgrade and increasing). Because of the possibility that the human player has the vehicle with a poor driveability, we add a constraint for the engine and breaks upgrade level which prevents the creation of a vehicle with the powerful engine and poor brakes.

Chapter 14

Persistence

This chapter describes local data persistence.

14.1 Data to Keep

The application requires to keep local information about user profiles, their progress through the game, their keyboard and audio settings, and information about video settings (which were decided to be separated from user profiles to prevent unintentional resolution changes when switching users).

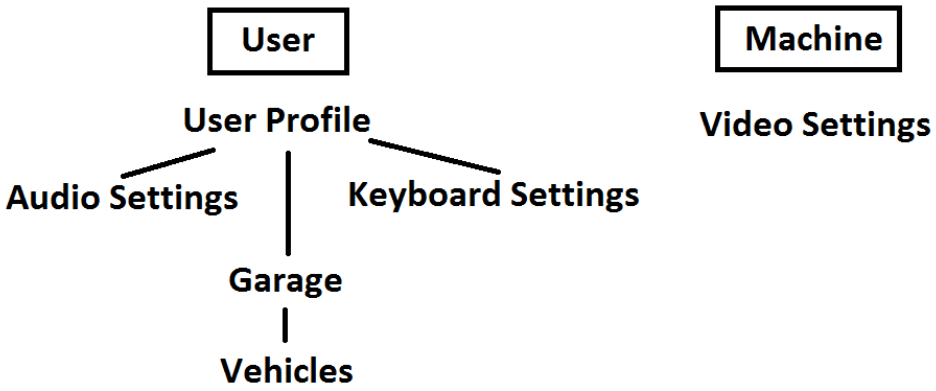

Figure 14.1: Structure of user and application related settings. Nodes contain information in string format.

14.2 Serialization and Deserialization

To keep necessary information, the application uses XML[1] files which are saved in user's local drive. Typically `%appdata%\Roaming\FightRaceGame\`.

To convert object structure (serialize) into XML, *DataContractSerializer* and *XmlSerializer* is used. The settings are separated (as clear from Figure 14.1) into two files - `profiles.xml` and `gfxsettings.xml`.

To recreate object structure (deserialize) from a XML file, *XmlSerializer* is used. Note that the file may be missing or corrupted. In that case, the exception from deserializer is caught and the file is created with default content (video settings or a default profile).

14.3 Security

The developers realize that keeping game data in text (XML) allows the user to alter it (e.g. to cheat up credit). Since the game has not been released, it would only make debugging more difficult to implement some cryptography. If the game gets released, it is simple to change the serializer to a safer one.

[1] http://en.wikipedia.org/wiki/XML

Chapter 15

Audio

In this Chapter we will discuss sound effects and music. We are using the Cross-platform Audio Creation Tool (XACT) system to play all the sounds and music in the game. Support for XACT is part of Microsoft XNA.

One major downside of XACT we would like to mention here is that there is no way to query a sound library for whether a sound exists in it (without triggering an exception). Therefore, we had to carefully prepare placeholders for all the sounds we were implementing. This created unnecessary production pipeline slowdown.

Apart from that, XACT is a powerful tool, it's part of XNA and our sound designer was already skilled in its usage, so we decided to go with this technology.

Most of the principles and implementations described in this chapter are contained in the *SoundManager* class.

15.1 Cue

In XACT terminology, a cue is a high-level abstraction of a sound. It is the interface between the programmer and the audio designer.

A cue can be composed of several wave samples with different probabilities of being played. For instance, in MotorDead, there is one cue representing the sound of a machine gun being shot, so the programmer plays it when necessary. However, the cue is composed of 12 different wave samples, each having a $1/12$ probability of being played when the cue is triggered. This is common practice in sound design to achieve high-fidelity and thanks to XACT, it is completely transparent from the programmer's point of view.

Cues can have so-called "RPC" variables associated with them. These are floating point values assigned by the programmer in order to give him some procedural sound control. For instance, the sound of a running engine is dependent on the RPM (revolutions per minute) of the engine and on the throttle (a value indicating "how much is the gas pedal pressed"). These two values are passed to the XACT engine from the game by calling the *SetVariable(variableName, variableValue)* method on an instance of the *Microsoft.Xna.Framework.Audio.Cue* class.

15.2 Positional sounds

For an immersive game experience, it's necessary for some of the sound effects to emulate positioning in space. This means, for instance, that when a grenade explodes to the left of the game avatar, the sound effect associated with the explosion should be louder in the left speaker. Moreover, all cues have a distance RPC variable by default that the audio designer can use to make the loudness of a sound fall off with distance. This one variable is automatic (implicit) and gets its value from the sound positioning information.

The audio engine handles these calculations, but we have to provide information about the positions of the 'emitters' (sources of sound effects) and of the 'listener' (the camera used to render the game). We do that by calling the method *Apply3D(AudioListener, AudioEmitter)* on a *Cue* instance. If the emitter is not moving, we only do this only once, before the cue is played for the first time.

When the *Apply3D* method is called for the first time (which has to be before starting the cue's playback or never), the sound sample is collapsed into a monaural (mono) sample. If the *Apply3D* method is not called at all, the sound is played without positioning, but in its stereophonic form (if available). Using a stereo sound is a better choice for abstract sounds that do not have a physical emitter (such as the game announcing that you passed a checkpoint) or for sounds that are at all times positioned at the same place with respect to the listener. We will explain the latter case in the following paragraph.

Sounds emitted by the player's vehicle are always emitted in front of the camera and if we used positioning on them, the resulting effect would be almost equal in both the left and right channel (speaker). By using stereo sounds without positioning for the player's vehicle, we end up with better sounding results. Take, for instance, the engine sound. By using a stereo sample instead of a mono sample for the player's car, the sound designer can immerse the player much better. In the code, the property *IsSpatialized* in the *Vehicle* class decides whether to use a 3D sound or not.

15.3 Engine sounds

Engine sounds are the most difficult to create. This is largely due to the fact that they are controlled by two variables: RPM (revolutions per minute of an engine) and throttle (parameter from range $[0, 1]$. Throttle says "how much is the gas pedal pressed": zero - no throttle, one - full throttle). At the time of writing this document, we are using only one engine cue in MotorDead. This cue was created using the samples from `http://www.sonory.org/`, that can be used for non-commercial purposes. To combine the sample to the sound of an engine we used method described in [6].

15.4 Collision sounds

Sounds of the collisions among game entities are played by the *PhysicsManager* class. The details of the method can be found in Section 8.7.1. The *PhysicsManager* class uses the materials of the colliding entities to determine which sample should be played. For more details about entity materials, see Section 5.7.

Sound properties of an entity material are given by a member of the *SoundMaterial* enum. We decided to use only limited number of sound materials, because our sound designer decided to create one cue for each pair of materials. Each entity material has assigned a member from the *SoundMaterials* enum that most closely resembles the material.

We have two available collision cues for each pair of sound materials. One of them is used for smaller collisions and the other for large ones. To determine which cue should be used, we look at the magnitude of the collision impulse. Note that for the collision sounds that involve vehicles we use three cues instead of just two. The vehicles are the central point of the game and using three collision levels for these sounds gives a better impression. The *CueNamesMapping* class is used to combine a pair of the sound materials and a strength of the collision impulse into a name of the cue that should be played.

15.5 Optimization

We make use of a very simple optimization. The positional sounds that are further away from the listener's position then the given threshold are discarded and not played. We use this optimization to prevent from playing sounds that are too far away and the user wouldn't hear them anyway because of the distance attenuation. It helps especially in case of the sounds caused by the collisions among objects. The optimization is implemented in the *PlaySound3D* method of the *SoundManager* class.

This optimization is not ideal. In a case when the played sound is long, the position of the listener might change in a way the user should hear the sound. However, the sound effects currently used in MotorDead are short enough and the position of the listener is changed with the movement of the user's vehicle (which moves within a reasonable velocity limits).

15.6 Scraping sounds

Scraping sounds, in reality, are produced when an object is in contact with other object and moving along it (e.g. fingernails on a blackboard). When using discrete rigid body dynamics as the basis for the physics simulation (see Section 8.1), it is difficult to trigger such sounds, that may have arbitrary duration. Therefore, we neglect scraping sounds in general and implement them only for the vehicles (i.e. for the continuous contacts between a vehicle and other objects).

A scraping sound is implemented by a looping cue. A scraping sound is triggered when the vehicle is in continuous contact with other entity - it is in contact with the entity and it has been so in the previous frame (see Section 8.7). Discrete rigid body dynamics cause a small issue -

what appears to be one continuous contact, may be series of the separate collisions (e.g for two frames the vehicle is in contact with a wall, then for one frame it isn't, then again for two frame it is). To ensure that a cue is not played and paused in every few frames, it is always played at least for a minimal duration (currently set for 0.2 seconds). When a new contact comes during this period, the cue is again played at least for 0.2 seconds from the moment of the new contact.

There is also a support for different materials of the objects colliding with a vehicle. For each of the audio materials (see Section 5.6) a cue is prepared. The scraping sounds are implemented by the *VehicleSound* class. The *Vehicle* class sends information about the collisions to the *VehicleSound* class and the *VehicleSound* class manages the cues.

Chapter 16

Plugin

16.1 Introduction

Fibix is not only a 3D graphics engine. It is developed together with Fibix Editor. The main goal of the editor is architectural visualization. The editor is designed for creating scenes and animations. In MotorDead the editor is used as tool for creating the content of the game: preparing the models, converting them to the game's internal format and creating the tracks.

Fibix Editor allows its extension by adding new plugins. The decision to create the plugin was motivated by the need to prepare the data for AI controlled vehicles (see section 16.3). Originally the plugin should support only the editing of the AI data. Later, more features were added. The features are mostly related to creating tracks for MotorDead.

The idea of placing various physical entities to the game world together with the track has been active from the beginning of the project (e.g. adding debris to the track). Without the artist, it was clear that the tracks would be created by the programmers. Therefore the support of placing entities was not at all user friendly. The entities were included according to the dummy points (for definition of the dummy points, see Section 6.5.1). The name of a dummy point would define the entity that is included and the world transform of the dummy point would be used as the world transform of the entity.

Later the MotorDead team has been joined by graphics artists. Naturally, they need better support for placing entities on the track. Therefore the plugin was radically extended. The plugin allows adding some of the physical entities to the scene and exporting them with the track. A living instance of *FightRaceGame* is present inside the plugin. It is used to create new physical entities and it can run the physics simulation (which moves the objects in the editor).

Note that the plugin is **not a part of the game** and it is not distributed with game. It is intended as an **internal tool** that supports the development of the game. It can be even called "experimental" due to the missing documentation of the plugin system in Fibix Editor. As a result the plugin in current state is not entirely stable and bug-free. However, it can be used to create new tracks (and it was used extensively to create the track distributed with the game).

Note that the works on plugin started before the name of the game was chosen. Therefore, there are occurrences of the "FightRace" string which was the the working title of this project.

16.1.1 Editor terms

In this chapter the following terms are used:

Editor scene
: Editor scene is the scene being edited in the editor and displayed in the editor rendering window. In this chapter when it might be called "scene" for convenience.

Entity object
: Every object present in the scene is an entity object. The class hierarchy for the objects at the editor level reflects the hierarchy at the level of engine (or engine wrapper). The base class is *EntityObject* which wraps *C_RenderEntity* class. The most used derived object is render object that represents object with renderable geometry (i.e. a model or part of the model). It is implemented by *RenderObject* class (it wraps *C_RenderObject*). For more information about the render entity hierarchy, see Chapter 6. New types of entities can be added by deriving new class from *EntityObject*.

Editor mode
: Editor mode changes the the behaviour of the rendering window. For example, when the left mouse button is clicked, an object under the cursor is selected. This is the default behaviour implemented by the "Editor"" mode. However, when the left button is clicked in the "Painter" mode, a new decal is added to the scene. It is possible to implement custom editor mode and react to the mouse commands in any desired way.

16.2 Plugin structure

The plugin is implemented by the *FightRacePlugin* class. The *FightRacePlugin* class implemented the *I_Plugin* interface. The *I_Plugin* interface is part of the Fibix source code and it is used for communication between the editor and the plugins. The *FighRacePlugin* class is relatively complex and it use the mechanism of partial classes and it is divided into three files.

16.3 AI data

FightRace plugin allows creation and editing of two new objects, which are used for the AI purposes. These objects are waypoints and track borders. Their usage in the game is described in Section 13.2.

In the plugin (i.e. at the level of the editor), the waypoints and the track borders are implemented by the *Waypoint* and *TrackBorder* classes respectively. These classes are located in the *FightRacePlugin* namespace. Do not confuse them with the *Waypoint* and *TrackBorder* classes from the *FightRace.AI* namespace, which are used by the AI in the game.

The *FightRacePlugin.Waypoint* and *FightRacePlugin.TrackBorder* classes are used for the plugin purposes only. The connection between the plugin's classes *Waypoint* and *TrackBorder* and the equally named classes in the game is only through the serialization. Plugin's classes

contain much information needed only for Fibix editor (and useless for the AI purposes). For the serialization the *XMLWaypoint* and *XMLTrackBorder* classes are used. The serialization classes contain only information, needed to prepare the instances of the *Waypoint* and *TrackBorder* classes in the game.

During the serialization, for each waypoint or track border in the editor scene, the plugin creates a corresponding instance of the *XMLWaypoint* and *XMLTrackBorder* classes. These instances are serialized into one file, which can be used in the game.

For more user friendly editor interface, the plugin allows an option of loading previously serialized AI data. After a user selects the file with the AI data, all waypoints currently present in the scene are removed. When the removal is complete, the plugin creates new instances of *Waypoint* class for each waypoint entry in the selected file and adds them into the scene.

For managing the waypoint and the track border objects in Fibix editor, a custom editor mode ("FightRace edit" mode) is used. The mode is implemented by the *FightRaceWaypointModeHandler* class. This mode is special for its method of creating new objects. When the user clicks into the editor scene, a ray is cast to the scene. A new object (waypoint or track border) is created at the ray hit position. Detailed description how we can create and edit these object can be found in Appendix E, in Section E.3.1.

16.4 Entities

The plugin also supports placing of physical entities to a track. This allows an artist to add the entities to the scene and tune their properties and world transform. The description of the entities are saved with the track. When the track is used in the game these entities are created (according to the saved description) and added to the game world.

For each type of the physical entity that should be added to the scene there is class that is used as scene object. Such a class represents the physical entity at the editor level. Each of these classes implements the *IPhysicsEntityNode* interface. The *IPhysicsEntityNode* interface is used to determine the objects representing physical entities among all of the scene objects (for example in situation when all physical objects have to be found).

Currently there are three types of the physical entities that can be added to the scene and saved with the track:

- Debris - represented by the *DebrisNode* class (for more information about the debris, see Section 12.7).
- Static collision object - represented by the *StaticObjectNode* class (for more information about the static collision objects, see Section 12.8).
- Trigger - represented by the *TriggerNode* class and particularly by the subclasses of the *TriggerNode* class (for more information about the triggers, see Section 12.9).

The classes for the scene objects representing the physical entities must be registered in the editor to be used. The registration is done during the plugin initialization in the *RegisterNodes*

method of the *FightRacePlugin* class. Together with the classes for scene objects, the *RegisterNodes* method also register the XML files with the description of the UI of the scene objects. The XML files defines which of the properties of the objects are displayed in the editor.

16.4.1 Physical entity wrapping

The classes that implements new scene objects at the editor level make use of the generic *FightRacePropertyWrapper* and *DotNetPropertyWrapper* classes. It allows the automatic wrapping of properties of the physical entity and adding them among the editable values of the scene object.

For all of the properties of the wrapped physical entity that are annotated with the *EditableProperty* annotation, the editable values are created and added to the scene object. The property wrapper also keeps the link between each property and the corresponding editable value. The values can be updated in both ways. The relationship between a scene object and its wrapped physical entity are showed in Figure 16.1.

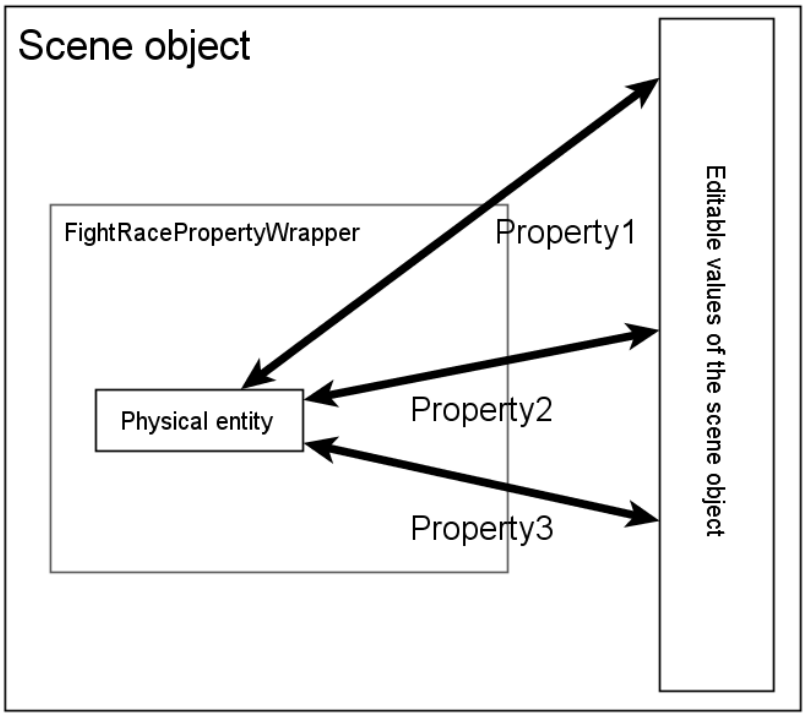

Figure 16.1: Relationship between a physical entity object and its wrapped physical entity. The arrows display the links between the properties of the physical entity and the editable values of the scene object.

16.4.2 Debris

At the editor level the debris object is represented by the *DebrisNode* class. It wraps on instance of *Debris* class (see Section 12.7) using the *FightRacePropertyWrapper* class. The *DebrisNode* class also uses an instance of the *CollisionShapeVisualizer* class to visualize the collision shape of the debris. *CollisionShapeVisualizer* automatically adds values to the user interface of *DebrisNode* and also reacts on their changes.

The debris are added to the scene by the menu command. This command is added during the plugin initialization in the *InitUI* method of the *FightRacePlugin* class. When the command is activated by the user, it raises an event. The event is handled in the *CreateDebrisAction* method of the *FightRacePlugin* class.

The *CreateDebrisAction* method displays dialog for choosing the file that contains model dump. The chosen file is then set to a newly created debris node. The file is used as the model of the debris and it is serialized with the debris. The file name is among the properties of the *DebrisNode* class that are displayed to the user, but it can't be changed.

16.4.3 Static collision objects

The *StaticObjectNode* class implements the static collision objects at the editor level. Similarly to *DebrisNode* it uses the *FightRacePropertyWrapper* class to wrap the *StaticCollisionEntity* class and the *CollisionShapeVisualizer* class for displaying the collision shape.

Static collision objects are also added to the scene using the menu command. The events from the command are handled in the *CreateStaticObject* method.

16.4.4 Triggers

The triggers are represented by the subclasses of the *TriggerNode* class. The *TriggerNode* class contains the most of the implementation needed to represent a trigger. There is the *CreateWrappedTrigger* method that is overridden in the subclasses and is called to create a particular type of the trigger.

There are four types of the triggers in MotorDead:

- Speed-up trigger - represented by the *SpeedUpTriggerNode* class.
- Repair trigger - represented by the *RepairTriggerNode* class.
- Ammo trigger - represented by the *AmmoTriggerNode* class.
- Environment trigger - represented by the *EnvironmentTriggerNode* class.

The *SpeedUpTriggerNode*, *RepairTriggerNode* and *AmmoTriggerNode* classes are very simple and they contain only the *CreateWrappedTrigger* method. The *EnvironmentTriggerNode* class, on the other hand, implements the link between the wrapped environment trigger and a debris

object (see Section 12.9). In the constructor of the *EnvironmentTriggerNode* class a new editable values is used which allows the artist to choose the debris object that is controlled by the environment trigger.

The triggers are added in the specific editor mode. It is implemented in the *TriggerEditorModeHandler* class. Since the triggers are graphically represented as decals, the *TriggerEditorModeHandler* class is derived from the *PainterModeEventHandler*. The *PainterModeEventHandler* is part of the Fibix source code and it is used for adding decals in the scene.

The *TriggerEditorModeHandler* class changes overrides the *OnLeftMouseButtonDown* method which is used to implement behaviour of the left mouse button click. The *OnLeftMouseButtonDown* method casts a ray into the editor scene. If the ray hits any object, the *OnLeftMouseButtonDown* displays a dialog for choosing the type of the trigger. The user chooses the type and the trigger is created at the position of the hit.

16.4.5 Entity serialization

The export of the entities from the editor is done via XML serialization. Because of this, the type of the *WrappedEntity* property of the *IPhysicsEntityNode* class is the *XMLSerializablePhysicsEntity* (and not the *PhysicsEntity*). The *XMLSerializablePhysicsEntity* class contains the of the *GetSerializedData* method which returns an instance of the *XMLSerializedEntityData* class. This instance contains all data necessary to recreate the entity. The particular serialized entity data are implemented as internal class in all of current serializable physical entities (i.e. *Debris*, *StaticCollisionEntity*, *VehicleTrigger* and the subclasses of the *VehicleTrigger* class).

XMLSerializedEntityData are serialized and deserialized using the data contracts. This approach is used to automatically serialize all necessary private fields by using annotations. The *XMLSerializedEntityData* class also contains the *Deserialize* method that reconstructs the entity from the data.

Before the serialization as such all of scene objects that implement the *IPhysicsEntityNode* interface are found. To each of the found objects an unique ID number is assigned. ID number is used to resolve the references among the entities (the environment triggers reference the debris). After the IDs are assigned, each of the scene objects creates an instance of the *XMLSerializedEntityData* class. The *XMLSerializedEntityData* instances are put in the list and serialized into the XML file. The serialization is implemented in the *ExportEntities* method of the *FightRacePlugin* class.

When loading the track in the game the XML file is read and the entities are reconstructed from the instance of the *XMLSerializedEntityData* class. The references are reconstructed between the environment triggers and debris. This isn't valid solution for general references, but since currently there aren't other possible references, it is sufficient.

16.4.6 Simulation editor modes

Adding the instance of *FightRaceGame* to the plugin has come with an opportunity to integrate the game into the editor in such a way that the game can be played in the editor. Having such a

integration allows an artist to try the track without switching from editor to the game. As a result the game content should be created faster.

The instance of *FightRaceGame* class is initialized during the plugin initialization in the *InitializeGame* method. The *InitializeGame* class is called when the editor creates a render window (it is called from the *FightRacePluginRenderWindowCreated* method which handles the event raised when the render window is created). The render window of the editor is used as the target window for the game.

Note that the graphics engine is not initialized specifically for the game. Rather, the renderer for the game uses the same instance of the engine as the editor. The game also uses the same rendering scene that is used by the editor. The rendering scene is displayed in the editor's render window. As a result the draw called is called from the editor and not from the game. Therefore the plugin does only the update calls of the game and not the draw calls (see Figure 16.2).

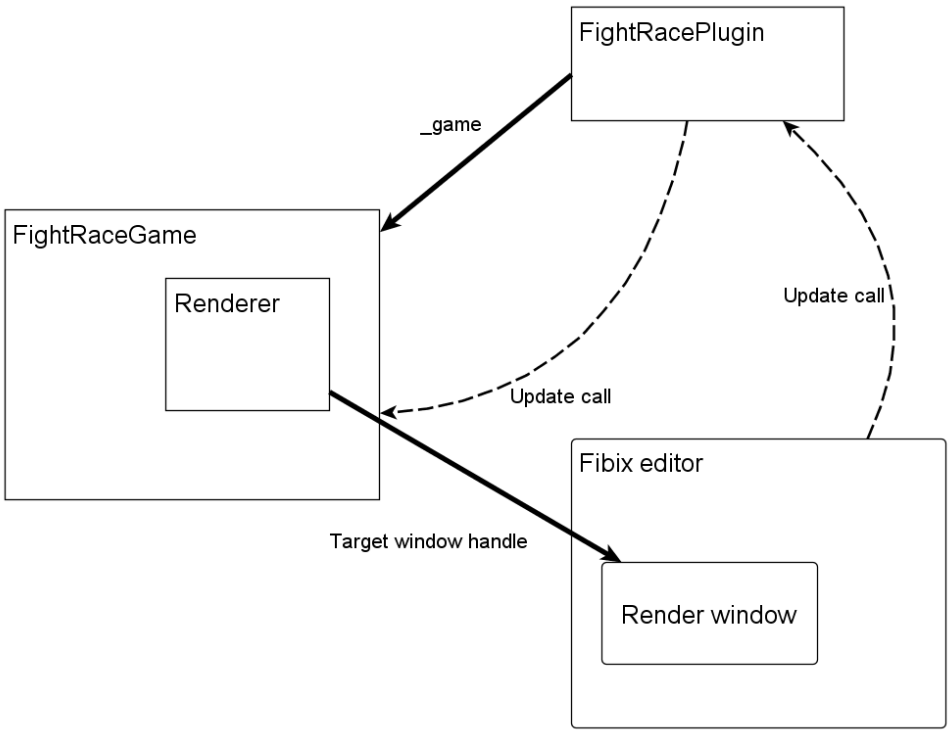

Figure 16.2: Context of the *FightRaceGame* class in the fibix editor. The game's renderer shares the graphics engine instance and the target window with the editor. Note that the plugin only does the update calls. The editor automatically calls draw on the graphics engine.

Task of running the game is implemented by a custom editor mode. When the mode is entered, the plugin prepares the game and starts the simulation. The simulation stops when the mode is exited. There are two different editor modes:

- "Run FightRace" mode - it starts new race session with AI opponents. The user of the editor can test the the behaviour of the AIs with current configuration of the waypoints and trackborders.

- "Simulate selected entities" mode - simulates the physical entities from the currently selected scene objects. When the mode is entered, the plugin creates new *SimulationWorld* and adds the selected entities to the world. Also the entity that represents the static geometry of the track is added (an instance of the *Environment* class - see Section 12.1). When the mode is exited, a dialog is showed that allows setting the world transform of the simulated entities to their current positions. The goal of this simulation is to find the world transforms for the debris, so they're placed on the surface and not hanging in the air.

Since the modes are similar they are implemented by the same class - (*FightRaceGameModeHandler*). The *FightRacePlugin* class creates two instance of this of the *FightRaceGameModeHandler* class which are registered as two different modes in the editor (see the *RegisterEditorModes* method).

Both of these modes need to represent the static collision geometry of the track (either using the *Environment* class in the "Simulate selected entities" mode or with the *Track* class in the "Run FightRace" mode). The collision shape of the track is formed by a geometry that is simpler than the one displayed in the editor. The name of the scene dump used for the collision geometry is saved in the scene using the special scene object implemented by the *TrackCollisionModelNode* class. This object is created when one of the above mentioned modes is entered and there is no *TrackCollisionModelNode* present in the scene (typically when the mode is entered for the first time). The *TrackCollisionModelNode* scene object can't be added explicitly. Therefore there can't be more than one instance of the *TrackCollisionModelNode* in the scene.

16.5 Export

The plugin is also capable of exporting the track to the game. The exporting is implemented in the *FightRacePlugin* class. The export is activated by the menu command (added in the *InitUI* method). The events from the export menu command are handled by the *ExportTrackHandler* method. The *ExportTrackHandler* method displays the exporting dialog (the *ExportTrackDialog* class). The dialog allows to choose path to the game's data folder, set the identifier and the name of the track and select which of the track files should be exported (for information about the files that define a track, see Section 12.2).

The export creates a new folder for the track in the folder "data\entities\tracks\". The name of the new folder is the same as the ID of the track. Inside this folder the track files (selected in the export dialog) are placed. The track files include:

- EDF file - deifnition of the track as a game entity.

- AI data files - data for AI navigation (both for the forward and reversed version of the track).

- Entities data file - data for creating game entities and including them to the *GameWorld* class with the track.
- Explicit collision geometry - scene dump with the collision geometry. It can be different (i.e. simpler) from the graphics geometry.
- Scene dump - scene dump with the graphical geometry of the track.
- Set of postprocesses - rendering postprocesses used for the track.
- Thumbnail - track's thumbnail image displayed in the menus.

Note that currently the AI data for reversed version of the track is **not** created. Also the the thumbnail image must be added manually. If the scene dump containing collision geometry is known (i.e. there is *TrackCollisionModelNode* present in the scene) it is also copied to the track folder. If it is unknown it is left to be later added manually.

Chapter 17

Test Cases

This chapter contains simple test cases and/or scenarios that test the functionality conditions of Fight Race Game / MotorDead.

Test cases are divided by topic and are in format:

1. Test summary (< ticket# >)

   More detailed test description, including key feature tested and pass/fail condition(s).

17.1 Menus and GUI

1. Background vehicle in menu screens is unable to move (319)

   In the (Quickrace) garage, browse owned vehicles using the carousel. Change vehicle several times. Test passes if vehicles in background do not move on the surface.

2. Applying settings has a 15sec timeout to possibly revert changes (301)

   Enter options menu, modify settings (especially resolution) and apply changes. This test is focused on the situation when the new settings are NOT displayed correctly. A confirm dialog box with timeout is displayed (although might not be visible). Test passes if the settings are kept if confirmed or changes are reverted if rejected or timed out.

3. Exhaust fumes move outwards from the exhaust pipe(s) (328)

   In any menu screen, rotate car (right click + drag) to see the rear end with exhaust pipes. Test passes if the exhaust fumes appear to fly out from the pipe.

4. The background car is equipped with weapons according to the user's settings

   Test fails if there is any inconsistency between the equipment of the user's last used car and the vehicle displayed in background

5. Dropdown list of a combobox is not higher or shorter than necessary.

Dropdown list is only as small as it can be to contain list items. It gains height as items are added. Dropdown scrolls if and only if the height (determined by number of items) reaches maximal height.

17.2 Shopping

1. Buying items works.

   When an item is purchased at the garage or shop, its price is subtracted from the user's cash and the item is added/upgraded/available.

2. User can only buy what they can afford.

   Every item (upgrade/weapon/car) has a displayed price. User has cash and it is displayed in each screen where are items for sale. Test fails if item can be purchased even though the user has less cash than the item costs OR user can not buy an item even though they have enough cash.

3. User can not have invalid cash.

   Test fails if the cash of any user is below zero or an invalid value (infinity, undefined, not a number).

4. Any owned weapon can be used for a race.

   Player can mount any owned weapon on any car. Not owned weapons can not be mounted. Only one weapon (of each kind - offensive and defensive) can be mounted at each time. Player can unmount weapons.

5. Owned items persist through restarting the game.

   Once an item is purchased in the shop or garage, it is saved in the user's profile. Test fails if a purchased item disappears / not owned item appears after restart.

6. Last used weapons persist.

   Last previously mounted weapons in a race are remembered for next race.

7. User can only own one vehicle of a kind

   Test fails if the user is able to buy the same vehicle again.

8. Vehicle's health can not exceed 100%.

   Test fails if user is able to repair the vehicle beyond 100% of health.

9. Weapons can be only bought once

   Each user has their own inventory of weapons. Those can be mounted on any vehicle. Test fails if user has to buy the same weapon again for a different vehicle

10. Upgrades are bought for each vehicle individually

    Test fails if upgrades are applied to another car without being bought separately

11. Program is resistent to texture filename extension absence.

    When using textures in GUI, filename may or may not contain the extension. Test passes if the program tries to load texture with the designated filename as if it contained the extension. If no such file exists/can be loaded, the program tries extensions for all supported formats.

12. Program is resistent to missing textures.

    When using textures in GUI, filename can be mistyped or the file can be missing. Test passes if any missing texure is replaced with a default, error texture. The only exception is when the error texture is missing. In that situation, an Exception is thrown.

17.3 Users, Profiles and Settings

1. Somebody must be logged in at all times

   - When starting the game for the first time / after update / after crash / profiles database missing, a new profile must be created and logged in.
   - The last profile in list can not be deleted.
   - User can not log off, only switch to another profile.

2. Every player has a default car

   After a new player profile is created, a default (bad as possible) car is already purchased in the garage.

3. Player can not select an unsupported screen resolution.

   The resolution combobox in options screen can only contain valid and supported resolutions.

4. Player name must be unique and not empty.

   Create a new user profile. Testing the situation when the player name is empty or collides with a yet existing profile. Test passes if user is asked to enter another name. Test fails if user accomplishes creating a profile with a duplicate or no name.

5. Fullscreen mode works.

   User can select the game to run in the fullscreen mode. Test passes if user can switch back and forth between windowed and fullscreen mode. Test fails if any inconvenience occurs and is not reverted after timeout.

6. Fullscreen on Multiple Screens

   Test passes if the game uses fullscreen mode on the screen where the system's cursor is.

7. Switching skidmarks on during gameplay (223)

   In gameplay, turn the skidmarks on. Test passes if skidmarks start appearing and the game does not crash. Including the situation when the skidmarks are turned on while the vehicle is sliding.

17.4 Gameplay

1. Intuitive vehicle behavior - not sliding sideways with handbrake on (331)

   accelerate, steer right (or left) and pull the handbrake. The test passes if vehicle starts to drift. Test fails if the vhicle rotates while stopped (no forward speed) or behaves otherwise unnaturally.

2. Metal plates in Docks are harmless (229)

   Drive over (either/all) plate piles (lying on railway crossings). Test passes if the vehicle is not damaged by driving over the plates.

3. Acceleration in race is joyful/not too slow (arcade impression) (305)

   Accelerate from complete stop - either at the start rack, or after stopping during the race. Test passes ifthe vehicle accelerates pleasantly fast.

   note: test for all available surfaces - tarmac, grass, sand, gravel, etc.

4. Rocket launcher has reasonable range (306)

   Start a quick race with a car equipped with rocket launcher (with ammo) and fire it on a straight and flat part of track. Test passes if the fired rocket impacts ground or an object in a reasonable distance

5. Nitro is either purchased or visibly marked as not owned. (328)

   Understand the gameplay HUD. Test passes if the user can quickly and clearly understand from HUD whether the vehicle has the nitro upgrade installed or not.

6. Vehicle is destroyed when health reaches zero.

   Collide your vehicle with scene OR let other players attack you. When health reaches zero, the car is considered destroyed. Test passes if these three events happen:

   - death animation (explosion) is shown
   - car is no longer conductible and stops
   - The race ends.

7. End of race (109)

   The race ends in these three situations: player dies, finishes the race or eliminates all opponents. Test passes if the game displays a table of results, with a comment correspondig to these cases:

   - winning the race (finishing first) OR eliminating all opponents → a congratulation
   - finishing 2nd or worse → a wish for better luck in next race
   - destroying vehicle → an encouragement for taking more care

   Either way, player recieves cash reward for playing (accumulated bonus for damage, race position bonus).

8. Vehicle must stay on the track

   All vehicles in race must not get to any illogical/unintended place, for example:

   - inside of a container
   - fall into water
   - drive/fall off the map
   - inside any decorative building (not intentionally drivethrough)
   - fall through the ground
   - get stuck in a lamppost/chimney/debris/wall of any kind
   - etc.

   Test passes if the player and AI controlled vehicles are unable to get "stuck". If a vehicle leaves the map and it is destroyed, the test also passes.

9. Reverse mode works (341)

   Select "Reverse mode" when starting quick race. This test is focused on robustness against missing reverse checkpoints for AI. Test passes if the game starts and the race is in the opposite direction, including the AI. Test fails if the game crashes, if the opponents are not moving, go the original direction or are otherwise confused, if the vehicles at the start rack are not rotated to head backwards.

10. All game features work in the Reverse mode.

    Test passes if all gameplay features are working properly. For example:

    - AI go the opposite direction
    - Laps are counted correctly
    - The lap is not a one-way - jumps are bidirectional, no obstacle is blocking the race flow

- Triggers boost in the right direction
- Weapons fire forward
- Checkpoints are passed in the right order

11. Game runs With no memory leakage

    This test is aimed at long time run. It is important to watch memory usage (in Windows Task Manager). Reasonable time to test: 2 hours

    - Game is running (inactive in menu)
    - Browsing the menu
    - Quick race starts, but user does not play
    - Player ignores lap counting and drives around level

List of figures

List of tables

Bibliography

[1] Bullet tutorials: Collision callbacks and triggers. http://bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Callbacks_and_Triggers#Contact_Information.

[2] Bullet tutorials: Hello world. http://bulletphysics.org/mediawiki-1.5.8/index.php/Hello_World.

[3] Bullet tutorials: Rigid bodies. http://bulletphysics.org/mediawiki-1.5.8/index.php/Rigid_Bodies.

[4] Bullet tutorials: Simulation tick callbacks. http://bulletphysics.org/mediawiki-1.5.8/index.php/Simulation_Tick_Callbacks.

[5] Bullet tutorials: Stepping the world. http://bulletphysics.org/mediawiki-1.5.8/index.php/Stepping_the_World.

[6] Sonory forum: How can i get the engine sound from the samples. http://www.sonory.org/forum/5-Engine-sound-in-games/6-How-can-I-get-the-engine-sound-from-the-samples.html.

[7] Xna simple gui. http://simplegui.codeplex.com/.

[8] J. Lander. The ocean spray in your face. *Game Developer*, July 1998:13–19.

[9] Tom McReynolds and David Blythe. 6.10 billboards. http://www.bluevoid.com/opengl/sig00/advanced00/notes/node74.html, November 2001.

[10] Ian Millington. *Game Engine Physics Development*. Elsevier, 2007.

Appendix A

DVD Contents

- `data` folder contains the game data, such as models, textures, sounds, etc.
- `documentation` folder contains the electronic version of this text, the doxygen-generated code reference and the user manual
- `install` folder contains the installer executable for the game
- `lib` folder contains the DLL libraries used by the application
- `src` folder contains the source code of the application
- `systemData` folder contains the data needed by the Fibix engine
- `tmp` folder contains miscellaneous files needed for creating the executable installer
- `tools` folder contains tools needed for creating the executable installer
- `README.txt` file with basic instructions. Please read it.
- `LICENSE.txt` file containing a legal notice.

Appendix B

List of sounds

Description	mono/stereo	Cue name	Looping
UI			
General button click sound	stereo	ButtonConfirm	once
Back / cancel button click	stereo	ButtonCancel	once
Carousel spin	stereo	CarouselSpin	once
Successful purchase button click	stereo	ButtonPurchase	once
Car repair button click	stereo	ButtonRepair	once
"Start race" button click	stereo	ButtonRace	once
Mouse hover over a GUI control	stereo	MouseHover	once
VEHICLE			
Engine sounds - # is number 1, 2, 3...	mono+stereo	Engine#	loop
Change transmission up	stereo	GearUp	once
Change transmission down	stereo	GearDown	once
Wheel rolling sound depending on the surface material	mono+stereo	Wheel<material>	loop
Wheel skid sound depending on the surface material	mono+stereo	Skid<material>	loop
Vehicle crashing into hard objects - number determines strength of the crash	mono	HitCar1	once
Contact of 2 objects with materials mat1 and mat2 (alphabetical order)	mono	Hit<mat1><mat2>	once
Vehicle grinding the surface of another vehicle	mono	SlideVehicle	loop
Vehicle grinding surface with material	mono	Slide<material>	loop
Active trigger activate	mono+stereo	TriggerActivate	once
Inactive trigger unsuccessful activate	mono+stereo	TriggerFail	once
Nitro start if available	mono+stereo	Nitro	

Nitro unsuccessful start if unavailable	stereo	NitroEmpty	once
Nitro usage sustain	mono+stereo		loop
Nitro depletion after usage	mono+stereo	NitroDepleted	once
Caltrops with wheel contact	mono+stereo	WheelCaltrops	once
Vehicle destroyed sound	mono+stereo	VehicleFail	once
Aerodynamic friction	stereo	AerodymicNoise	loop
WEAPONS			
Minigun shot	mono+stereo	MachineGunShoot	?
Minigun bullet impact on a material	mono	BulletImpact<material>	once
Empty minigun click	stereo	MachineGunEmpty	once
Rocket launcher shot	mono+stereo	RocketLauncherShoot	once
Rocket explosion	mono	RocketExplode	once
Empty rocket launcher click	stereo	RocketLauncherEmpty	once
Grenade launcher shot	stereo	GrenadeLauncherShoot	once
Grenade explosion	mono	GrenadeExplode	once
Empty grenade launcher click	stereo	GrenadeLauncherEmpty	once
Oil spiller - activate and splash on ground	mono+stereo	OilSpillerShoot	once
Empty oil spiller click	stereo	OilSpillerEmpty	once
Smoke screen - activate	mono+stereo	SmokeScreenShoot	once
Empty smoke screen click	stereo	SmokeScreenEmpty	once
Caltrops - shot and landing sound	mono+stereo	CaltropsShoot	once
Caltrops empty click	stereo	CaltropsEmpty	once
GAME			
Starting countdown sound (beep, beep, BEEEEP)	stereo	RaceCountdown	once
Successful checkpoint passed	stereo	CheckpointPassed	once
Lose race sound	stereo	RaceLost	once
Quit/destroyed sound	stereo	RaceQuit	once
Win race sound	stereo	RaceWon	once
ENVIRONMENT			
Seagulls ambient	stereo	DocksEnviro1	
Water splashing in the docks	stereo	DocksEnviro2	
Docks old rusty constructions squeaking	stereo	DocksEnviro3	
Ambient wind in docks	stereo	DocksEnviro4	

Table B.1: List of sound effects

Appendix C

List of particle effects

Identifier	Description	Type
	VEHICLE	
ExhaustFume	Fumes from the car exhaust	Continuous
ExhaustFire	Intensive, fiery exhaust fumes (when nitro is ON)	Continuous
EngineSmoke	Smoke from the engine when a car is damaged	Continuous
EngineFire	Fiery from the engine when a car is heavily damaged	Continuous
EngineExplode	An explosion of the car when it's destroyed	One-time
WheelRoll<Material>	Wheel effect for normal driving on a surface	Continuous
WheelSkid<Material>	Wheel effect for when the car is skidding	Continuous
VehicleSlide	Grinding of the vehicle on other surfaces	Continuous
	WEAPONS	
MachineGunShot	Minigun muzzle flash	One-time
MachineGunSmoke	Minigun overheat smoke	Continuous
BulletHit<Material>	Minigun bullet impact on material	One-time
BulletTrail	Minigun bullet mid-air	Continuous
RocketLauncherShot	Rocket launcher muzzle effect	One-time
RocketSmoke	Rocket trail - smoke	Continuous
RocketExplosion	Rocket explosion	One-time
GrenadeLauncherShot	Grenade launcher muzzle effect	One-time
GrenadeSmoke	Grenade trail - smoke	Continuous
GrenadeExplosion	Grenade explosion	One-time
SmokeScreenShot	Smoke screen effect, its visual has an actual gameplay role!	Continuous
OilSpillerShot	Oil spilling from the oil spiller weapon	One-time
SpikesSetterShot	Spikes falling out from the spike setter weapon	One-time
	ENVIRONMENT	
Hit<Material>	Collision of 2 objects (material of the softer object is used)	One-time
Explosion	Generic explosion - used for explosive debris so far	One-time
CheckpointMarker	Next checkpoint marker	Continuous

Appendix D

List of entity materials

Semantic ID	Name	Description
0	Default	Default material
1	Asphalt	Material for asphalt (and concrete)
2	Grass	Grass material
3	Vehicle	Vehicle material
4	WoodDebris	Material used for wooden debris
5	Walls	Material for (concrete and brick) walls
6	Metal	Metal material
7	Gravel	Gravel material
8	Soil	Material for dirt and similar soil materials
9	Pneu	Material for tyres-debris
10	Plastic	Plastic material

Table D.1: List of entity materials

Appendix E

Adding new entities

E.1 Introduction

This text explains the process of adding new entities to MotorDead. Specifically the task of adding new vehicles and tracks is discussed. In this section the terms used in the rest of the this text are defined.

Fibix editor
: Fibix Editor is the main tool for preparing data for the game. Although it is not possible to edit the 3D models in this editor, the model (prepared in other editor) can be opened and dumped (exported) to the game's internal format. Moreover, the custom plugin to the editor allows creation of new tracks.

Model dump
: A model in MotorDead is defined by the file containing the dump of the model. The dump is created in Fibix Editor by the menu command "Utils -> Dump -> Dump all entities". This command saves (i.e. dumps) the objects currently present in the editor scene to the file. The model dumps are located in the folder "Data\ModelDumps".

Scene dump
: A scene dump in MotorDead is very similar to a model dump. In addition to a model dump it contains also various scene properties (background, etc.). It is created by the menu command "Utils -> Dump -> Dump scene".

Blending model
: A blending model is defined by the two models with the same topology. Such model can be created in Fibix Editor using the menu command "Utils -> Model -> Convert two models to one deformation model". It creates new model that can be then loaded to the editor and dumped. The vehicles in MotorDead requires a blending model.

E.2 New vehicles

MotorDead allows data-driven adding of new vehicles to the game. A vehicle is defined in the EDF file located in the directory “data\entities\vehicles”. New vehicle is defined by adding new EDF file with a following format (each of the items corresponds with one line in the EDF file):

Identifier
Identifier of the vehicle. This value must be unique among all other entities. It must be different from the first line in all other EDF files. This include all EDF files in the “data\entities” folder and its sub folders, not only the vehicles in the “data\entities vehicles” folder.

Implementation class - “Vehicle”
This line is always “Vehicle” (because new vehicle is defined in this EDF file).

Displayed name
The name of the vehicle. The value is displayed in the game menus.

Description
A longer description of the vehicle. This line might be left empty since the description is currently not displayed anywhere in the game.

Price
Price of the vehicle in format: “$ = value"". The value must be positive.

Model
Name of the model dump with vehicle’s 3D model. This file must be located in the folder “data\ModelDumps”. The model that is dumped must contain all dummy points that are listed in the modelling conventions in Appendix F. An example of the vehicle’s models with the dummy points is showed in the figure E.1.

The dump must contain the blending model. Moreover, the vertices must have assigned the vertex color according to the color codes in Figure E.2. The vertex color is needed for the deformations of the vehicle’s model.

Collision model
Name of the file with vehicle’s collision model. The model must contain vertex color which defines the hitzones of the vehicle. The hitzones and the corresponding colors are defined in Figure E.2. The color codes are the same as for the vehicle’s model.

Wheel model
Name of the file with the model used for the wheels of the vehicle.

Center of mass
Offset of the vehicle’s center of mass. It controls the stability of the vehicle in turns. The higher (in the Y-axis) the offset is the more unstable the vehicle is.

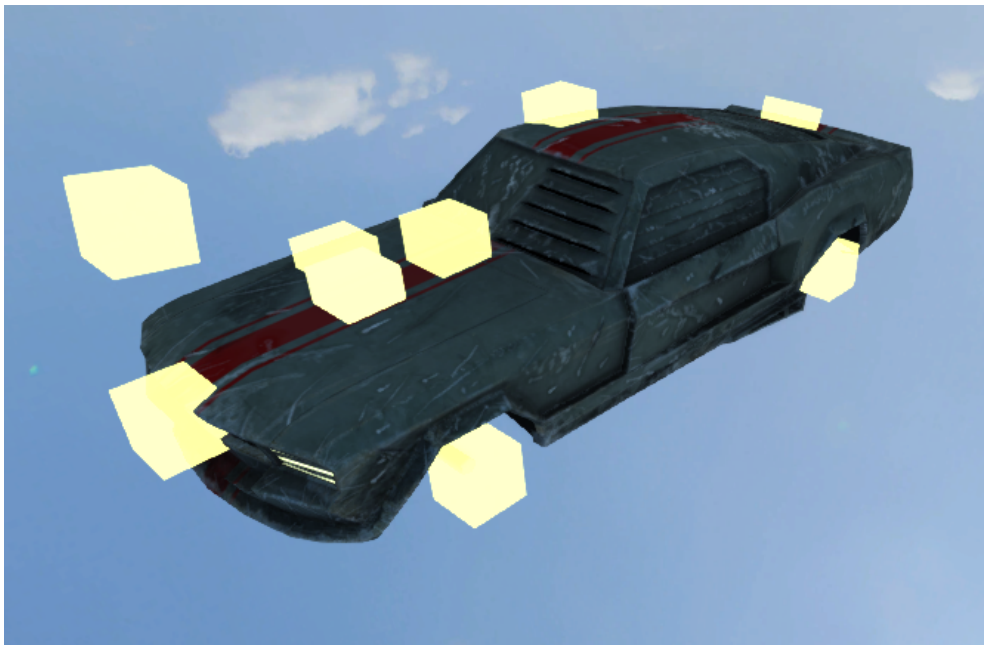

Figure E.1: Model of the vehicle with the dummy points.

Mass
Mass of the vehicle.

Physics height
Height of the box that approximates the vehicle's inertia.

Suspension rest length
The length of the suspension when zero suspension force is exerted on the chassis.

Suspension min length
The minimal length of the suspension. The suspension can't be compressed more than to this length.

Suspension max length
The maximal length of the suspension. It controls how low the wheels hang when the vehicle jumps in the air.

Stiffness
The stiffness of the suspension. Note that the value must be high enough to support the vehicle's mass. Recommended value is given by formula given in Equation E.1, but it can be tuned.

$$stiffness = \frac{0.25 \cdot 10 \cdot vehicleMass}{(restLenght - minLenght) \cdot 0.2} \tag{E.1}$$

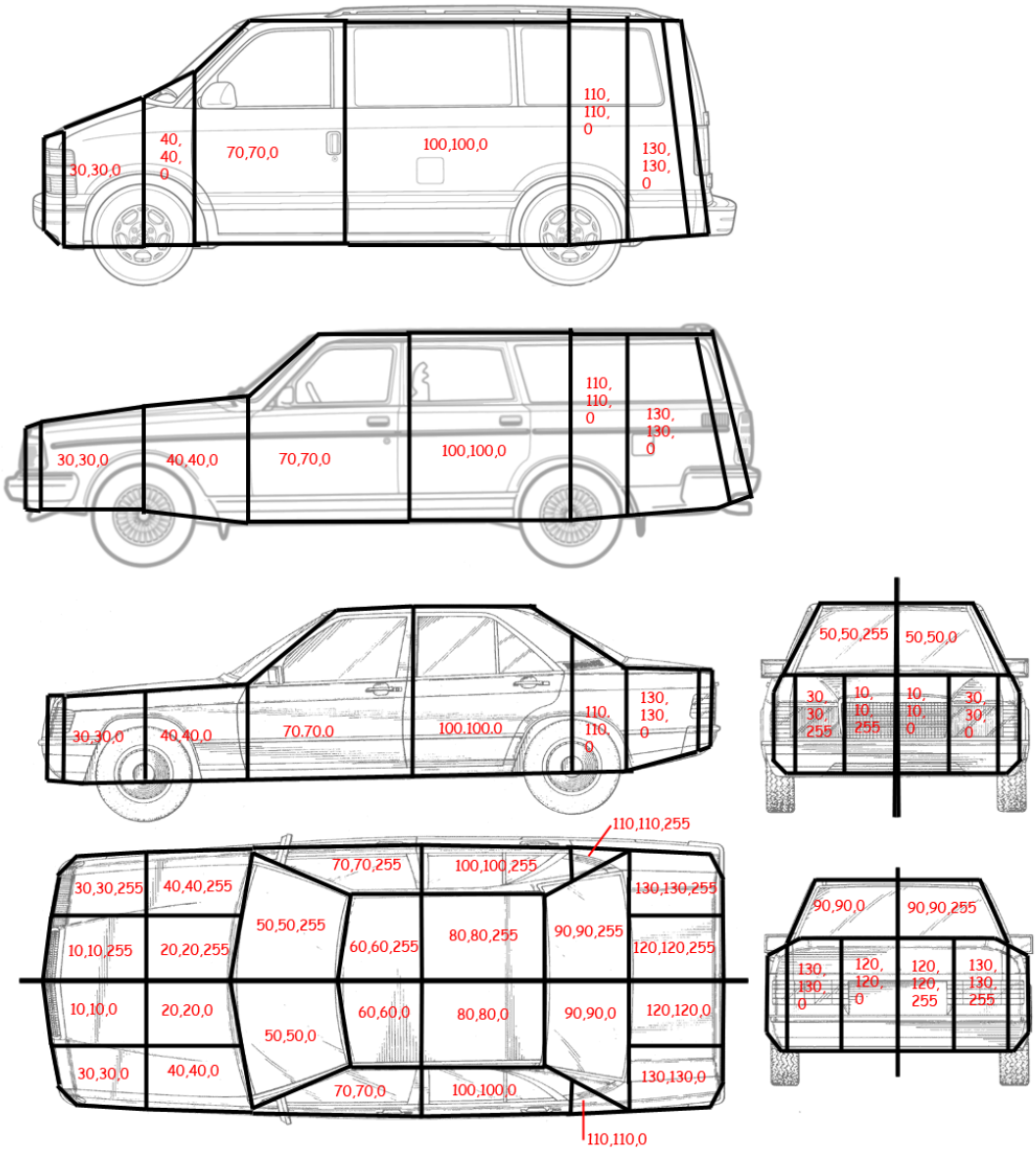

Figure E.2: Hitzones of the vehicle and their color codes.

Damping
: The damping of the suspension. The damping reduces the "bumpiness" of the suspension. The higher the value the more of the "bumpiness" is reduced.

Forward friction
: Coefficient of the forward friction of the tyres. Forward friction coefficient controls how well is the torque from the engine and from the brakes transferred "on the road". The higher the value the more torque can be transferred and vehicle accelerates and stops more rapidly.

Side friction
: Coefficient of the side friction of the tyres. It controls the handling. The higher the coefficient the better the handling.

Rear side friction multiplier
: Multiplier used to raise the side friction coefficient of the rear tyres. This helps the stability of the vehicle in high speeds. The value should be in the range $[1.0 - 1.1]$.

Wheel radius
: The radius of the vehicle's wheels. The value must correspond with the model of the wheels. Otherwise the wheels will be in in the air or sunk in the ground.

Brake power
: Controls the power of the brakes. The higher the value the better the brakes, but note that the effect of the brakes is limited by the forward friction. If the friction is low, the vehicle will slide and stop slowly.

Sliding power
: Controls how the vehicle drifts in turns. Use higher values for more over-steering.

Drive
: Which wheels transfer the torque from engine. There are 3 options: RWD, FWD, AWD.

Thumbnail
: Name of the file that is used as a thumbnail for the vehicle in the menu. The thumbnail file is located in the folder "data\textures\gui".

At last, a piece of advice for anyone who wants to create a new vehicle for MotorDead: **The best way to add new vehicle is to start with existing vehicle and modify it.**

E.3 New tracks

The tracks are located in the folder "data\entities\tracks". Each track has its own sub-folder. The sub-folder contains everything that is necessary for the track:

EDF file
Defines the track as the entity in the game. Remaining files from the folder are referenced in this file.

Thumbnail
Thumbnail displayed in the menu.

AI data file
Contains AI data for the track - waypoints and trackborders.

Reversed AI data file
Contains AI data for the reversed track - waypoints and trackborders.

Collision dump
Scene dump containing the collision geometry. The collision geometry should be simpler than the scene dump used for the graphical geometry. The materials in the collision dump should contain proper values of semantic IDs (semantic IDs are listed in Appendix D), otherwise the particle effects and sounds might work incorrectly.

Dump
Dump containing the 3D model of the scene (i.e. the graphical geometry). Note that semantic IDs of the materials should be set to proper values (semantic IDs are listed in Appendix D).

Entities data file
Contains data for entities included in the track - debris, static collision objects, triggers.

Postprocess
Contains settings of the rendering postprocesses for the track. Each track have its own set of postprocesses for better visual experience.

A new track can be created in Fibix Editor with FightRace plugin. The best first step is to model the geometry of the track. This is done by adding the models to the scene using the menu command “Create -> Create model"". Alongside the models the geometry must also contains all track’s dummy points that are listed in the modelling conventions in Appendix F. After the geometry is set, the AI data must be created and various entities (e.g debris, triggers) can be added to liven the track up.

E.3.1 AI data

AI data are necessary for the AI controlled vehicles to determine where the vehicle is and where it should go next. There are two types of the AI data:

- Waypoints - they define the driving line which should be followed by each AI driver. A waypoint object has the following properties:

ID

An unique identifier of the waypoint object. The value is assigned automatically by the editor.

Start

Waypoints form a closed circuit. There is always one waypoint that marks the start of the circuit. The value of the "Start" property is used to determine this waypoint. The waypoint with the value of the "Start" property set to *true*, is the first waypoint, that should be followed by the AI drivers after the race starts. In a track there must be only one waypoint marked as the Start.

Jump

Waypoints with this boolean property set to true lies before and on the ramp or any other place where vehicle can jump.

Checkpoint

Waypoints that should be marked as checkpoint have this boolean property set to true, otherwise false. Checkpoints are used to prevent the drivers from cheating by taking unfair shortcuts.

Previous

Holds a waypoint object that is before the current waypoint in the driving line. Because of the forks on the road there can be more instances of this property in a waypoint.

Next

Holds a waypoint object that is after current waypoint in the driving line. Because of the forks on the road there can be more instances of this property in a waypoint.

- Track borders - help the AI control to determine if a vehicle is on the track or not. Track borders lines are created from the track border objects which have the following properties (similar to the properties of the waypoints):

ID

An unique ID of the track border object. The value is assigned automatically by the editor.

Track side

Determines the side of the track border. Value *LEFT* is used if the track border line lies on the left side of the track, otherwise the value RIGHT is used.

Start

Track borders of a track always form a closed circuit. Both sides (left and right, see the "Track side" property) must have one track border object with the value property "Start" set to *true*. This track border should be the one that is the closest to the starting waypoint.

Previous
: Holds a track border object that is before current track border in border line. Each track border must have exactly one previous track border object.

Next
: Holds track border object that is after current track border in border line. Each track border must have exactly one next track border object.

Both the waypoints and the trackborders (i.e. items) and the connections among them can be added in the "FightRace edit" editor mode. New items are added by clicking at the point in the scene. Newly added item is automatically connected with the currently selected item. The newly added item is set as the next item that follows after the selected item. Consequently, the selected item is set as previous item of the newly added item. The connection between the existing items can be also added by selecting an item a "Ctrl+clicking" on the other one. The clicked item is set as the next for the selected item. When the "FightRace edit" mode is entered the waypoints can be added by default. Switching between adding waypoints and adding trackborders is done by clicking the mouse's middle button.

Both the waypoints and the trackborder must form a closed circuit. Therefore it is necessary to connect the last waypoint to the first and similarly for the trackborders. The trackborders define the section of the road where the vehicle should be. They limit the road from the two sides. Therefore there is a property of the trackborder that tells whether it is on the right or on the left side. The values of this property should be set correctly.

There is also way for loading previously saved waypoints via "Utils -> FightRace -> Load waypoints". The command removes the current waypoints from the scene, loads the waypoints from the file and adds them to the scene. The trackborders are kept unchanged. This command should be used to prepare AI data for the reversed version of the track.

E.3.2 Entities data

Entities data define the entities that are added to the game world with the track. The entities that can be added into the track include:

Debris
: Debris are objects like crates, barrels, traffic cones, etc. The debris are added by the menu command "Create -> Create debris"". The command displays a file dialog for choosing the model dump for the debris object. The file with model dump must be prepared before the debris object is created.

  When the model dump file is chosen, a new debris object is added to the scene. The debris object displays not only the chosen model but also a collision shape (it has green color). The default collision shape is a sphere. However it can be configured in properties of the debris object. The collision shape is given by the type of the shape (sphere, box, cylinder, cone and explicit) and the size. The explicit shape type is defined by a model dump (may be different from the graphical model of the debris object).

  The debris object has following properties that define its behaviour in the game:

- Mass - ass of the debris object.
- Friction - friction coefficient of the debris object.
- Semantic ID - defines the logical material assigned to the debris object (see list of materials in Appendix D).
- Is destroyable - defines whether the debris object has health and is damaged by the collisions and weapon hits. This option **experimental** only, it is better to leave it set to false.
- Is explosive - defines whether the debris object explodes when it health reaches zero. If the debris object is explosive, the three parameters defines the properties of the explosion - the radius, the damage it causes to the object within the radius, the impulse that is applied to the objects within the radius. This option **experimental** only, it is better to leave it set to false.

For fine tuning of the positions of the debris objects the "Simulate selected entities" editor mode can be used. When it is entered it start the physics simulation for the selected debris objects (they are influenced by the gravitational force and collisions). When the mode is exited, the dialog is showed, for choosing whether the debris objects should keep the current position (i.e. the positions resulting from the physics simulation).

Static collision object Static collision objects are invisible colliders. They should have been used to approximate the complex geometry of some track object. However, this is done by the collision scene dump. Still, the static collision objects can be used as resetting objects. When a vehicle collides with a resetting static collision object it is reset back on the track. The resetting object should be added to part of the track, where the vehicle should not be able to get. To turn on the resetting, the “ResetsVehicle” property must be set to true. New static collision object is added by the menu command “Create -> Create static object”.

Triggers Triggers are activated by the vehicles running over them. The triggers are added in the “Create triggers” editor mode. In the mode new trigger is added by right mouse button click at the position where the trigger should be created. After the button is clicked, a dialog is showed for choosing the type of the trigger. There are following types of the triggers:

- Speed-up trigger - boost speed of the vehicle.
- Repair trigger - repairs the vehicles
- Ammo trigger - recharges ammo of the vehicle’s weapons.
- Environment trigger - trigger only once. Activates given debris object, that starts moving after activation. Can be used for creating environment traps. The debris object, that will be activated by the trigger, is set using the “Linked entity” property.

The trigger contains the collision shape that is sued to detect vehicles, that activate the trigger. It is displayed using the green color. It can be configured the same way as for debris object.

E.3.3 Export

The track is exported using the menu command “Utils -> FightRace -> Export track to FightRace”. The command creates the folder for the track and the EDF file. The scene in the editor is used as the track 3D model and the currently set postoprocesses are also saved. The waypoints and trackborders are exported as the AI data. The entities are exported too.

The collision dump is exported only if the “Run FightRace"" or the “Simulate selected entities"" mode has been entered previously. Note that the reversed version of AI data must be added explicitly - the file must be added to the track’s folder and the option to the track’s EDF file. For the tracks the same is true as for the vehicles: **the best way is to look at the existing working tracks and modify them**.

Appendix F

Modeling conventions

F.1 Introduction

This chapter provides the conventions used in the game content of MotorDead. The chapter is structured as the lists of conventions, rather then the continuous text. Naturally, it is necessary to follow all of the conventions to create content that is compatible with the game.

F.2 Meshes

Real scale
: There is no scaling in the game, therefore the models should be provided in their real scales. Units used in the game are meters. Therefore one unit in the model is equal to one meter in the game.

Axis alignment
: The left-to-right direction must be aligned with the positive X axis, the bottom- to-up direction with the positive Y axis and the back-to-front direction with the positive Z axis.

Pivot on objects
: On models for the objects that will be physically simulated (cars, grenades, barrels, etc.) the position of the pivot must be aligned with the position of the center of mass. It is recommended to follow this particular convention in all models, if not stated otherwise.

Pivot on attached models (wheels and weapons)
: On wheels and weapons, position of the pivot must be in the point of attachment to a vehicle. The models are attached to the points given by the vehicle's dummy points. All wheel models must be oriented in the same way (i.e. the model can be used directly for the right wheels and will be rotated for the left wheels automatically).

Triangle count (vehicles)
: The number of vehicle in the race is limited to seven. As a result there are at most eight

vehicle visible on the screen. It is important to keep the triangle count below 30k-40k per vehicle.

Duplicated vertices
Duplicated vertices should be removed by the weld operation.

F.3 Textures

All textures should be in the .dds file format with DXT1 compression. If the texture contains an alpha channel, use DXT5 compression. Note that both the width and the height of the texture have to be multiples of 4.

F.4 Materials

Semantic ID
Semantic ID on the materials should be set to proper value that corresponds with the appearance of the material. If the value is set incorrectly there will be inconsistency between the appearance and the additional effects (particle, skid marks, bullet hits, etc.). Available values of semantic IDs are listed in Appendix D.

F.5 Dummy points

Position Dummy points are used to determine various important points in the model. The dummy points must be places carefully at the correct position.

Name The dummy points of a model are found according to their names. The name of a dummy point must end with the ";" character (i.e. semicolon).

F.6 Vehicles

Deformation model
Vehicle body is given by one render object. It is assumed that this object is able to blend the fully new and the fully damaged state. In the game this render object is found according to its name: "DeformationBody".

The blending model can be created in Fibix editor with the "Utils -> Model -> Convert two models to one deformation model" menu command . The blending model is created from the two models (one for the fully new state and the other for the fully damaged state) with the same topology. The vertices in both models must be painted with vertex color according to Figure F.1.

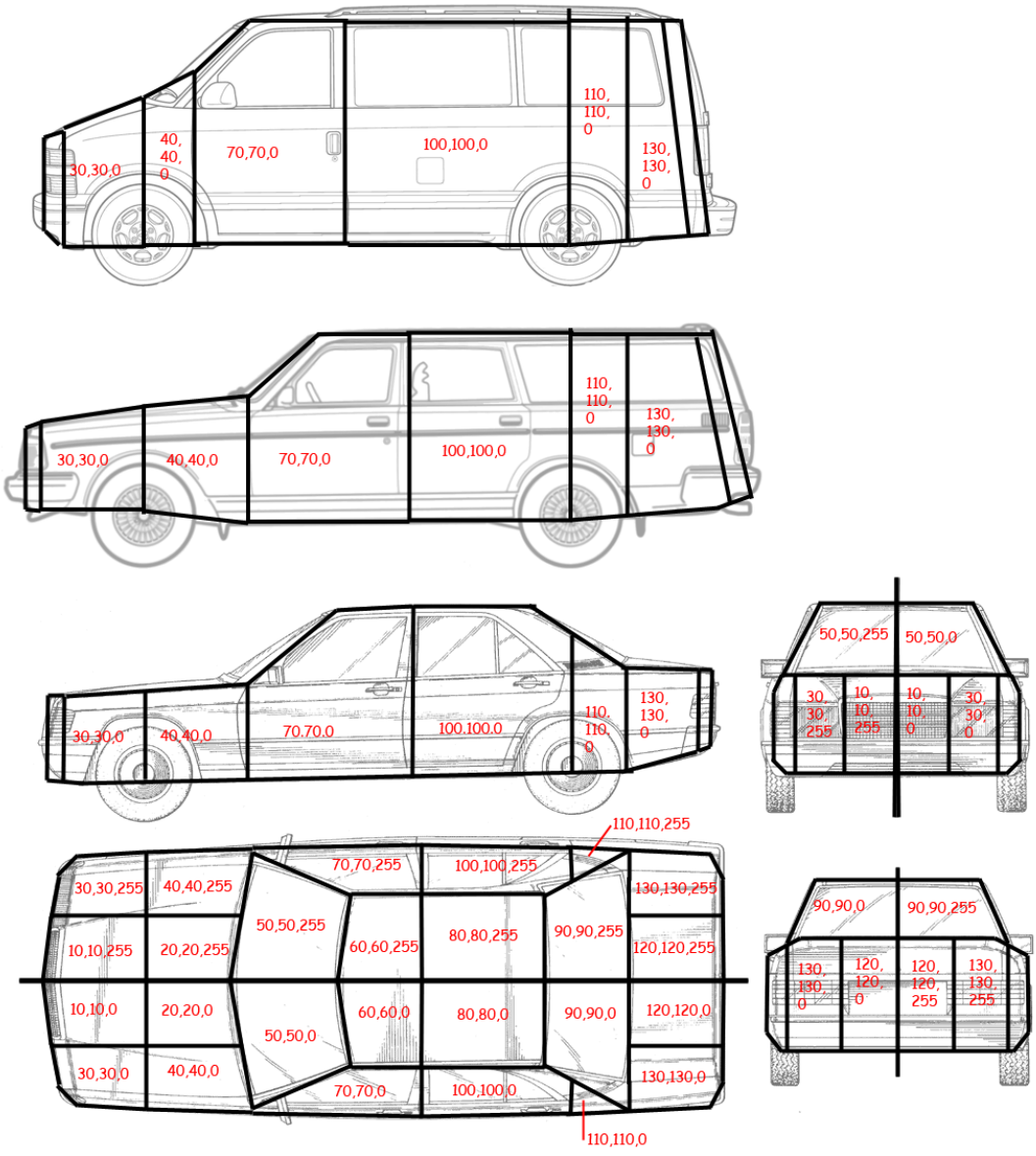

Figure F.1: Hitzones of the vehicle and their color codes.

Collision model Collision model for a vehicle contains 26 hitzones. The vertices of the model must be painted with the vertex color according to Figure F.1. Each hitzone should be convex. A collision model for a vehicle must be much simpler than the model used for the appearance. The number of vertices should not exceed 30 per hitzone.

Dummy points The scene with vehicle (and consequently the model dump) of a vehicle must contain following dummy points:

- Wheels positions:
  "Wheel:FL;" - front left wheel.
  "Wheel:FR;" - front right wheel.
  "Wheel:RL;" - rear left wheel.
  "Wheel:RR;" - rear right wheel.

- Lights positions:
  "Light:FL;" - left headlight.
  "Light:FR;" - right headlight.
  "Light:RL;" - left tail light.
  "Light:RR;" - right tail light.

- Weapons positions:
  "WeaponOffensive;" - default position of the offensive weapon.
  "WeaponDefensive;" - default position of the defensive weapon.
  "WeaponOffensive:WeaponID;" - position of the specific offensive weapon (non-compulsory dummy point).
  "WeaponDefensive:WeaponID;" - position of the specific defensive weapon (non-compulsory dummy point).

- Cameras:
  "Camera:FrontBumper;" - position of the camera on the front bumper.
  "Camera:Hood;" - position of the camera on the hood.
  "Camera:LookBack;" - position of the back view camera.

- Particle effects: "LightDamageParticles;" - position of the particle effect activated when vehicle's health falls below 50. There can be multiple dummy points with this name.

  "HeavyDamageParticles;" - position of the particle effect activated when vehicle's health falls below 10. There can be multiple dummy points with this name.

"VehicleKilledParticles;" - position of the particle effect that is activated when the vehicle is killed (i.e. its health reaches zero).

"ExhaustFumesParticles;" - position where the particle effect for exhaust fumes will be added. It is possible to create multiple dummies of this type.

F.7 Weapon models

Dummy points

Each model for a weapon must contain following dummy points

- Muzzle position (offensive weapons only):
  "Muzzle;" - the position and the direction from where the weapon fires.

F.8 Tracks

Dummy points

A scene for the track must contains following dummy points:

- Starting positions:
  "StartPosition:1;" - starting position for the first vehicle on the grid
  "StartPosition:2;" - starting position for the second vehicle on the grid
  ...
  "StartPosition:7;" - starting position for the seventh vehicle on the grid

Collision model

A track has defined collision model that is different from the graphical appearance. This model must be much simpler than the graphical model. It is never displayed in the game, therefore approximate geometry is appropriate. The semantic IDs must be assigned in the collision as well. Available values of semantic IDs are listed in Appendix D.

F.9 Bullet holes

A bullet hole is created when a bullet hits an object in the scene. A bullet hole is represented by a decal which is added to the model of the hit object. Bullet holes are created only if the hit object has assigned an entity material on which the bullet holes should be created. The entity material also define which model is used for the bullet holes (on the material). The model dump for the material's bullet holes must contain only one render entity - the decal for the bullet holes.