Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Karel J a l o v e c
Study programme:	Software Engineering and Management
Specialisation:	Intelligent Systems
Title of Bachelor Project:	Physical Simulation of Ground Vehicles

Guidelines:

1. Study physical simulation engine JBullet and multiagent platform Alite.
2. Develop physical simulation support for Alite.
3. Design and implement physical simulation entities for various types of ground vehicles (e.g. car, motorcycle, truck, etc.).
4. Design and implement interfaces for interaction with entities (actuators and sensors).
5. Implement experimental environment and demonstrate simulation capabilities of particular vehicles.

Bibliography/Sources: Wil be provided by the supervisor.

Bachelor Project Supervisor: Ing. Jiří Vokřínek

Valid until: the end of the winter semester of academic year 2011/2012

[illegible]

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
-4-
L.S.

[illegible]

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Boris Šimák, CSc.
Dean

Prague, February 2, 2011

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Karel J a l o v e c
Studijní program:	Softwarové technologie a management
Obor:	Inteligentní systémy
Název tématu:	Fyzikální simulace pozemních vozidel

Pokyny pro vypracování:

1. Seznamte se s prostředím pro simulaci fyziky JBullet a multiagentní platformou Alite.
2. Vytvořte podporu pro simulaci fyzikálního prostředí v Alite.
3. Navrhněte a implementujte fyzikální simulační entity pro různé druhy pozemních vozidel (např. automobil, motocykl, nákladní vůz, apod.).
4. Navrhněte a implementujte rozhraní pro interakci s entitami (aktuátory a sensory).
5. Implementujte experimentální prostředí a demonstrujte simulaci jednotlivých vozidel.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Jiří Vokřínek

Platnost zadání: do konce zimního semestru 2011/2012

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

[illegible]

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

[illegible]

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 2. 2. 2011

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Project

Physical simulation of ground vehicles

Karel Jalovec

Supervisor: Ing. Jiří Vokřínek

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Engineering

May 26, 2011

Acknowledgements

I would gladly like to thank my supervisor Ing. Jiří Vokřínek and Ing. Antonín Komenda for their professional support and valuable ideas. I would also like to thank my friends Jan Havlíček and Martin Kinčl for their moral support.

Declaration

Abstract

This work focuses on physical behavior of ground vehicles in multiagent simulation. I have tried to implement common interface, which can be used for connecting to other simulations providing physical behavior of simulated ground vehicles. Resulting program demonstrates, how already existing technologies can be used to create physical simulation environment.

Abstrakt

Svou práci jsem zaměřil na multiagentní fyzikální simulaci pozemních vozidel. Pokusil jsem se vytvořit aplikační rozhraní tak, aby bylo možné projekt využít i v jiných simulacích, kterým by poskytoval fyzikální chování pozemních vozidel. Výsledný program demonstruje, jak je možné využít již existujících technologií k vytvoření fyzikálního simulačního prostředí.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I would like to describe used technologies first. It will create knowledge background for my work and show why I am interested in this project. Firstly, I would like to describe the jBullet physical library[4], because I used this toolkit the most in my project. Next, I would like to discuss raycast vehicle model. I used this simplified vehicle model for all vehicles in my simulation. It is commonly used vehicle model for significant amount of physical based simulations and computer games. Then, I would like to describe the Alite[1] software toolkit. This toolkit is used in both projects - I use it in my own project as well as it is used in Highway project[6]. The Highway project is the last one I would like to describe in this chapter. I tried to extend the Highway project using technologies mentioned above.

1.1 jBullet physical library

The JBullet[4] physics library is Java port of the Bullet Physics Library[2]. The Bullet physics library is professional open source collision detection, rigid body and soft body dynamics library. Main task of a physics engine is to perform collision detection, resolve collisions and other constraints, and provide the updated world transform (world transform of the center of the mass for rigid bodies, transformed vertices for soft bodies) for all the objects. The reason I chose jBullet is that it is implemented in Java programming language. This is big advantage, because all the projects I work with are implemented in Java programming language as well. Next reason is that jBullet provides very realistic and easy to use implementation of physical behavior. Main features and advantages are:

- Open source C++ code under Zlib license and free for any commercial use on all platforms including PLAYSTATION 3, XBox 360, Wii, PC, Linux, Mac OSX and iPhone

- Discrete and continuous collision detection including ray and convex sweep test. Collision shapes include concave and convex meshes and all basic primitives

- Fast and stable rigid body dynamics constraint solver, vehicle dynamica, character controller and slider, hinge, generic 6DOF and cone twist constraint for ragdolls

- Soft Body dynamics for cloth, rope and deformable volumes with two-way interaction with rigid bodies, including constraint support
- Maya Dynamica plugin, Blender integration, COLLADA physics import/export support

Bullet has been designed to be customizable and modular (see main bullet modules on figure 1.1). The developer can

- use only the collision detection component
- use the rigid body dynamics component without soft body dynamics component
- use only small parts of a the library and extend the library in many ways
- choose to use a single precision or double precision version of the library
- use a custom memory allocator, hook up own performance profiler or debug drawer

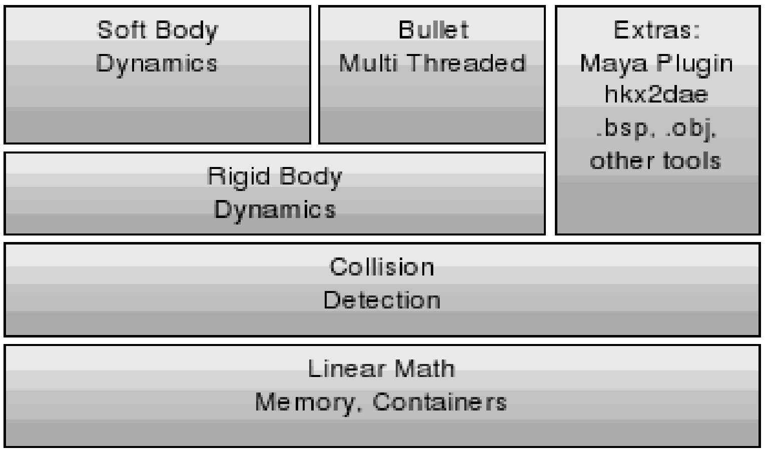

Soft Body Dynamics
Bullet Multi Threaded
Extras: Maya Plugin hkx2dae .bsp, .obj, other tools
Rigid Body Dynamics
Collision Detection
Linear Math Memory, Containers

Figure 1.1: Schema of Bullet Physics Library

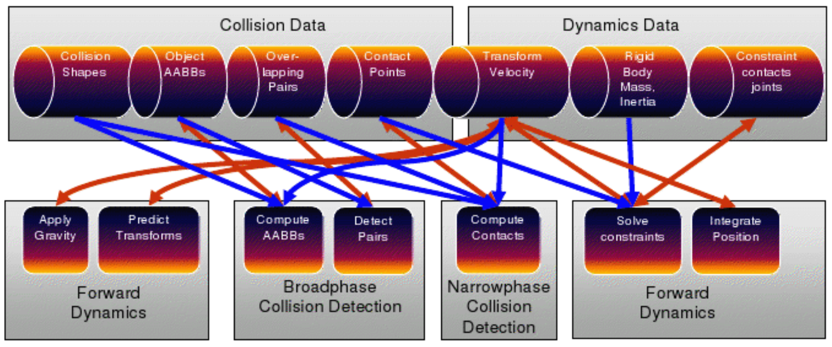


Figure 1.2: Rigid body physics pipeline

The diagram above (see figure 1.2) shows the most important data structures and computation stages in the Bullet physics pipeline. This pipeline is executed from left to right, starting by applying gravity, and ending by position integration, updating the world transform. The entire physics pipeline computation and its data structures are represented in Bullet by a dynamics world.

1.1.1 Basic data types and math library

The basic data types are located in jBullet/src/com/BulletPhysics/LinearMath. There are four main data types:

- Scalar

A Scalar is a posh word for a floating point number. In order to allow to compile the library in single floating point precision and double precision, the Scalar data type is used throughout the library. By default, Scalar is defined as a float.

- Vector3

3D positions and vectors can be represented using Vector3. Vector3 has 3 scalar x,y,z components. It has, however, a 4th unused w component for alignment and SIMD compatibility reasons. Many operations can be performed on a Vector3, such as add subtract and taking the length of a vector.

- Quaternion and Matrix3

3D orientations and rotations can be represented using either Quaternion or Matrix3.

- Transform

Transform is a combination of a position and an orientation. It can be used to transform points and vectors from one coordinate space into the other. No scaling or shearing is allowed. As I talk about positions, rotations and transformations, it would be useful to mention the coordinate system that is used within jBullet physical library. jBullet physical library uses a right-handed that differs from commonly used cartesian coordinate system (see figure 1.3).

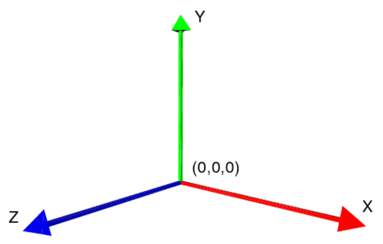


Figure 1.3: Right handed coordinate system

1.1.2 Collision detection

The collision detection provides algorithms and acceleration structures for closest point (distance and penetration) queries as well as ray and convex sweep tests. The broadphase collision detection provides acceleration structure to quickly reject pairs of objects based on axis aligned bounding box (AABB) overlap. Several different broadphase acceleration structures are available. The broadphase adds and removes overlapping pairs from a pair cache. A collision dispatcher iterates over each pair, searches for a matching collision algorithm based on the types of objects involved and executes the collision algorithm computing contact points.

1.1.3 Collision shapes

Bullet supports a large variety of different collision shapes, and it is possible to even add user-defined collision shapes. For best performance and quality it is important to choose the collision shape that best suits given problem.

Convex primitives Most primitive shapes are centered around the origin of their local coordinate frame.

- BoxShape - Box defined by the half extents (half length) of its sides
- ConeShape - Cone around the Y axis. Also ConeShapeX/Z.
- SphereShape - Sphere defined by its radius
- CylinderShape - Cylinder around the Y axis. Also CylinderShapeX/Z

Compound shapes Multiple convex shapes can be combined into a composite or compound shape, using the CompoundShape. This is a concave shape made out of convex sub parts, called child shapes. Each child shape has its own local offset transform, relative to the CompoundShape.

Convex hull shapes Bullet supports several ways to represent a convex triangle meshes. The easiest way is to create a ConvexHullShape and pass in an array of vertices. In some cases the graphics mesh contains too many vertices to be used directly as ConvexHullShape. In that case, good way is to reduce the number of vertices.

Concave triangle meshes For static world environment, a very efficient way to represent static triangle meshes is to use a BvhTriangleMeshShape. This collision shape builds an internal acceleration structure from a TriangleMesh or StridingMeshInterface. Instead of building the tree at run-time, it is also possible to serialize the binary tree to disc.

1.1.4 Collision matrix

For each pair of shape types the Bullet will dispatch a certain collision algorithm (see collision matrix table 1.1), by using the dispatcher. By default, the entire matrix is filled with the following algorithms. Note that Convex represents convex polyhedron, cylinder, cone and capsule and other GJK[7] compatible primitives. GJK stands for Gilbert, Johnson and

Keerthi, the people behind this convex distance calculation algorithm. It is combined with EPA[3] for penetration depth calculation. EPA stands for Expanding Polythope Algorithm by Gino van den Bergen. Bullet has its own free implementation of GJK and EPA.

	box	sphere	convex	compound	triangle mesh
box	BB	SB	gjk	C	CC
shpere	SB	SS	gjk	C	CC
convex	gjk	gjk	gjk	C	CC
compound	C	C	C	C	C
triangle mesh	CC	CC	CC	C	gimpact

Table 1.1: Collision matrix

Abbreviation	Algorithm
BB	boxbox
SB	spherebox
SS	spheresphere
C	compound
CC	concaveconvex

Table 1.2: Description of collision algorithms abbreviations

The user can register a custom collision detection algorithm and override any entry in this collision matrix by using the Dispatcher.registerCollisionAlgorithm.
Some examples of bullet physics engine can be seen on figure 1.4, figure 1.5 and figure 1.6.

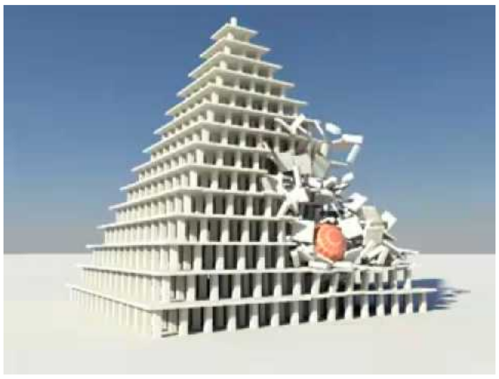


Figure 1.4: Example of bullet physics engine 1

Figure 1.5: Example of bullet physics engine 2

Figure 1.6: Example of bullet physics engine 3

1.2 Raycast vehicle

The ray cast vehicle[5] is simplified vehicle model. It consists works by casting a ray for each wheel. Using the ray's intersection point, suspension length can be calculated, and hence the suspension force. The suspension force is applied to the chassis, keeping it from hitting the ground. In effect, the vehicle chassis "floats" along on the rays. The friction force is calculated for each wheel where the ray contacts the ground. This is applied as a sideways and forwards force. The rays should originate inside the vehicle chassis's btCollisionShape. If the start point of the suspension ray is outside the world, then the ray may never find a ground contact point, and the vehicle will get stuck. Roll influence effectively lowers the vehicles center of mass, reducing the chance of the vehicle rolling over.

Simulation of the suspension is provided by the spring force plus a damping force. The damping force stops the car from bouncing forever. There are coefficients for suspension stiffness, suspension rest length and two coefficients for damping: one for spring compression, and one for spring relaxation.

- suspensionStiffness
- suspensionDamping

- suspensionCompression
- suspensionRestLength

In a real vehicle, the compression damping is set much lower than the relaxation damping. This means, when the vehicle hits a bump, it won't be transmitted to the chassis, resulting in a smooth ride. Bullet's suspension has one major drawback: when the spring is fully compressed, it cannot provide enough force to keep the vehicle's chassis off the ground. In a real vehicle, the spring will hit a bumper, keeping the wheel away from the chassis. To simulate this, the constraint is used.

A suspension constraint has two parts: the suspension limits, and the suspension force. The suspension force is responsible for the spring force applied by suspension, and the suspension limits stops the wheels penetrating the chassis, or flying off the vehicle.

The friction model in Bullet is implemented as separate impulses applied to each wheel. This means that it is possible for a single wheel to counteract all horizontal motion of the chassis, and for multiple wheels to add additional velocity, resulting in oscillation or jitter of the vehicle. The solution to this is to create a friction constraint model. Expressing the friction as constraints allows Bullets constraint solver to perfectly balance the friction on each wheel, to counteract any sideways velocity.

1.3 Alite

Alite software toolkit (see complete schema of Alite toolkit on figure 1.7) is helping with particular implementation steps during construction of multi-agent simulations and multi-agent systems in general. The goals of the toolkit are to provide highly modular, variable, and open set of functionalities defined by clear and simple API. Highway project (see section 1.4) is implemented using Alite[1] software toolkit as well.

The toolkit does not serve as a pre-designed framework for a complex purpose, it rather associates number of highly refined functional elements, which can be variably combined and extended into a wide spectrum of possible systems. It consists of these parts:

Creator is responsible for initialization of whole environment, entities, visualization etc.

Event Queue can be used for any queued processing of discrete-time events. The API provides methods for adding and handling of events. Additionally the events are prioritized by their time and processed in this particular order. Events with same timestamps are treated as simultaneous and the particular ordering of their processing has to be considered as random (or simultaneous if they are processed concurrently).

Entity is an object with a name encapsulating bounded phenomenon. The purpose or usage of entities is not restricted. Entities can represent agents, simulation embodiments, parts of other entities, information mediating objects and so on.

Capability Register associates various capabilities with identities (e.g., entities). Both the identity and the capability is represented by a String. The main functionality of a register is to filter all identities by one capability.

Environment in Alite consist of three main building blocks: state storages, actuators, and sensors.

- A storage holds data structures representing a state of the elements in the environment (position of a person, velocity of a car, etc.).
- Actuators are responsible for changes of the data in storages. Actuators can be chained, and should form various levels of abstraction upon the environment state described in the storages. Actuators can use sensors to form shortened control loops (sensor -> actuator -> storage -> sensor -> ...), which can represent implicit mechanics of the environment.
- Sensors mediate information of the environment to the decision making algorithms (entities, agents, actions, other sensors). The sensors can be also chained and should form abstraction layers of the received information. There are two types of sensors we consider: push and pull. The push sensors are based on callbacks implemented in the scope of the decision making and the callbacks are invoked by the sensor.

Communication package serves as a tool for inter-agent messaging. Each agent has to implement two basic features – communicator and at least one communication channel.

Spatial maneuver path planner using multi-granular discretization of a continuous planning space.

Visualizations use various technologies to display the simulation

- vis lite 2D visualizer using Java2D backend, one drawing window and canvas, hierarchically organized set of layers drawing on the canvas separated sets of similar visual elements (circles, lines, points, sprites, ...), providing zooming and panning of the displayed area.
- Google Earth connector

1.4 Highway

Project Highway[6] focuses on the multiagent simulation in highway traffic. Multiagent approach offers powerful solution to create simulations of such complex systems as fully automated highway system is. The platform derives benefits from many of its advantages, such as cooperation and interaction abilities of autonomous agents. It is also fully distributable on more computers, which allows of testing more sophisticated scenarios on large highway infrastructures. A part of this thesis is to design and implement the simulation world and creation realistic highway infrastructure (see screenshot of simulation on figure 1.8), which is the main environment for testing various scenarios of non-cooperative agents. The main goal is to design and implement non-cooperative agents with physical model approached to real world physical model with focused on collision avoidance, smooth drive and efficient capacity of highway.

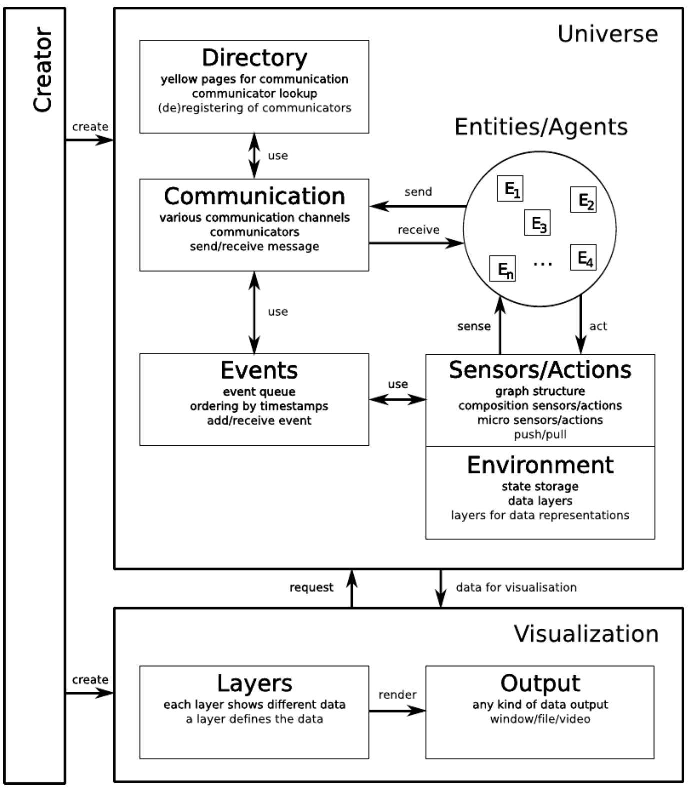


Figure 1.7: Alite toolkit overview

Why do I talk about the Highway project? Original Highway project implemented physical behavior of the vehicles. This implementation was not as good as it could be. This is the reason I am trying to extend this project with the implementation of more realistic vehicle models. But not only vehicles behave more realistic. Environment and all the other objects behave more realistic as well. In other words, I am trying to add to the simulation more realistic implementation of the real world using existing technologies (see sections 1.1, 1.2 and 1.3).

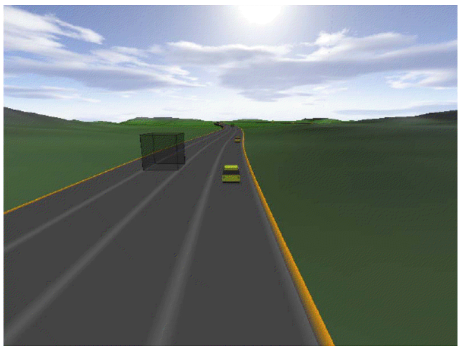


Figure 1.8: Screenshot from Highway project

Chapter 2

Problem description and goals

In this chapter, I would like to write about goals of my work. First of all, I will write about interface of the application. This is one of main goals I have to achieve. Provide simple application interface for other simulations. Simplicity of the interface is vital to my project. Next, I would like to discuss simulation entities. A simulation with only one kind of vehicle is quite odd. So, I have to implement more than one type of vehicle. I am limited by one restriction: the interface. All the vehicles must share one common interface, so they can be handled together in one way. As next goal, I want to mention, is independence on visualization. The simulation must be independent of the visualization part. It can be run without graphics. This is used mainly for achieving good simulation performance. Running the visualization is good for presenting results of experiments. The last goal is to make the project as much modular as it can be made. This is vital for exchanging parts of the project in case of unsuitability of some of the parts.

Interface : One of main goals is to create physical module with suitable interface. The interface should be as simple as possible. It should use the jBullet physical library for simulating physical behavior of simulated environment and it should provide necessary functions for other applications, which are intended to use this physical module.

Entities : Next goal is to create various simulation entities such as trucks, motorbikes, cars etc. Physical behavior depends on settings of the entity parameters. For example, time necessary to stop a car depends on its weight and braking capability. Another task is to ensure autonomous behavior of the entities. The simulation entities must share common interface, so they can be handled the same way.

Visualization : Visualization is provided by the OpenGL. One necessary assumption is that the visualization should be independent of the simulation. I managed that by multithreading. Multithreading brings another problem. Data necessary for visualization should not be acquired directly from the simulation. After each simulation step, data is stored in an corresponding entity object. After that, the visualization can use this data for displaying entities to the user. This way ensures independence between the simulation and the visualization and eliminates need for synchronization at the same time.

Alite support : Next vital goal is to make the application run upon the Alite[1] software toolkit. This is important because of the Highway[6] project. This project uses the Alite as well. It is good to run both projects upon the same platform.

Modularity : My project can be easily extended or any part of the project can be replaced by another implementation of given functionality. This goal is important, because if any of the parts is unsuitable for given task, it could be easily exchanged. For example, another types of vehicles can be easily added or new type of the visualization can be included.

Experimental scenarios : When the application is completed, I have to demonstrate its capabilities on some scenarios. First of all, I will show crash of the vehicle with some static object. Next scenario is situated to common road traffic. This shows that my application can handle even more complex situations than simulating single crash. There are more complex maneuvers that must be taken into account. The last scenario shows crash of two vehicles. Various scenarios show that my application does not focus on particular problem. It can handle various types of situations.

Chapter 3

Analysis and design solutions

First of all, I will kindle programming language and related programming environment. Next, I will write about the application structure and its parts. Splitting the application into more-or-less independent parts is vital. It ensures modularity of the application and meets one of the goals I intended to achieve. Application schema can be seen on figure 3.1. It shows parts of the project and how the parts are related to each other.

3.1 Programming environment

My project can be used on various types of devices. Therefore, the programming language should be platform independent. I chose Java as my programming language. Next reason for doing so is that my project is intended as a "module". Other projects, which could potentially use my module, were written in Java as well. Related to this is choosing the programming environment. I chose the Netbeans integrated development environment. It is more than sufficient for creating such application. In addition, it is my preferred programming environment.

3.2 Application structure

The application should be as modular as possible (the application structure can be seen on figure 3.1). Each part can be replaced under the condition that new part solves given problem. For example one "headlong" agent can be replaced by another agent, which drives more carefully. I divided it into several parts.

- The agent is the "intelligent" part of the vehicle. In the real world it could be called the driver. It is stored in the storage and it contains sensors and actuators which communicates with the storage (the agent gets information of the vehicle state through the sensors and change its state through the actuators).
- The visualization is optional. It may be included in the simulation, but in the application in the real world it is not necessary. So its usage depends on the case of usage of the simulation. I implemented the visualization using the openGL.

- The creator is responsible for initialization of whole environment. It connects my application with the Alite toolkit (see section 1.3).
- By the sensors and the actuators the agent can control its vehicle. The actuators are responsible for changes of the data in the storages. They can be chained, and should form various levels of abstraction upon the environment state described in the storages. The sensors mediate information of the environment to the decision making algorithms (entities, agents, actions, other sensors). The sensors can be also chained and should form abstraction layers of the received information.
- The storage is the only object in the application, which communicates with the bullet physical library. It holds data structures representing a state of the elements in the environment, all the physical objects and it contains the simulation loop.
- Entities represents vehicles. They contain data about particular vehicle and information about its current state. They are stored in the storage together with their agents. Another goal is to make more than one type of the vehicle. I implemented several types of vehicles. They share common interface so they can be handled as one type of vehicle.

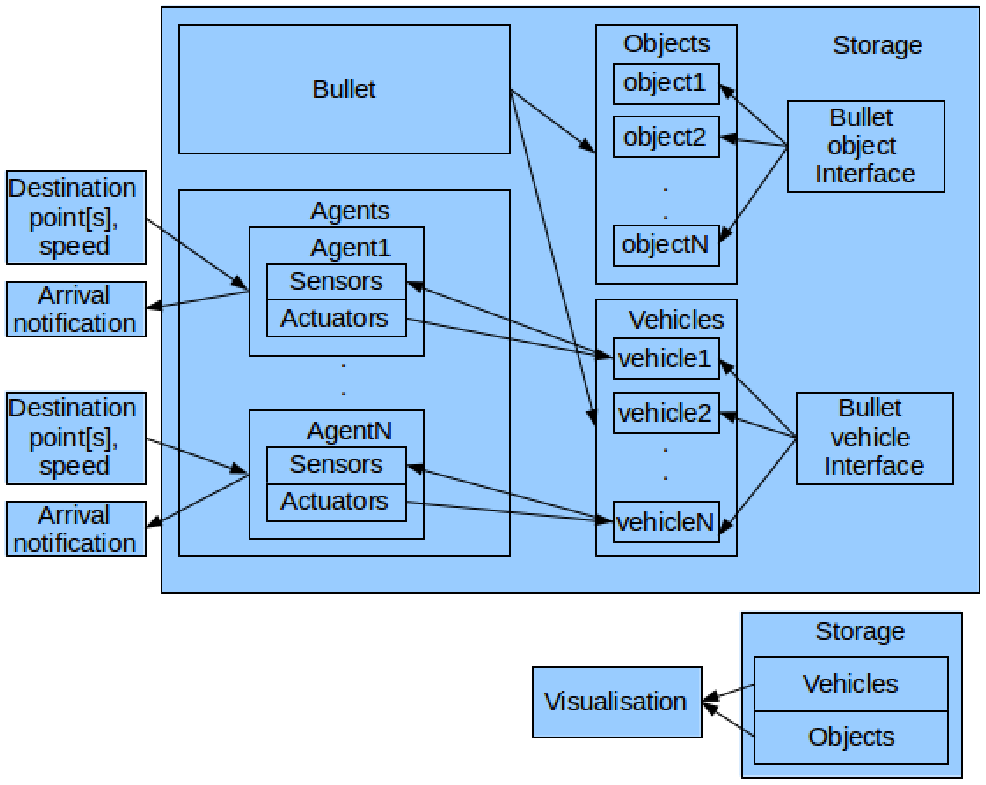


Figure 3.1: Application structure

As can be seen on figure 3.1, the storage contains all data necessary to run the simulation. The Bullet updates the data in each simulation step. This data is used for visualization and sensoric output. Objects share common Bullet object interface and vehicles share Bullet vehicle interface. Each vehicle has its own agent that contains sensors and actuators. The sensors provide data to their agent. Agent can make decisions over that data and it can affect the state of the vehicle with the actuators. Actuators and sensors form the "intelligence" of the agent.

Important parts of the application are input and output parameters. Agents are provided with destination points and speed parameters. These parameters affect agent behavior. When the agent is supplied by this data, he tries to reach the destination point with given speed. After the agent arrives to the destination position, he informs the provider of the data about the fact that he reached desired position.

By the provider of the data, I have in mind another agent. Let's call this agent the planner. The planner is an agent, which is capable of planning the route for the vehicle. It can supply navigation points, which the vehicle must reach and speed, by which the vehicle has to reach them.

Chapter 4

Implementation

First, I would like to discuss all the parts independently and then show connections between them. In the beginning, I want to describe objects, entities and their interfaces and how they are related to each other. Then I will discuss the simulation itself and objects that are needed to run the simulation. That includes the event processor, the bullet environment, the storage and other bullet related classes. Also, I will show how the storage is related to the jBullet physical library. Next, I will describe the visualization. That includes technology I used, connection to the simulation, camera modes and keyboard control of the camera. After that, the agent takes the turn with its actuators and sensors. I will show how sensors and actuators are implemented and connected together to form "intelligent" behavior. How they communicate with the storage and the agent. And finally, I would like to describe how this fits together - the creator class. This class is responsible for connections between main parts of the application. It shows how easily the application could be extended if necessary and how simple the application interface is.

4.1 Entities, objects and their interfaces

Simulation of the ground vehicles would not be complete without the vehicles and some additional objects (obstacles, buildings, etc.). The basic building blocks of the simulation are simple objects. I implemented several basic objects, from which more complex objects could be created. These objects are box, cone, cylinder and sphere. A cone and a cylinder can be created around one of the axis of the coordination system (x, y or z). All of these objects implement common BulletObjectInterface. This ensures that all the objects behave identically. Every object has its name for addressing, weight, size, collision shape, rigid body and transform. The collision shape determines its rigid body shape. The transform, rigid body and collision shape are used for computing object parameters in the bullet engine. Special case of an object is the ground object. It consists of a triangle mesh and is used as a ground surface.

The vehicles are created from these simple objects. Basically, they consists of a box as a chassis and a certain number of cylinders as wheels. All the vehicles share common Bullet vehicle interface. That ensures that all the types (types of the vehicles I implemented for the simulation is shown on figures 4.2, 4.3, 4.4 and 4.5) of the vehicles can be handled identically. Each vehicle creation (the schema of the vehicle creation can be seen on figure 4.1) is provided with its position and a name. At first, the collision shape and the default vehicle raycaster are created and the transform is computed.

```
vehicleRayCaster = new DefaultVehicleRaycaster(
    SingletonDiscreteDynamicsWorld.getInstance());
chassisShape = new BoxShape(size);

Transform localTrans = new Transform();
localTrans.setIdentity();
localTrans.origin.set(0, 1, 0);
```

Then the compound shape is created from this transformation and the collision shape.

```
compound = new CompoundShape();
compound.addChildShape(localTrans, chassisShape);
```

In the next step, the rigid body is created from the position, the mass and the compound.

```
carChassis = CreateRigidBody.create(
    mass, compound, positionX, positionY, positionZ);
```

The compound determines the shape of the rigid body. At last, the raycast vehicle is created from this rigid body, tuning (physical parameters of vehicle) and the vehicle raycaster (simple common vehicle).

```
vehicle = new RaycastVehicle(tuning, carChassis, vehicleRayCaster);
```

Finally, wheels are added to the vehicle and the vehicle is completed.

```
float connectionHeight = 1.2f;
boolean isFrontWheel = true;
vehicle.setCoordinateSystem(rightIndex, upIndex, forwardIndex);

Vector3f connectionPointCS0 =
    new Vector3f(
        CUBE_HALF_EXTENTS -(0.3f * wheelWidth),
        connectionHeight,
        2f * CUBE_HALF_EXTENTS - wheelRadius);
vehicle.addWheel(
    connectionPointCS0,
    wheelDirectionCS0,
    wheelAxleCS,
    suspensionRestLength,
    wheelRadius,
    tuning,
    isFrontWheel);
    .
    //another wheels
    //amount depends on vehicle type
    .
for (int i = 0; i < vehicle.getNumWheels(); i++) {
    WheelInfo wheel = vehicle.getWheelInfo(i);
    wheel.suspensionStiffness = suspensionStiffness;
    wheel.wheelsDampingRelaxation = suspensionDamping;
    wheel.wheelsDampingCompression = suspensionCompression;
    wheel.frictionSlip = wheelFriction;
    wheel.rollInfluence = rollInfluence;
}
```

Behavior of each vehicle is determined by the set of its parameters such as a maximal engine force, maximal braking force, steering clamp, suspension stiffness etc.

```
//tunning parameters
private final int mass = 800;
private float gEngineForce = 0.f;
private float gBreakingForce = 0.f;
private float maxEngineForce = 1000.f;
private float maxBreakingForce = 100.f;
private float gVehicleSteering = 0.f;
private float steeringIncrement = 0.04f;
private float steeringClamp = 0.3f;
private float wheelFriction = 30f;//1e30f;
private float suspensionStiffness = 20.f;
private float suspensionDamping = 2.3f;
private float suspensionCompression = 4.4f;
private float rollInfluence = 0.1f;//1.0f;
private final float suspensionRestLength = 0.6f;
```

The vehicle (see the complete schema of the vehicle creation on figure 4.1) contains methods and functions for manipulating with its parameters. Each vehicle stores its internal state in its object. Some parameters are redundant and could be omitted. But, because I want to separate the simulation and the visualization, I have to store these parameters as well as those I need. This redundant parameters (transform, position, speed and direction angle) are periodically modified by the storage and serves as an interlayer between the simulation and the visualization.

```
//omissible parameters
private float actualPositionX;
private float actualPositionY;
private float actualPositionZ;
private float currentSpeedInKm;
private Vector3f directionAngle = new Vector3f();
```

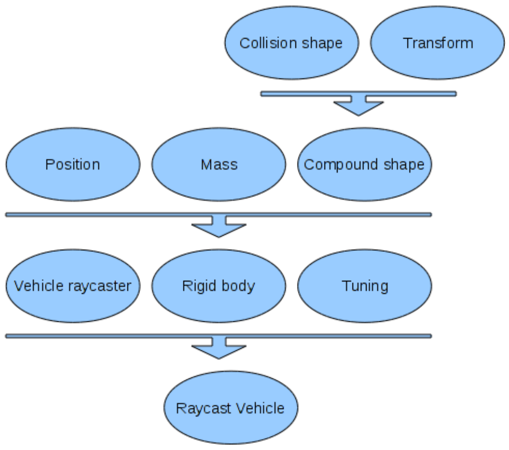


Figure 4.1: Creation of the vehicle

Bus vehicle type that represents a bus (see figure 4.2)

Car vehicle type that represents a common car (see figure 4.3)

Bike vehicle type that represents a motorbike (see figure 4.4)

Threewheeler vehicle type that represents a three-wheeled vehicle (see figure 4.5)

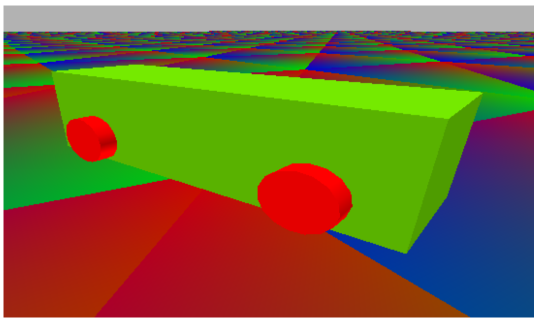


Figure 4.2: Type of the vehicle: Bus

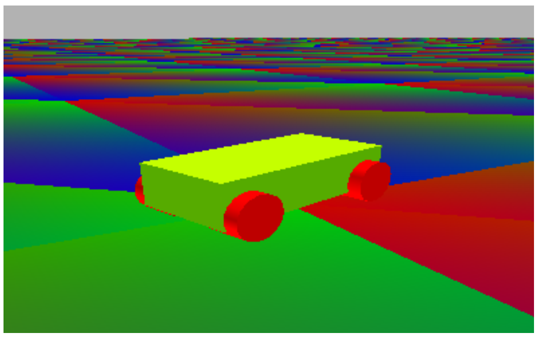


Figure 4.3: Type of the vehicle: Car

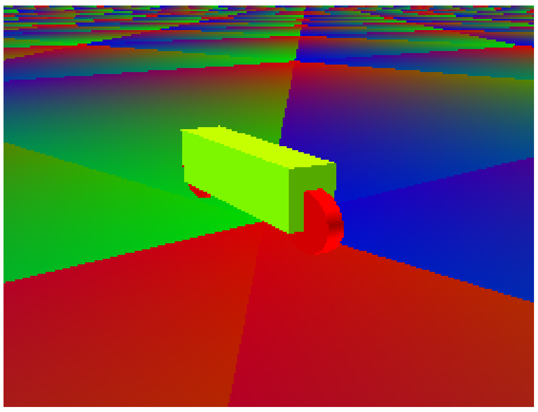


Figure 4.4: Type of the vehicle: Bike

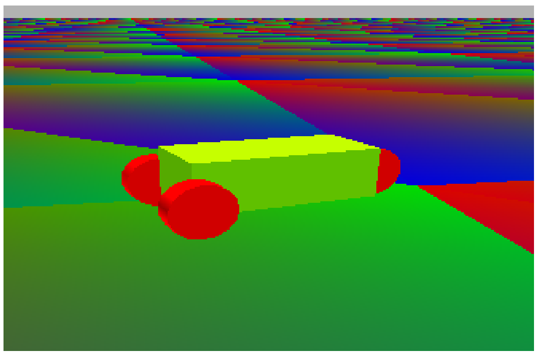


Figure 4.5: Type of the vehicle: Threewheeler

4.2 Simulation

The simulation class is in fact the event processor. The event processor is part of the Alite toolkit (see 1.3). It contains an event priority queue.

An event queue can be used for any queued processing of discrete-time events. The API[1] provides methods for adding and handling of events. Additionally, the events are prioritized by their time and processed in this particular order. Events with same timestamps are treated as simultaneous and the particular ordering of their processing has to be considered as random (or simultaneous if they are processed concurrently).

An event queue is a core of the simulation. Each event is created with specified time when the event has to happen and it is stored in the priority event queue with this time as priority. When the simulation is started, the first event in the queue is drawn, the time of event is set as time of simulation and event is passed to its handler. When the handler is done the next event is drawn. Every handler must implement EventHandler interface.

Additionally, the simulation queue is extended by methods for pausing, stopping and delaying of the queue processing to provide variable speed of simulation.

Each element of the simulation can use the events as it wants. For example, there can be processes in the environment, which duration is mathematically computed and the time of the event differs according to the needs of the process. Together with such processes, there can be regular event loops ticking each N milliseconds. And finally, there can be also processes, which are triggering complex hierarchies of events dependent on each other to simulate dichotomies in the simulated phenomenons.

Event processor (which can be used to describe successive processes in the environment) mentioned above is included in the bullet environment. The bullet environment is inherited from the EventBasedEnvironment. It contains event processor and routines for creating sensors and actuators. In addition, it contains storage.

The storage is the only object in the application, which communicates with the bullet physical library. It holds data structures representing a state of the elements in the environment. There are two typical approaches[1] how to slice the environment state using the storages. One is grouping of the element features according to the element type and the other is grouping of the element features according to the data representations of the features. I am using storage of the first type for storing vehicles, objects, sensors and agents.

Vehicles, objects, agents and sensors are stored in hashmaps like <name, object>. Related to this hashmaps are methods and functions, which manage them. There are methods and functions for adding, getting and removing objects, vehicles and sensors to or from the hashmap. They are very simple. For example method, that adds vehicle into the hashmap looks like this:

```
public void addVehicle(BulletVehicleInterface vehicle) {
    vehicleMap.put(vehicle.getName(), vehicle);
}
```

The storage also contains methods for updating vehicles', objects' and sensors' data. These methods are invoked in each simulation step to refresh the information about states of objects in the environment. Position, speed, direction angle, transformation and sensors of the vehicle must be updated in each step. Objects require only transformation update.

The most important construction in the storage is an event loop. An event loop is a repetitious process. It is started by a newly added event into an event queue. The respective handling process of the event then adds the same event into the queue at the end of its processing, which invokes next iteration of the same process. In my case it looks like this:

```
@Override
public void handleEvent(Event event) {
    if (event.isType(TacticalEventType.STEP)) {
        long realStepTime =
            getEventProcessor().getCurrentTime() - lastStepTime
        lastStepTimeStamp =
            getEventProcessor().getCurrentTime();

        SingletonDiscreteDynamicsWorld.
            getInstance().
                stepSimulation(realStepTime, maxSimSubSteps);

        for (
            Map.Entry<String, BulletVehicleInterface> entry :
            vehicleMap.entrySet()
            )
            {
                entry.getValue().move();
                entry.getValue().updatePosition();
                entry.getValue().updateSpeed();
                entry.getValue().updateDirectionAngle();
                updateTransformation(entry.getValue());
            }

        for (
            Map.Entry<String, BulletObjectInterface> entry :
            objectMap.entrySet()
            )
            {
                updateTransformation(entry.getValue());
            }

            updateSensors();

            getEventProcessor().addEvent(
                TacticalEventType.STEP, this, null, null, STEP_TIME);
    } else if (
        event.isType(SimulationEventType.SIMULATION_FINISHED))
    {
```

```
    System.out.println("Simulation over...");
    }
}
```

Let's describe the whole event loop. I'll do it, because it is the main computational loop in the application.

- Firstly, time difference between the actual event time and the last event time is computed. This has to be done, because the bullet engine computes its values in dependence of this difference.
- Next, there is the command, which says to the bullet engine to recompute its values in dependence of the time difference computed in the previous step.
- After the bullet engine recomputes its values, the service methods of vehicles and objects are called.
- As last command, there is our cause of looping. It is command, which adds (with some delay) to the queue the same event as is the one currently processing. This invokes next iteration.

The storage is connected to the jBullet physical library in this event loop. Calling method stepSimulation() inside the event loop forces the jBullet physical library to recompute its state. Those recomputed data are fetched by update methods and stored in the storage for next use.

```
SingletonDiscreteDynamicsWorld.
            getInstance().
                stepSimulation(realStepTime, maxSimSubSteps);
```

The method stepSimulation() is implemented in SingletonDiscreteDynamicsWorld. This object wraps the original DiscreteDynamicsWorld and makes singleton from it. The DiscreteDynamicsWorld provides discrete rigid body simulation. The original DiscreteDynamicsWorld automatically takes into account variable timestep by performing interpolation instead of simulation for small timesteps. It uses an internal fixed timestep of 60 Hertz. For my purposes I set the time step to 100 ms. The simulation step will perform collision detection and physics simulation. But to use the DiscreteDynamicsWorld, it must be created first. Responsible for creation of the DiscreteDynamicsWorld is the InitPhysics class.

The InitPhysics class is responsible for creating the DiscreteDynamicsWorld. It creates objects which are needed by the DiscreteDynamicsWorld to run the simulation and is capable of adding new objects into the world. The most important objects are:

- The BroadphaseInterface is capable of broadphase collision detection. This collision detection provides acceleration structure to quickly reject pairs of objects based on axis aligned bounding box (AABB) overlap.
- The ConstraintSolver is a mechanism for computing constraints between two and more rigid bodies.

- The CollisionDispatcher iterates over each pair, searches for a matching collision algorithm based on the types of objects involved and executes the collision algorithm computing contact points.

4.3 Visualization

I use OpenGL for visualization. Visualization runs in a separated thread. Therefore, it is independent of the simulation. This also causes that the visualization can be disabled in the case that it is not necessary to run it. This brings certain problems. The visualization cannot fetch the data directly from the storage. All the computed data must be stored outside the storage and then have to be fetched by the visualization thread.

4.3.1 JBGraphicsThread

This is a separated thread, that runs in parallel with the simulation thread. It contains an infinite loop that handles the visualization. After each step, the thread sleeps over a certain period of time, otherwise it would slow down the computation.

```
while (true) {
    keyboardHandling.manageKeyboard();

    long startNanos = System.nanoTime();

    doRender(vehicles, objects);

    long endNanos = System.nanoTime();

    long sleepNanos = (long)
        (1.0 / FPS_MAX * 1000000000.0) - (endNanos - startNanos);
    sleepNanos = sleepNanos < 0 ? 0 : sleepNanos;
    long sleepMillis = sleepNanos / 1000000;
    sleepNanos -= sleepMillis * 1000000;

    try {
        Thread.sleep(sleepMillis, (int) sleepNanos);
    } catch (InterruptedException ex) {
    }
}
```

The method doRender(vehicles, objects) (line 6) causes the visualization to draw the vehicles and the objects along with the environment (ground). It also allows to control the camera with the help of the keyboardHandling.manageKeyboard() (line 2) method in case that the vehicle is not being followed by the camera.

4.3.2 BulletGraphics

This is the main class used for the visualization. Its method doRender(vehicles, objects) draws the objects and the environment. It uses native OpenGL libraries for creating the visualization. The user can switch between three modes (see the camera modes on figures 4.6, 4.7 and 4.8) of visualization:

Static camera camera is static and it can be controlled by the keyboard (see figure 4.6)

Dynamic camera camera is following a certain vehicle (user can switch between the vehicles) (see figure 4.7)

Sticky camera camera is placed on the roof of the vehicle and is facing the same direction as the vehicle. (see figure 4.8)

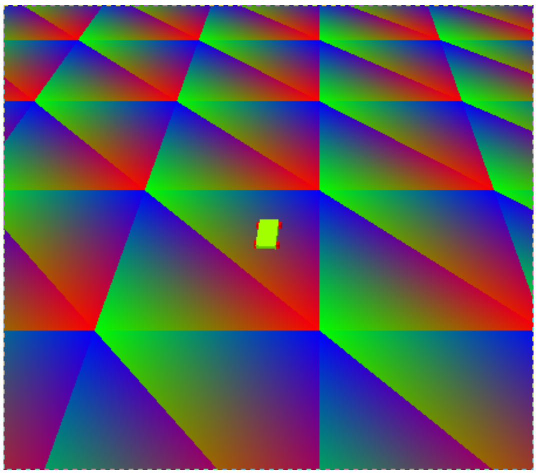


Figure 4.6: Static camera view

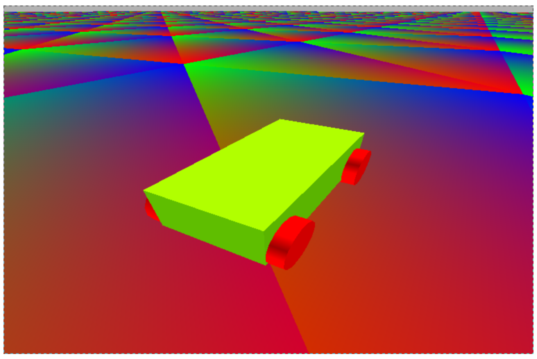


Figure 4.7: Dynamic camera view

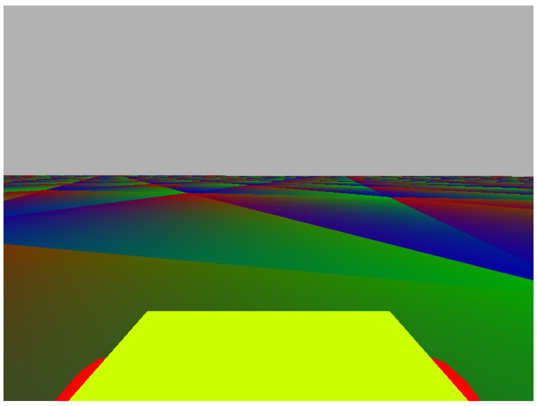


Figure 4.8: Sticky camera view

4.3.3 KeyboardGraphicsHandling

This class can control the camera within the visualization. It handles an event that was invoked by the keyboard. This action influences the graphics thread only. There is no direct way to control the camera. It can only affect parameters of the visualization thread and then the thread updates the BulletGraphics to behave as intended.

4.4 Agent

The agent is an autonomous object with flexible behavior. In my case, the agent is an object that controls the vehicle, which is addressed to him. The agent is capable of getting the vehicle to desired position with specified speed. It is able to do so because of sequel of sensors and actuators. The agent can communicate (see the communication schema on figure 4.9) with the environment through actuators and sensors only. The sensors are used for checking the state of the vehicle and the environment. The actuators are responsible for actions invoked in response to the sensoric input.

The sensors[1] mediate information of the environment to the decision making algorithms (entities, agents, actions, other sensors). The sensors can be also chained and should form abstraction layers of the received information. There are two types of sensors we consider: push and pull. The push sensors are based on callbacks implemented in the scope of the decision making (e.g., an agent implements reaction on a perceived change in the environment) and the callbacks are invoked by the sensor (storage -> sensor -> behavior). The pull sensors are called by the decision making code and the perceived information is returned by the method call (behavior -> sensor -> storage ..> sensor ..> behavior). The actuators[1] are responsible for changes of the data in storages. The actuators can be chained, and should form various levels of abstraction upon the environment state described in the storages. The actuators can use sensors to form shortened control loops (sensor -> actuator -> storage -> sensor -> ...), which can represent implicit mechanics of the environment (a rock falls even if it is not explicitly moved).

I have implemented two layers of sensors (complete overview of implemented sensors can be seen in table 4.2). High level layer communicates with the agent. Low level sensor layer communicates with the storage. The actuators (complete overview of implemented actuators can be seen in table 4.1) are handled in similar way. As well as the sensors, I have divided actuators into low level actuators and high level actuators. Low level actuators control the vehicle stored in the storage. High level actuators provide some more or less intelligent behavior (see the schema of sensors and actuators on figure 4.9).

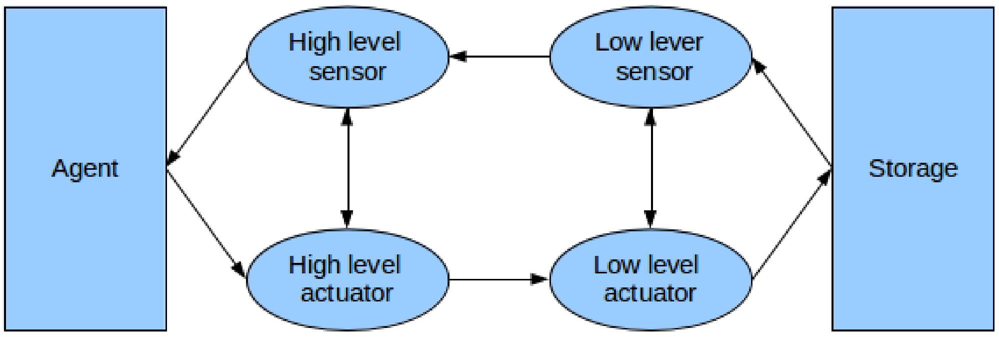


Figure 4.9: The schema of controlling the vehicle

Low level actuators	High level actuators
brake	changeSpeed
accelerate	moveToPosition
decelerate	stop
steerLeft	
steerRight	

Table 4.1: Actuators overview

Now, I will describe the actuators on both layers (see table 4.1). Low level actuators are simple methods that affect parameters of the vehicle. These methods takes parameters of the high level layer and affect state of the vehicle. They work with physical part of the vehicle. Brake causes the vehicle to put the brakes on and stop the vehicle. Accelerate and decelerate causes the vehicle to speed up or slow down. It is equivalent stepping or releasing a gas pedal of the vehicle. Steering works with a steering wheel and sets the angle of the front wheels of the vehicle. This concludes the low level layer of actuators which works directly with the vehicle. High level actuators layer works more intelligently. Actuator "Stop" forces the vehicle to immobilize. This actuator is very simple and almost the same as the one within the low level layer. ChangeSpeed and moveToPosition are more interesting. They both contains regulators. ChangeSpeed actuator regulate actual speed of the vehicle. The agent passes its input parameter for desired speed to this actuator. The actuator senses actual speed of the vehicle and if it is lower than desired speed, it computes regulation constant and pass it to the low level layer actuator "accelerate". If the speed is higher, it works the same. Only exception is that it calls "decelerate" actuator of the low level layer. The actuator moveToPosition is even more interesting. The agent passes its parameter of desired position to this actuator. The actuator senses actual position of the vehicle and compares it to the desired position. If the desired position differs from actual position, regulation constants must be computed. Firstly, steering regulation constant must be computed. I managed to compute it as an angle difference. This constant tells how much the steering wheel must be deflected. Next, I take an azimuth of the vehicle direction angle and an azimuth from the vehicle to the desired position. The decision, whether to steer left or right, is based on the azimuthal difference. After that, the regulation constant is passed to the low level actuator layer. SteerLeft or steerRight actuator is used in dependence of the azimuthal difference.

Low level sensors	High level sensors
senseEngineForce	senseSpeed
senseMaxEngineForce	sensePosition
senseBreakingForce	updatePosition
senseMaxBreakingForce	updateSpeed
senseSteering	updateDirectionAngle
senseSteeringClamp	
senseSpeed	
sensePosition	
updateSpeed	
updatePosition	
updateDirectionAngle	

Table 4.2: Sensors overview

The sensors I implemented are shown in table 4.2. Firstly, I will describe low level layer of sensors. The low level sensors are used for sensing vehicle parameters. There are two types of the sensors: push and pull sensors. The pull sensors are called by the decision making code and the perceived information is returned by the method call. The push sensors are based on callbacks implemented in the scope of the decision making. The low level layer of sensors implements three push sensors. UpdateSpeed, updatePosition and update DirectionAngle. As names says, updateSpeed updates actual speed of the vehicle, updatePosition updates actual position of the vehicle and updateDirectionAngle updates actual vector the vehicle is facing. These sensors are called in each event loop of the storage. They pass the information of these parameters to the high level sensors layer. Next, there are some pull sensors. One is used for sensing actual speed of the vehicle and one for actual position. These two sensors are pull variants for updateSpeed and updatePosition sensors. The user can sense these two parameters even in between two steps of the storage event loop (for example for testing purposes). There are more pull sensors. Engine force can be sensed by the senseEngineForce sensor. Braking force can be sensed by the senseBrakingForce sensor. SenseMaxEngineForce and senseMaxBrakingForce sensors are used for regulation of speed. Actual steering of the vehicle can be sensed by the senseSteering sensor. The SenseSteeringClamp sensor is used for navigation control.

4.4.1 Low level sensor layer

The low level sensor layer communicates with the storage. It consists of two types of sensors. Push and pull as was mentioned above.

- Push sensors in my case looks like this:

  ```
  public void updateSpeed(float speed) {
      bulletHighLevelSensor.updateSpeed(speed);
  }
  ```

  This method is invoked by the storage in its event loop. It has no return value and it invokes push sensor in the upper sensor layer.

- Pull sensors are good for asking for the values at any time during the simulation. They are independent of the storage event loop. But the information could not be actual. The value of the parameter was computed at the end of the last simulation step. Pull sensor could look like this:

  ```
  public float senseSpeed() {
      return storage.
          getVehicle(getRelatedEntity().
              getName()).
                  getCurrentSpeed();
  }
  ```

For my purposes I have implemented push sensors for speed, position and direction angle of the vehicle. Pull sensors implements sensors for speed and position as well. In addition, I have implemented sensors for sensing engine force, maximal engine force, braking force, maximal braking force, steering and steering clamp. Some of these parameters are used by the low level actuator layer and some of them are passed to the high level sensor layer.

4.4.2 High level sensor layer

As well as the low level sensor layer, this layer consists of two types of sensors. I have implemented pull sensors for speed and position and push sensors for position, speed and direction vector of the vehicle. This time the push sensors invoke actions in the high level actuator layer. I will describe actuators later.

4.4.3 Low level actuator layer

The low level actuator layer consist of methods that add events to the event processor. Then these events are processed in the handleEvent method. In my case the method that adds the event into the event processor looks like this:

```
public void accelerate(float regulateSpeed) {
    this.regulateSpeed = regulateSpeed;
    getEventProcessor().addEvent(
        TacticalEventType.BULLET_ACCELERATE,
        this, null, null);
}
```

This method is invoked by the upper actuator level. Then the event is processed in the handleEvent method.

```
public void handleEvent(Event event) {
    if (storage.getVehicleMap().get(agent.getName()) != null) {
        BulletVehicleInterface vehicle =
            storage.getVehicle(getRelatedEntity().getName());

        if (event.isType(TacticalEventType.BULLET_ACCELERATE)) {
            vehicle.setEngineForceUp(
                bulletLowLevelSensor.
                    senseMaxEngineForce() * regulateSpeed);
            vehicle.brakeDown(
                    bulletLowLevelSensor.
                        senseMaxBreakingForce());
        }
        .
        .
        .
        if (event.isType(TacticalEventType.BULLET_DECELERATE)) {
            vehicle.setEngineForceDown(
                bulletLowLevelSensor.
                    senseMaxEngineForce() * regulateSpeed);
            vehicle.brakeUp(
                bulletLowLevelSensor.
                    senseMaxBreakingForce() / regulateSpeed);
        }
    }
}
```

What kind of event was added can be identified by the type of the event. According to the type of the event the service routine is called. Processing of the event affects the state of the storage and objects stored in it. As can be seen in the application code, the handleEvent method of the low level actuator layer uses sensoric input from the low level sensor layer for its work.

This layer is capable of controlling acceleration and deceleration of the car as well as stopping it. For that it uses the possibility to affect the engine force of the vehicle. In addition, it is capable of turning the vehicle to the left or to the right by setting its steering parameter.

4.4.4 High level actuator layer

This layer is almost the same as the low level actuator layer with one exception: it communicates with the agent and is prohibited to affect the storage. This layer takes commands from the agent and process them. Then it uses its own commands to command the low level actuator layer. It is expected that this layer will behave more intelligently than the lower layer. It is capable of regulating vehicle speed and navigating the vehicle to desired position. I have achieved this by using the regulators. I will describe them later. First, I would like to talk about the overall structure of this layer. The principle is the same as the low level actuator layer. The agent invokes methods that add event to the event processor.

4.4.4.1 Events added by the agent

```
public void changeSpeed(float wantedSpeed) {
    this.stop = false;
    this.wantedSpeed = wantedSpeed;
    getEventProcessor().
     addEvent(TacticalEventType.BULLET_CHANGE_SPEED,
         this, null, null);
}

public void moveToPosition(Vector3f position) {
this.wantedPosition = position;
getEventProcessor().
addEvent(TacticalEventType.BULLET_WANTED_POSITION,
this, null, null);
}

public void stop() {
this.stop = true;
getEventProcessor().
addEvent(TacticalEventType.BRAKE,
this, null, null);
}
```

The first of them is invoked by the need of changing speed of the vehicle. The second expresses agent's desire to move the vehicle to another position and the last one expresses the request for stopping the vehicle.

4.4.4.2 Events added by the sensor

Next way how to add an event to the event processor is to invoke them by the sensoric input.

```
public void updateSpeed(float speed) {
        this.speed = speed;
        getEventProcessor().
```

```
         addEvent(TacticalEventType.BULLET_CHANGE_SPEED,
         this, null, null);
    }

    public void updatePosition(Vector3f position) {
        this.position = position;
        getEventProcessor().
         addEvent(TacticalEventType.BULLET_WANTED_POSITION,
         this, null, null);
    }
```

These methods are in fact push sensors from the high level sensor layer. Originally they were invoked by the storage event loop.

4.4.4.3 Event handling

For handling the events I use the same principle as in the low level actuator layer: handleEvent method. In this case it looks like this:

```
public void handleEvent(Event event) {
        if (!stop) {
            if (event.isType(TacticalEventType.BULLET_CHANGE_SPEED)) {
                regulateSpeed();
            }

            if (event.isType(TacticalEventType.BULLET_WANTED_POSITION)) {
                regulatePosition();
            }
        }

        if (event.isType(TacticalEventType.BRAKE)) {
            bulletLowLevelAction.brake();
        }
    }
```

An event is processed and according to the type of the event the service routine is run.

4.4.4.4 Regulators

First, I would like to discuss the speed regulator.

```
private void regulateSpeed() {
    float speedDifference = wantedSpeed - speed;
    float regulateSpeed = 0f;
    if (Math.abs(speedDifference) > wantedSpeed * 0.05f) {
        regulateSpeed = 1f;
    } else {
        regulateSpeed = Math.abs(1 - Math.abs(speed / wantedSpeed));
    }
    if (speedDifference > 0) {
        bulletLowLevelAction.accelerate(regulateSpeed);
    } else {
        bulletLowLevelAction.decelerate(Math.abs(regulateSpeed));
    }
}
```

Firstly, the difference between actual speed and desired speed is computed. Then the regulation constant is being computed. When the difference is big (over 75 percent of desired speed), I set the engine force to its maximum value. Then, as the difference lowers, I start to regulate subtly to achieve smooth regulation. When I know how much I will affect the engine force, I need to know whether to accelerate or to decelerate. This can be identified from the sign of the difference value.

In the lower level I use only the regulation constant to affect the speed of the vehicle.

```
if (event.isType(TacticalEventType.BULLET_ACCELERATE)) {
    vehicle.
        setEngineForceUp(
            bulletLowLevelSensor.
                senseMaxEngineForce() * regulateSpeed);
        vehicle.brakeDown(
            bulletLowLevelSensor.
                senseMaxBreakingForce());
}
```

4.5 Linking parts together

In the previous section, I described all the parts of the project independently. Now, I would like to describe how the parts work together to form the simulation and I will make overall description of the parts again. This will give complete picture of the application structure to the reader.

Responsible for creation of the simulation is the creator class. The creator is responsible for initialization of the whole environment, entities, visualization etc. The class must implement the Creator interface. The application is started by passing the appropriate creator to the Alite[1] main class. The rest of the command-line options is passed to the creator. The main method of the creator is the create() method. This method creates the simulation and I will now describe its content.

First of all, the physics must be created by creating the DiscreteDynamicsWorld. Responsible for that is the InitPhysics class. This piece of code creates the world for the simulation. It creates singleton instance of the DiscreteDynamicsWorld class. Also, InitPhysics class is capable of adding new objects to the dynamics world.

```
InitPhysics.initPhysics();
```

Next piece of code creates the Alite[1] simulation. This class is in fact the event processor that is responsible for handling the events provided by the application. The Simulation class starts and ends the simulation.

```
simulation = new Simulation();
simulation.setSimulationSpeed(simulationSpeed);
```

The event processor is included in the environment. The Environment class is inherited from the EventBasedEnvironment. This environment contains the event processor (which can be used to describe successive processes in the environment) and routines for creating sensors and actuators. In addition, it contains the storage. The storage contains all the objects in the simulation. The storage creates an event loop which is the main part of the simulation. In each loop, the storage computes all the data (vehicle positions, physical behavior of the vehicles, etc.) necessary to run the application.

```
environment = new BulletEnvironment(simulation);
```

Now, when we have completed the creation of the world, we are ready to add some objects into it. In this part of the create() method, the vehicles, the objects and the ground (special kind of object) are created. The process of creation is the same for the vehicles and the objects. First of all, we need to create an object as it is. Next, it is necessary to add the object to the DiscreteDynamicsWorld (to achieve physical behavior). And at last, add it to the storage. Adding to the storage is very important, because it is the storage that tells the bullet to recompute its data. For example:

```
//ground creation
Ground ground = new Ground("ground");
InitPhysics.addObject(ground.getRigidBody(), ground.getCollisionShape());
environment.getBulletStorage().addObject(ground);

//vehicle creation
Vector3f position = new Vector3f(0, 0, -250);
RegularCar car = new RegularCar(position, "car");
InitPhysics.addVehicleEntity(
    car.getCollisionShape(),
    car.getCompound(),
    car.getVehicle(),
    car.getRigidBody());
environment.getBulletStorage().addVehicle(car);
```

By this, we added some objects into our little world. For demonstration, I created visualization part. The visualization is optional. It may be included in the simulation, but when applied in the real world it is not necessary. Its usage depends on the case of usage of the simulation. The visualization runs in a separated thread. It uses the environment (in fact only the storage in it) to visualize the vehicles and the objects. The parameter vehicleToFollow is used for determining the camera type. When set to -1, the camera initially does not follow any vehicle. Otherwise, the camera follows selected vehicle. Of course, the type of the camera can be changed during run of the simulation.

```
int vehicleToFollow = -1;
Thread graphicsThread = new Thread(
    new JBGraphicsThread(environment, vehicleToFollow));
graphicsThread.start();
```

Up until now, we created the whole simulation environment, we can watch it using the visualization module, but if we start the simulation now, nothing is going to happen. That is caused by the fact, that the vehicles lack their drivers - the agents. An agent is the "intelligent" part of the vehicle. The vehicle and its agent are paired together by their names. The agent is stored in the storage as well as the vehicle. It contains sensors and actuators which communicate with the storage (the agent gets information of the vehicle state through the sensors and change its state through the actuators).

```
for (
    BulletVehicleInterface vehicle :
    environment.getBulletStorage().getVehicleMapAsArrayList()) {
    BulletAgent agent =
        new BulletAgent(vehicle.getName(), environment.handler());
    agent.getToPosition(waypoints);
}
```

The method getToPosition() provides navigation points and speed information to the agent. By the point is meant the target position (these data must be provided by an other agent that is capable of planning the route for the vehicle). This method is overloaded and can be used in four similar ways.

getToPosition(waypoints) The agent gets arraylist of waypoints. The Waypoint class is a class, which contains a data of wanted position, velocity and time of a maneuver. The agent drives through all of the waypoints and stops on the last one.

getToPosition(point, speed) The agent gets exactly one point, which represents wanted position. Speed of the movement is determined by the speed parameter. When the agent reaches its destination, it stops.

getToPosition(points, speed) The agent gets arraylist of points, which represents wanted positions. The agent drives through all of the points and stops on the last one. Speed of the movement is determined by the the speed parameter.

getToPosition(points, speedList) The agent gets arraylist of points, which represents wanted positions. The agent drives through all of the points and stops on the last one. Speed of the movement towards each point is determined by the speed parameter in the speedList arraylist.

Now, we have all the parts initialized and ready to go. Last thing remains to be done: run the simulation. The Simulation object is in fact the event processor. The method run() starts processing of the event queue. Starting the event queue causes creation of the event loop. Now, the simulation is running until the stop event is invoked.

```
simulation.addEvent(
    EventProcessorEventType.STOP,
    null, null, null,
    simulationTime);
simulation.run();
```

Now, the world is created and vehicles, objects and the ground are included in it. We can see it through the visualization. The vehicles are enriched by the agents. We are ready to run the simulation again. Everything looks as before, except one thing. The vehicle agents are now supplied with the data necessary to ride. The agents, which are supplied by such data, drive their vehicles to reach desired positions by desired speed.

Chapter 5

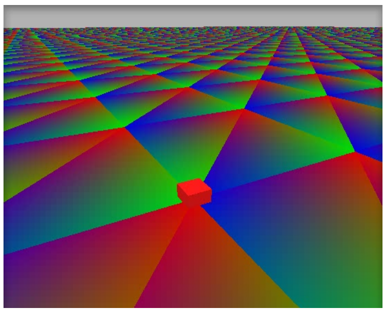
Testing

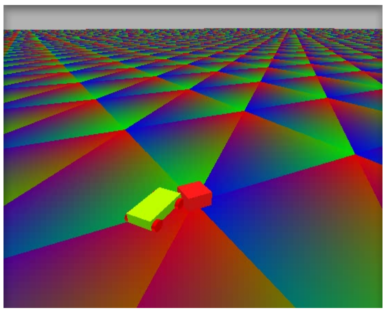
In this chapter, I will show some tests that will verify the functionality of my project. In first section, I would like to run tests with only single vehicle to show that the simulation works as intended. This tests include crashes and collisions of the vehicle with a static object and with another vehicle. I will show crashes of different types of vehicles to present differences in their behavior. For example, when the vehicle crashes into some other vehicle that is approximately of the same weight, the crash will look differently than the crash with light motorbike. After that, I will present some tests of regulation. That includes regulation of speed of the vehicle and regulation of ride on desired position. After these two sets of tests, I will combine both of them. I will show that the car is capable of safe driving and when the external impulse is applied (some other vehicle crashes into observed one), the car is capable of compensation of this impulse. This means that after the crash, the vehicle is still able to reach its destination position.

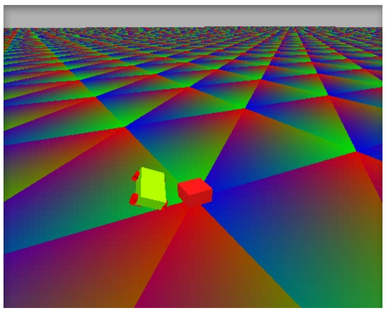
Next, I will test much bigger scenario. This include running multiple vehicles. I will show how much the simulation exploits computer system. This test focuses on multiple vehicles. Therefore, I will describe how rising amount of vehicles affects the time necessary to complete the simulation. I will run this test over several different scenarios. Firstly, I will present concurrent driving, when all the vehicles ride parallely next to each other. This means that no crashes are made and there is no need to compute collisions. This will show how fast the application can run in case that all agents drive safely. Then I will show scenario with random driving. Collisions are made randomly over the time. At last, I will create scenario with one huge conflict. All the vehicles in the simulation are placed in their initial positions and drive towards the single position point. At this point all of them crashes into each other. This shows the worst case that could happen.

5.1 Verification of the functionality of single vehicle

Now, I would like to present some tests over single vehicle. First of all comes the crash test, when the vehicle crashes with a static object. I prepared the scenario as an empty world with only this single vehicle (car) and only one static object. The object is simple rigid body with box-shaped collision shape.


(a)


(b)


(c)

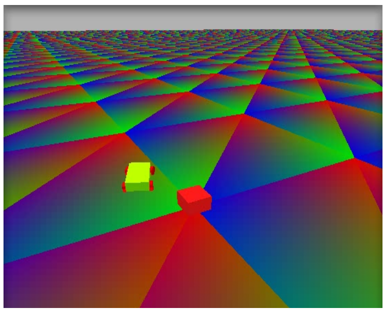
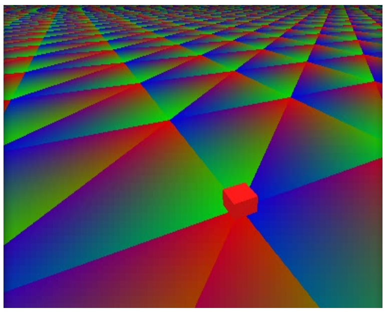
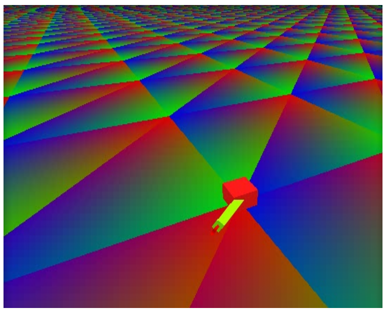
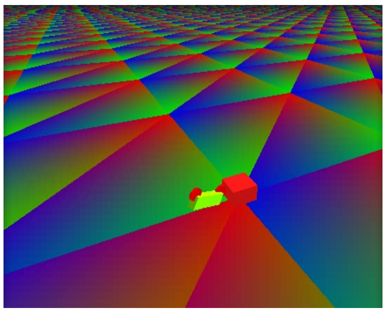

(d)

Figure 5.1: A car crashes into a static object


(a)


(b)


(c)

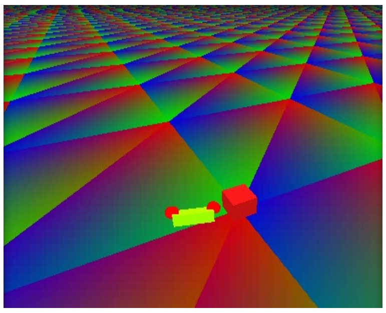

(d)

Figure 5.2: A motorbike crashes into a static object

The vehicle starts its ride with zero velocity and progressively accelerates up to 90 kmph. Then comes the crash. Screenshots of the crash can be seen on figure 5.1. I prepared the same test with a motorbike (see figure 5.2 for screenshots). Difference between the car and the motorbike is obvious. The motorbike is slightly lighter than the car. It shows how the weight of the vehicle affects the crash. The car crashes into the barrier and fends off, but still stays on its wheels. The motorbike crashes into the barrier and falls on the ground.

Now, I would like to show some crashes among vehicles. I will show what will happen when some light vehicle crashes into much heavier one. Next, I will show crash between vehicles approximately of the same weight. And finally, what will happen when heavy vehicle crashes into much more lighter vehicle. First case will include a motorbike crashing into a bus. Second case includes a car and a three-wheeled vehicle. Last case includes a bus crashing into a motorbike. This test will show differences between vehicle types. Following scenario consists of two parts. For each case I mentioned above I will show what will happen when moving vehicle crashes into another stopped vehicle. Second part will show crashes of vehicles that are both moving.

I shall start with a light vehicle crashing into a heavier vehicle. This case is shown on figure 5.3. The motorbike rides by 90 kmph and crashes into the bus. As can be seen, the motorbike again falls on the ground, but the bus holds still.

Then I will demonstrate what consequences will cause the crash between two vehicles of the same weight. This test includes a common car and a three-wheeled vehicle. Which car is moving is not essential because they both have the same weight. Again, one vehicle is moving and crashes into another one standing still. Screenshot from this case can be seen on figure 5.4. Now, the car crashes into three-wheeled vehicle bouncing him off. The car itself ends up on the roof.

At last, there comes the case, where a heavy vehicle crashes into a slightly lighter one. This test includes a motorbike as an "unsuspecting victim" of the crash and a bus as moving vehicle causing the crash. Screenshot of this crash can be seen on figure 5.5. From the screenshot is clearly seen that the motorbike had no chance. The bus swept it down and did not barely noticed that there is something under its wheels.

This test ends the part of static testing. From now on, I will run scenarios containing only moving vehicles. Stopped vehicles would be only crashed ones. These tests will be almost the same as three previous ones. I will create three scenarios, where a motorbike crashes with a bus, a car crashes with a three-wheeled vehicle and a bus crashes into a motorbike. But now, all the vehicles will ride by 90 kmph.

Short review:

- A light vehicle crashes into a heavy one standing still (figure 5.3).
- A vehicle crashes into another vehicle of the same weight standing still (figure 5.4)
- A heavy vehicle crashes into a light one standing still (figure 5.5)

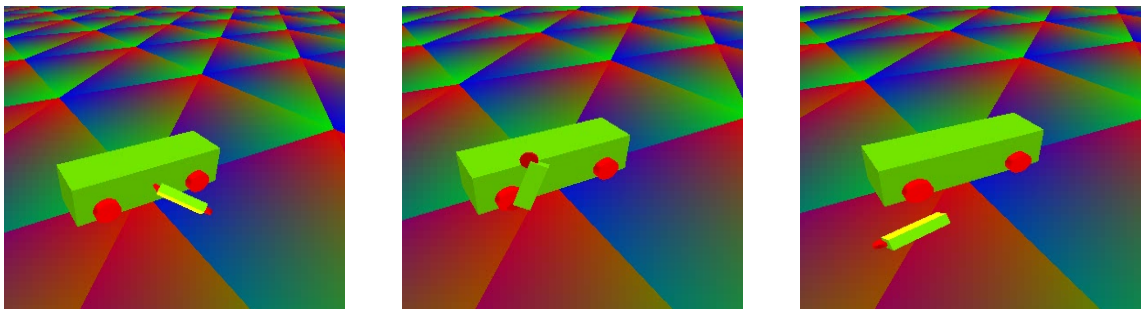

(a) (b) (c)

Figure 5.3: A motorbike crashes into a stopped bus

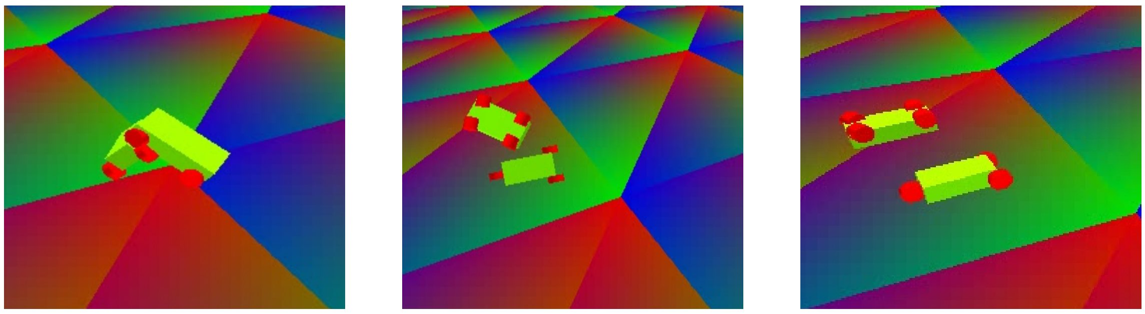

(a) (b) (c)

Figure 5.4: A car crashes into a three-wheeled vehicle

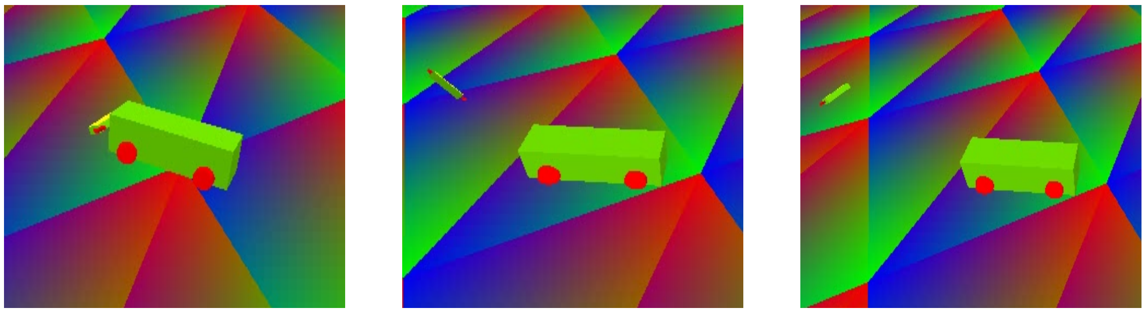

(a) (b) (c)

Figure 5.5: A bus crashes into a motorbike

The third in a sequence of tests is dynamic crashing. This includes crashing of vehicles that are both moving. I selected speed of 90 kmph. I thought this value would be the best to test because it is not as high as maximal highway speed and not as low as maximal road speed. As in the previous section, I created three scenarios with various types of vehicles.

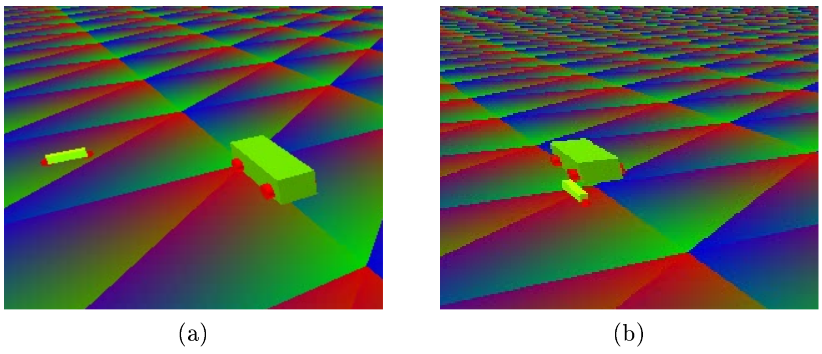
First scenario can be seen on figure 5.6. It shows how the accident can look like, when a motorbike crashes into a bus. As can be seen again, the bus almost did not notice any external impulse despite that it was a direct hit. It just continued its ride, but the bike was immobilized and in a real world, the driver would be probably dead.

Next screenshot (figure 5.7) shows two vehicles of the same weight. This crash does not look as terrible as the one in the previous section, where the car crashed into the stopped tree-wheeled vehicle. None of the vehicles overturned onto the roof. This is caused by the fact that it was not a direct hit. In this case, the three-wheeled vehicle only struck the back of the car.

Last screenshot (figure 5.8) shows crash of a bus and a motorbike again, but with one exception. The roles of the vehicles is now reversed. Now the bus strikes into the motorbike. And again, the bus did not notice any bigger disturbance. This can not be said about the motorbike. The motorbike flew about three bus lengths away.

Short review:

- A light vehicle crashes into a heavy one moving around (figure 5.6).
- A vehicle crashes into another vehicle of the same weight moving around (figure 5.7)
- A heavy vehicle crashes into a lighter one moving around (figure 5.8)


(a) (b)

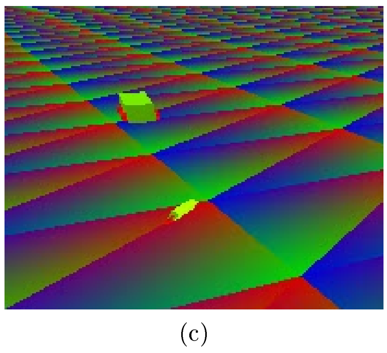

(c)

Figure 5.6: A motorbike crashes into a moving bus

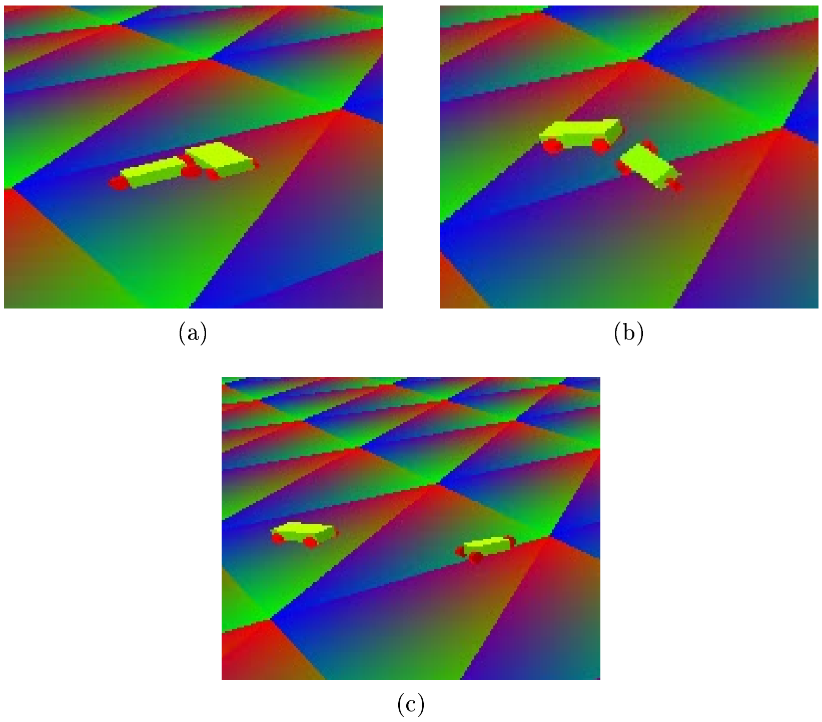


Figure 5.7: A car crashes into a moving three-wheeled vehicle

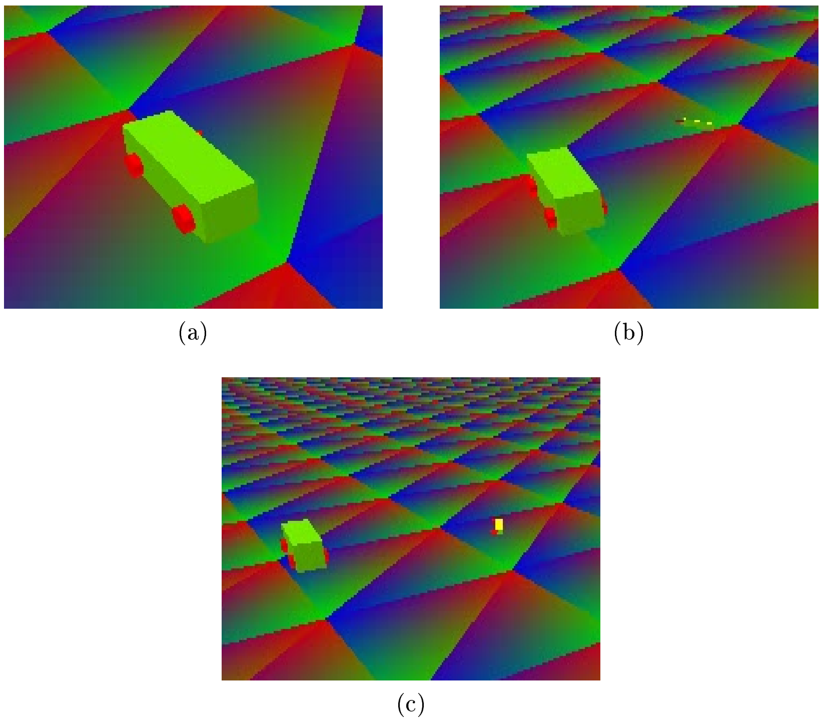


Figure 5.8: A bus crashes into a moving motorbike

5.1.1 Testing regulatory interventions

In the previous section, I presented tests proving that the jBullet physics library is capable of handling physical calculations. Next, I would like to present results of my regulators. The regulators ensures that the vehicles does not ride randomly, but under the control of some logic. I implemented two types of regulators. One is used for regulation of speed of the vehicle. The other one is used for navigating the vehicle from one point to another point (destined position). Speed regulation will now be discussed.

Speed regulation is extremely dependent on weight of the vehicle and its maximal engine force. It is quite logical that a lighter vehicle with powerful engine would be much more agile than a heavy vehicle with weak and inefficient engine. Comparison of the vehicles is shown in the graph on figure 5.9.

Goal of this test is to measure how much time takes the vehicle to reach speed of 90 kmph. I compared a light car with a heavy car. The light car weights 800 kilograms. The heavy car weights 1600 kilograms. The heavy car has two times more powerful engine than the light car. From the graph can be seen that the lighter vehicle can reach given speed faster than the heavy vehicle.

Next test shows direction angle regulation. I measure the angle difference between two vectors. One vector is the direction vector of the vehicle (direction the vehicle is facing). The second vector is computed as a vector from vehicle towards the destined position point. Therefore, maximal difference can be 180 degrees in case the destined position point is behind the vehicle and zero when the vehicle is facing towards the destined position point.

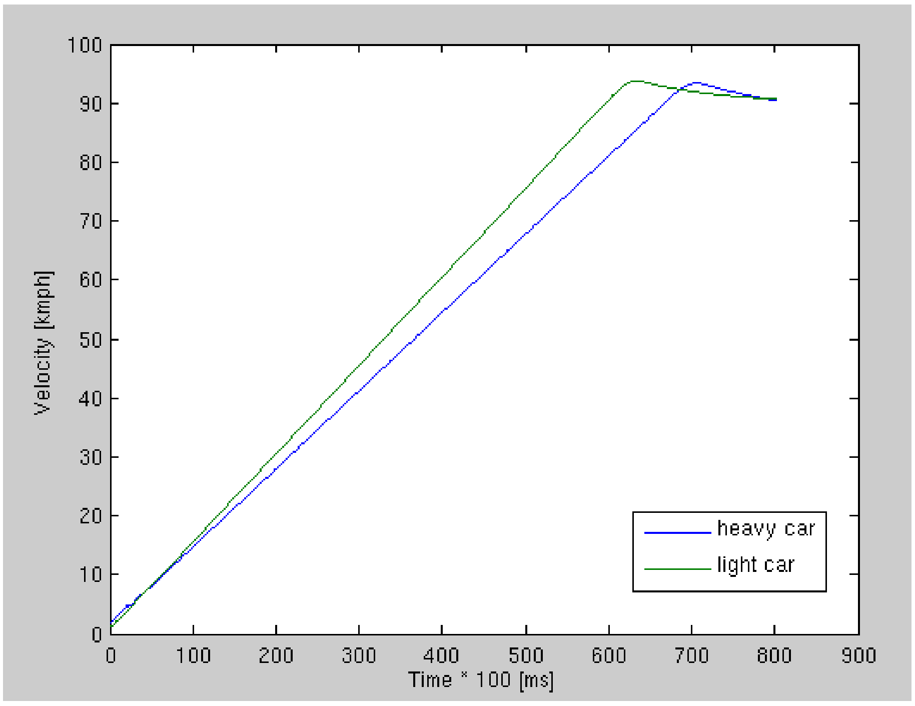


Figure 5.9: Comparison of regulation of the velocity

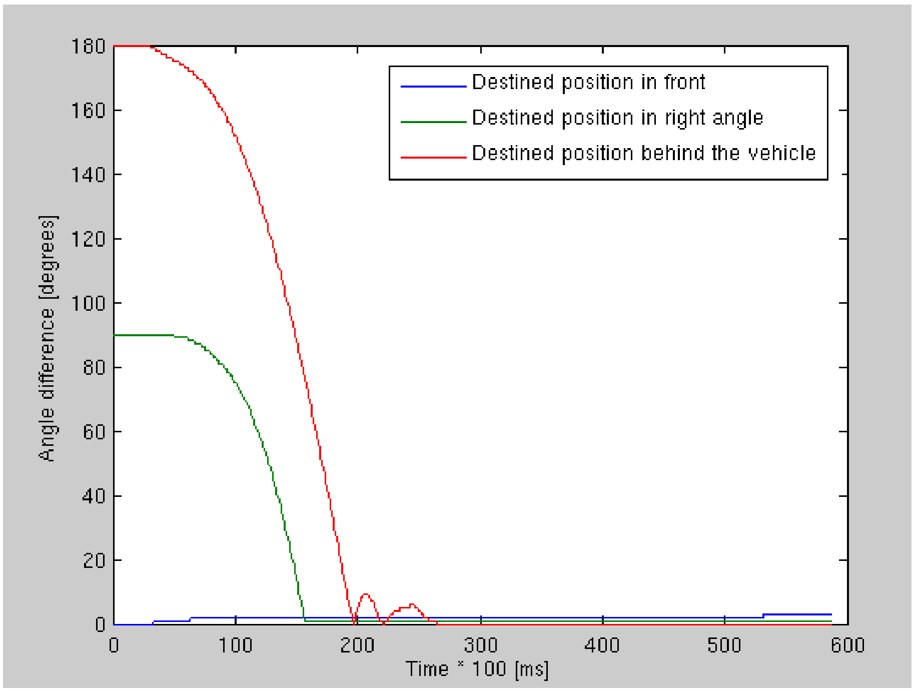


Figure 5.10: Comparison of regulation of the direction vector

I prepared several scenarios. First of them is that the destined position point is in front of the vehicle. Next shows the case that destined position is on the right side of the vehicle (in the right angle from the direction vector of the vehicle). In the third scenario, I placed the destined position point behind the vehicle. For results see the graph on figure 5.9. Some minor overshoots can be seen. It is caused by the regulator. I implemented it with 5 degrees tolerance to left and to from the direction vector of the vehicle. This tolerance must be taken into account when sending the vehicle at long distance. But it can be well compensated by the agent. When agent generates waypoints close enough to each other, the tolerance is negligible.

5.1.2 Testing reactions on external impulses

This section contains tests focused on external impulses. It means that fluent ride of the vehicle is interrupted by a crash. I will show the ability of the vehicle to restore its previous fluent ride after the crash. I prepared two scenarios. In first scenario, the vehicle itself crashes into another vehicle that blocks its way. In the second scenario, the car is being hit by another vehicle from the side.

In the first experiment, the car ride towards its destined position point. It accelerates up to 90 kmph. Everything would be fine if there were no obstacles. But there is a car blocking the way. Moving vehicle crashes into this immobilized vehicle. This crash causes changes in speed and direction vector. This changes can be seen on figures 5.11 and 5.13. On figure 5.11 can be seen that the vehicle was tilted by approximately 45 degrees from its original direction. This change was reverted by the direction regulator. On figure 5.13 can be seen speed change. The vehicle tried to reach 90 kmph, but then the crash has been made and the vehicle has been slowed. The regulator was able to reach desired speed after the crash.

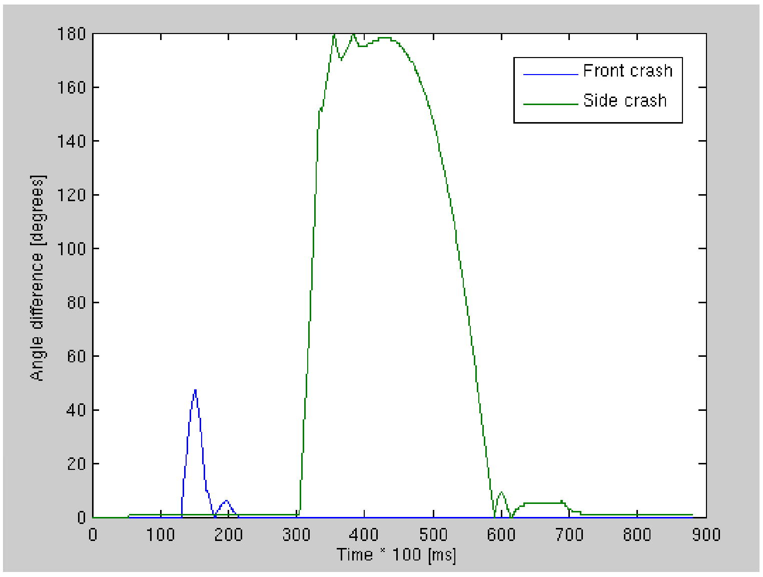


Figure 5.11: Direction vector regulation after a crash

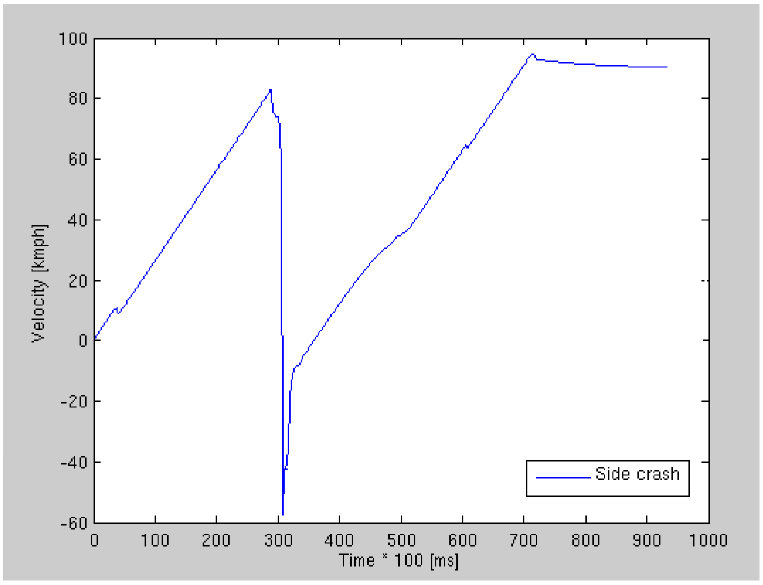


Figure 5.12: Speed regulation after a side crash

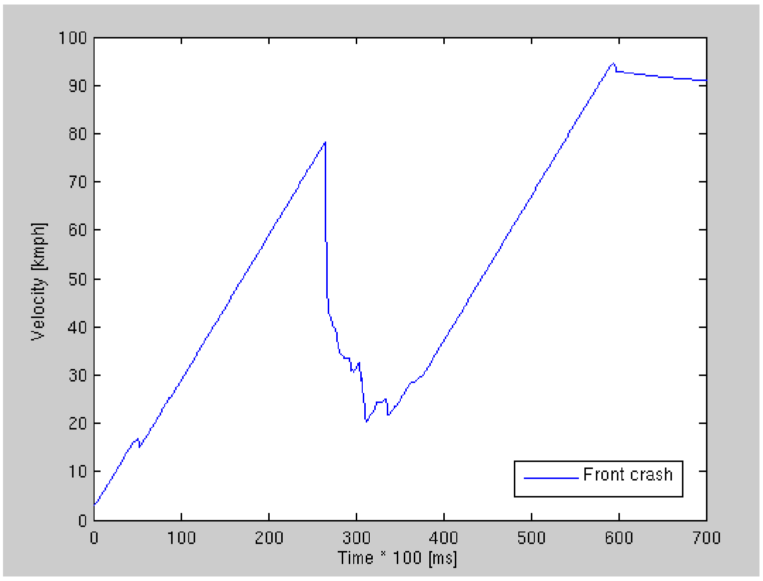


Figure 5.13: Speed regulation after a front crash

The second scenario is a bit different. The vehicle rides fluently to its destined position point. But another vehicle crashes it from the side. Consequences can be seen on figures 5.11 and 5.12. From the figure 5.11 can be seen that the vehicle was hit really hard. The vehicle was rotated by 180 degrees. From the figure 5.12 can be seen that the vehicle was stopped by this crash and even moved backwards for a moment. But both regulators were able to restore its previous ride (original direction and speed).

5.2 Verification of the functionality of multiple vehicles

This section focuses on testing time demands of the simulation. I would like to show how long it will take the simulation to finish. I prepared three scenarios to test this attribute of the simulation. The first scenario tests driving without collisions. Certain amount of vehicles are created and drive through the environment by 90 kmph. The vehicles have to ride out path which is 500 meters long. The vehicles ride concurrently to each other. No collisions are dispatched.

Next scenario presents random driving. The vehicles are created at random positions. Their agents drive through certain amount of random checkpoints. This causes random collisions. The vehicles drive by 90 kmph and randomly crash into each other. This will present an average case of the simulation run.

The last test consists of one big crash. The vehicles are created in one single line. All of them drive to opposite side of the map through the single checkpoint in the middle of the map. This causes the vehicles to crash in the central checkpoint. One huge crash is made. This will show how the simulation behave in the worst case.

The result of the first scenario can be seen on figure 5.14. This figure contains graph that shows time demands of the simulation in dependence of the vehicles amount. I run this test over a rising amount of the vehicles. From the graph can be seen that the time necessary to finish the simulation grows exponentially with the amount of the vehicles.

The second test consists of random driving. I created a certain amount of the vehicles at a time with random starting positions. I determined 5 random checkpoints to each vehicle. Five waypoints is sufficient amount. Starting points are generated within a square with 1600 meters long side. The waypoint are generated inside a square with 400 meters long side. The vehicles must drive through all the checkpoints. This scenario tests simulation running time in dependence of the vehicles amount when random collisions are dispatched. The results of this scenario can be seen on figure 5.15. As can be seen from the graph, when the amount of the vehicles is low, the probability of the crash is low. With a growing amount of the vehicles, the probability grows as well and the time necessary to run the simulation raises.

The last test focuses on a single huge crash of all vehicles. Again, I created a certain amount of the vehicles in a single line. All the vehicles try to reach their destined position points. But in this case all of them must drive through single checkpoint. This causes one huge crash around this checkpoint. The results of this test can be seen on figure 5.16. The graph shows that time grows exponentially. I must remark that last two values are not exact. The simulation ended after 10,000,000 events. Therefore, the simulation times for 400 and 500 vehicles could be higher than presented values.

From the tests I described above can be seen, and I assumed it from the beginning, that collision handling is quite demanding and time consuming calculation. All the tests result in a single conclusion: time necessary to finish the simulation grows exponentially with the growing amount of the vehicles. I tested this scenarios on another more powerful computer and I think that one computer can handle about 500 of vehicles.

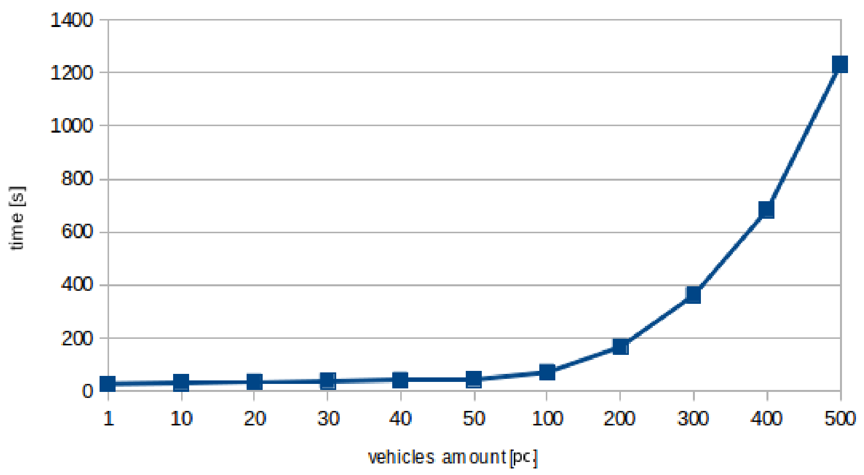


Figure 5.14: Testing time demands: Concurrent driving

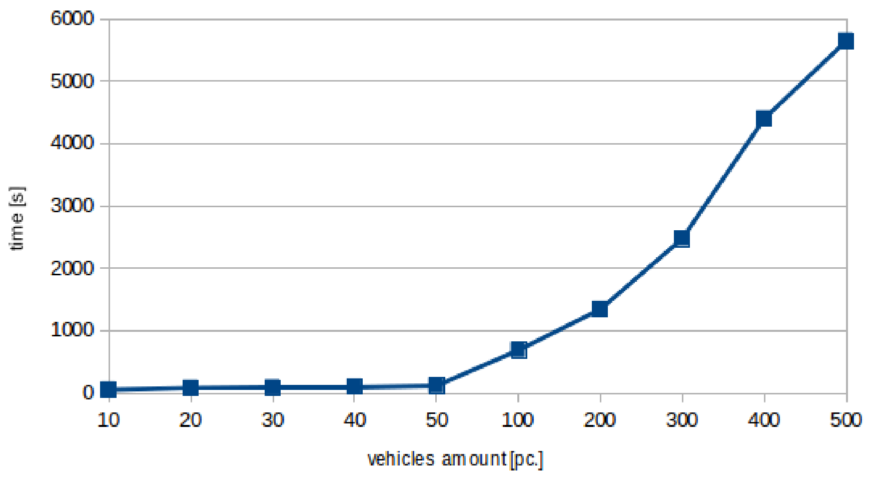


Figure 5.15: Testing time demands: Random driving

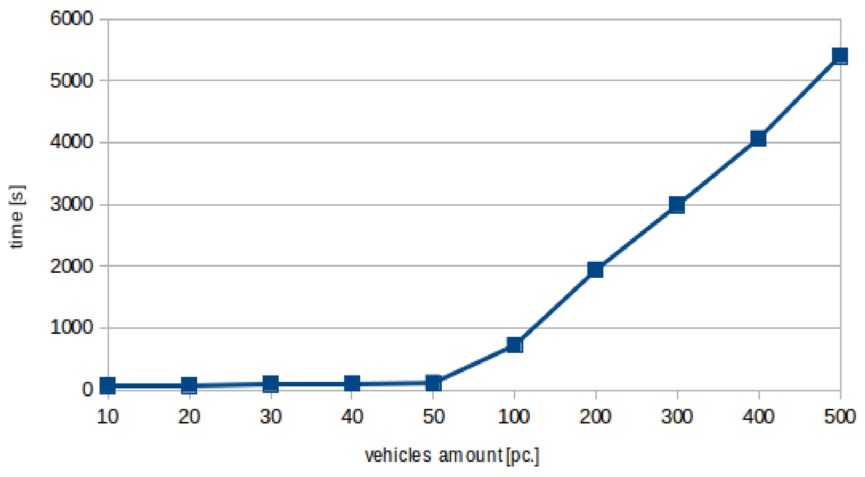


Figure 5.16: Testing time demands: Single crash

Chapter 6

Conclusion

In this chapter, I would like to summarize all the goals I had to achieve. Also, I would like to describe how I managed to achieve them. Then I would like to write about possible extensions of my application. At last, I would like to mention my attempt to use my framework in another simulation.

First of all, the application interface. I think it is quite simple. Everything that must be done to include the physics into another simulation is to include my project as an external module. Another simulation then communicates only with the agents implemented in my project. Only thing the simulation must provide is waypoints and desired speed of the vehicle.

Next goal was to create several types of vehicles. I implemented four types of vehicles. Types include motorbike, bus, common car and a three-wheeler. Each type of vehicle behave according to its set of parameters, but they share common interface. Therefore, they can be handled in the same way.

Separated visualization was another goal. As I wrote above, the visualization runs in a separated thread and can be disabled if necessary. But separated visualization means some additional omissible parameters. Because I did not want to synchronize the simulation with the visualization (it means additional overhead to the simulation) I had to create this interlayer between the simulation and the visualization parts.

Modularity and the Alite support are included as well. This goal was met from the beginning, because I based my project on the Alite software toolkit. The most indispensable part of the Alite simulation toolkit is the event processor with its event queue. I created my project as modular as I could. Almost every part of the simulation can be replaced or extended according to demands of concrete problem.

I also created some demos to show results of my work. They are stored on CD I included. I prepared three scenarios. The first of them is crash of a vehicle with a static object. The second shows ride of the car through prepared environment (in this case I implemented simple castle). The last one shows crash of two vehicles.

There are many things that could be extended in my application. In the first place in my mind is more complex regulator. The regulators I implemented are not bad, but I think that one complex regulator would be better than two separated regulators. I mean that the regulation of speed would be better if regulated in dependence of maneuver the vehicle is actually executing.

The second thing is texturing objects. It is minor extension, only stylistic change, but the visualization would look better.

Next thing that would be useful extension is determining exact set of parameters for each vehicle type. I am not an expert on vehicles so I tried to choose the parameters as good as I could. But I think there is a space for an improvement.

Last thing I have in mind is connecting my framework with some other application. I tried to connect my framework with another simulation. I almost succeeded, but there were some problems with the simulation. The simulation was badly designed and I was not able to use all the features provided by my framework. I was able to run single vehicle driving straight. But I was unable to run multiple vehicles. So, I was unable to test my framework properly with another application.

Bibliography

[1] *Alite software toolkit* [online]. 2011. [cit. 24. 4. 2011]. Available from: <http://merle.felk.cvut.cz/redmine/projects/alite/wiki/Alite_Overview>.

[2] *Bullet physics library* [online]. 2011. [cit. 24. 4. 2011]. Available from: <http://bulletphysics.org/wordpress/>.

[3] *EPA (Expanding Polytope Algorithm)* [online]. 2011. [cit. 11. 5. 2011]. Available from: <http://www.codezealot.org/archives/180>.

[4] *jBullet software toolkit* [online]. 2011. [cit. 24. 4. 2011]. Available from: <http://jbullet.advel.cz/>.

[5] *Raycast vehicle* [online]. 2011. [cit. 27. 4. 2011]. Available from: <http://tinyurl.com/ydfb7lm>.

[6] JIRáNEK, J. Non-cooperative agents for cars drive simulation, 2009. Diploma thesis.

[7] LINDEMANN, P. The Gilbert-Johnson-Keerthi distance algorithm, 2009. Research paper.

Chapter 7

List of abbreviations

API Application programming interface

2D Two-dimensional

3D Three-dimensional

6DOF Six degrees of freedom

IDE Integrated development environment

AABB Axis aligned bounding box

SIMD Single instruction, multiple data

GJK Gilbert, Johnson and Keerthi

EPA Expanding polythope algorithm

kmph Kilometres per hour

Chapter 8

Installation and user guide

In this chapter, I would like to describe usage of the framework as a standalone project and as a module in another simulation project. Also, I would like to describe how to run demos I included on CD.

8.1 Usage of the framework as standalone project

Using this framework is extremely simple. As the main class of the project the Main class from the Alite project is used. There are also several parameters that must be included when running the simulation from the command line. The first of them is the creator class. The second parameter is path to external XML file containing initialization parameters of the simulation.

When the user wants to run the simulation with the visualization, Java virtual machine parameter must be included. But because the OpenGL uses native libraries for Linux, Windows and MacOS, parameter differs in dependence of the operating system. When Linux is used, parameter looks like this: -Djava.library.path=libs/linux. When running on Windows parameter looks like this: -Djava.library.path=libs/win32. MacOS uses parameter like this: -Djava.library.path=libs/macosx.

8.2 Usage of the framework as a module

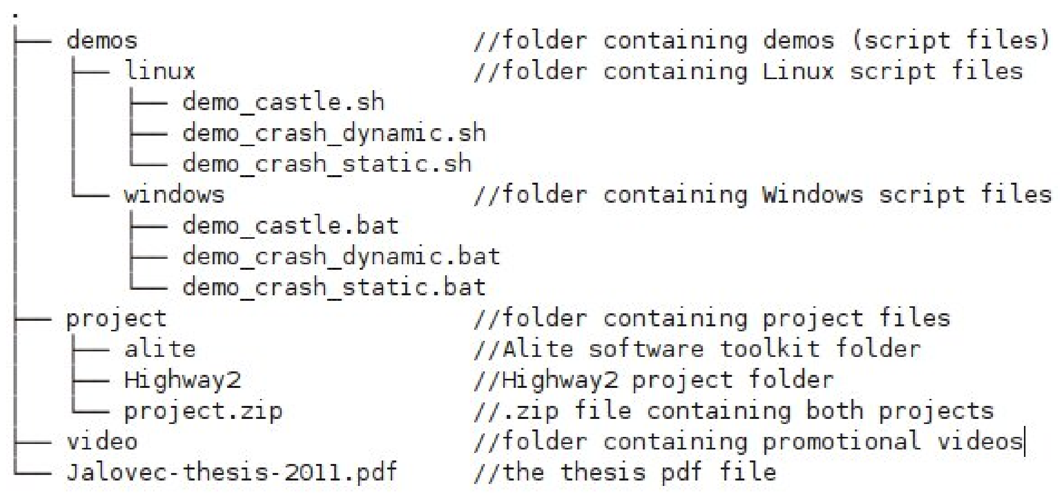
When the framework is used as a module in another simulation, it must be included as an external source with the parameters described above. Then the vehicles must be created along with their agents. And that is all. User's application communicates with this framework through the agents only. Application must generate waypoints with corresponding speed.

8.3 Running the demos

I prepared three demos to show some examples of the framework. Each demo has its batch file for Linux and Windows operating systems. To run the demo, simply run the file [demoName].bat on Windows and [demoName].sh on Linux.

Chapter 9

Contents of CD



```
.
├── demos                       //folder containing demos (script files)
│   ├── linux                   //folder containing Linux script files
│   │   ├── demo_castle.sh
│   │   ├── demo_crash_dynamic.sh
│   │   └── demo_crash_static.sh
│   └── windows                 //folder containing Windows script files
│       ├── demo_castle.bat
│       ├── demo_crash_dynamic.bat
│       └── demo_crash_static.bat
├── project                     //folder containing project files
│   ├── alite                   //Alite software toolkit folder
│   ├── Highway2                //Highway2 project folder
│   └── project.zip             //.zip file containing both projects
├── video                       //folder containing promotional videos
└── Jalovec-thesis-2011.pdf     //the thesis pdf file
```

Figure 9.1: CD content