Steering Multiple Agents for Autonomous Navigation in Virtual Environments

Johannes Zarl

August 26, 2010

Abstract

This work introduces the *ufo* steering library, a library for steering multiple agents within virtual environments using runtime-configurable behaviours. The approach used in this document allows for reuseability of steering behaviours and easy integration with existing virtual reality applications. This is demonstrated by providing an interface to the *inVRs* framework ([1]).

Contents

List of Tables

List of Figures

List of Listings

1 Introduction

Several decades have gone by since Reynolds wrote his seminal work on boids and the accompanying distributed behavioural model. Research in this field of research has not been standing still, and today virtually no computer animated film and hardly any computer game would be complete without flocking behaviours and steering algorithms.

However, surprisingly few general purpose (as in "not bound to a specific application") steering libraries exist. One notable example is the OpenSteer project [3], which, though not easily usable as a library, offers a complete set of steering algorithms as described by Reynolds [14].

One problem which possibly hinders development of a "standard steering library" is the need for flexible behaviours in an application. In order to be useful, such a library must allow application programmers to extend the library functionality, preferably without recompilation of the whole library.

The *ufo* library, which is presented in this work, addresses this problem by using a *plugin* system to manage extensions of the functionality, combined with *runtime configuration*. The runtime configuration aspect allows for developing complex behaviours from several simpler ones, without the need to recompile parts of an application in the process.

2 Concepts and Related Work

This section will give a short overview on the topics of flocking behaviours, steering concepts, and related topics. A terminology will be given, based on the distributed behavioural model described by Craig W. Reynolds [13]. This model stems from cinematographic computer animation and was developed as a means to animate large numbers of animals in a natural-looking way. Since its inception, behavioural models employing artificial intelligence have emerged and tested with success, but they do not scale well for large groups of agents[5].

2.1 Boids

The term *boid* originally refers to a "bird-like object", typically as a member of a larger group of similiar objects.

Historically, flocking boids are related to *particle systems*. Particle systems are often used to model fuzzy objects such as smoke, fire, clouds or water[10]. Both techniques use a large number of objects with individual behaviours. There are two main differences between boid flocks and particle systems: boids have complex geometries, whereas particles typically are 2-dimensional sprites without geometry. The other significant difference lies in the behaviours – while boid behaviours are neccessarily dependent on the states of other boids, particle behaviours are normally only dependent on internal state (age, color, etc.)[13].

A boid does not have to represent a bird in the literal meaning, but is used as a general term for any object exhibiting a similiar type of movement[13]. Therefore, a boid is an individual member of a flock, swarm, herd, or school of "animals". It can be thought of as a *social autonomous agent*, i.e. an autonomous agent which interacts with other agents of the same kind.

Traditionally, the interaction between boids is signified by three complementary component behaviours[13, 6]: *separation* (collision avoidance between flockmates), *cohesion* (flock centering or staying close to nearby flockmates), and *alignment* (velocity matching with nearby flockmates). Using these three behaviours, very convincing flocks of animals can be produced, though some patterns observed in nature (e.g. V-like formation flight) call for different approaches [9].

To avoid confusion, this work generally uses the word *flock* to denote a group of boids, regardless of which species is simulated. This term was chosen arbitrarily from the plethora of synonyms, and is also used in the *ufo* library for the same concept (see section 3.2 on page 8).

Even though the idea of boids is indeed very useful, in the *ufo* library several aspects of it are represented by different classes rather than one single boid-class. From the outside, i.e. from an application point of view, a boid is more or less equivalent to a `Pilot`. However, the inner workings of a boid are further encapsulated: its physical representation is embodied by the `Steerable` class, and its behavioural model is captured by the `Behaviour` class. All of these classes are discussed in detail in section 3 on page 7. A benefit of this separation

is that the same steering behaviours can be used with different vehicles [6].

2.2 Layers of Behaviour

The behaviour of an autonomous agent encompasses many things: from basic character animation and movement to advanced goal-oriented action and cognition every act by the agent is governed by a behaviour. A classification of behaviours into several layers helps to find meaningful ways of combining and selecting multiple behaviours.

Reynolds decomposes motion behaviours for autonomous agents into three layers[14]: *Action Selection* as the highest layer refers to strategy, goal making and planning, i.e. *what to do*. The middle layer, *Steering*, pertains to path determination, i.e. *where to move*. Finally, the lowest layer is *Locomotion*, which encompasses animation and articulation of a character, i.e. *how to move*. In this work, steering is used in the sense of path determination.

In the *ufo* library, the steering layer of this model corresponds to the `Behaviour` classes. The locomotion layer is dealt with in the `Steerable` classes. Action selection is outside the scope of this work.

A similiar approach is taken by Blumberg and Galyean[2]: They subdivide behaviours according to the level of autonomy required to achieve their associated goals, called the *Level of Input*. Three of these levels are identified: the *Motivational Level*, the *Task Level*, and the *Direct Level*. The final action is a result of the blending of different behaviours originating at these three levels.

Blumberg and Galyean focus on the blending of potentially competing behaviours and aim to create a complete behavioural simulation of a virtual animal/pet[2]. Reynolds focuses on the encapsulation between different layers, which allows using the same higher level behaviours for different animals or means of locomotion, and then discusses different steering behaviours in detail[14]. Further work on action selection has been done by Funge, Tu and Terzepoulos[4].

Any reasonably complex autonomous agent consists of several different behaviours, which influence the agent to varying degrees. Several techniques exist to switch between these behaviours and blend them into a meaningful ensemble. Blending can simply calculate the sum of all behaviours[6], or employ more elaborated schemes, such as the (weighted) average[14], or mutual inhibition between behaviours[2]. Switching between behaviours can either be assigned to the action selection layer [14], or be achieved using a weighted input for blending.

In order to give the application developer the opportunity to use the blending technique which suits his or her application, behavioural selection and blending is itself implemented as a behaviour in the *ufo* library: A boid has exactly one behaviour, but a behaviour can have child-behaviours. In this manner, a hierarchy of behaviours is created (see 3.3.1 on page 9).

3 Implementation

ufo is a single-threaded steering library. It can steer one or more independent entities (via `Pilots`), which can be optionally grouped to form `Flocks`. Figure 1 gives an overview over *ufo*'s core classes. As can be seen, an application can access the `UfoDB` (see section 3.6 on page 13), and via it the `Flocks` and `Pilots`. The `Behaviour` and `Steerable` of a `Pilot` is not accessible from the outside.

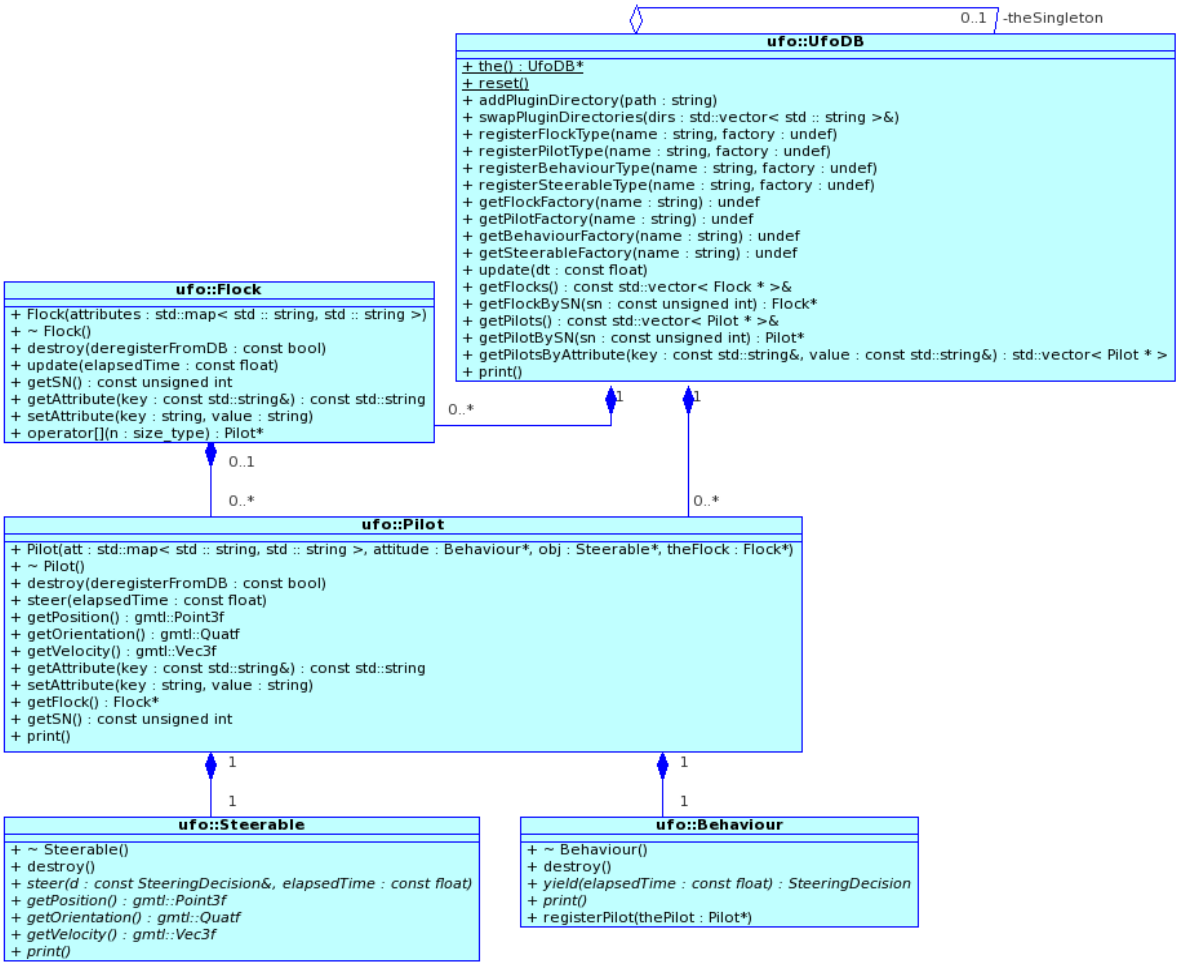

Figure 1: Core classes in the *ufo* library

3.1 Pilot

From an application point-of-view, the `Pilot` corresponds to Reynold's concept of a boid – it has a position and a speed[1], and it (optionally) belongs to a `Flock`.

[1]Within the scope of this document, *speed* always denotes the speed vector, not its magnitude.

The `Pilot`'s `Behaviour` and its `Steerable` define how the `Pilot` acts and which application object it moves around, but both are themselves invisible to the application. Figure 2 is a collaboration diagram of the *ufo* update phase. It shows how the `Pilot` consults its `Behaviour` to reach a steering decision and then uses the `Steerable` to steers an associated object in the application's domain. The base implementation of `Pilot` simply uses the result of `Behaviour::yield()` and feeds it to the method `Steerable::steer(SteeringDecision)`.

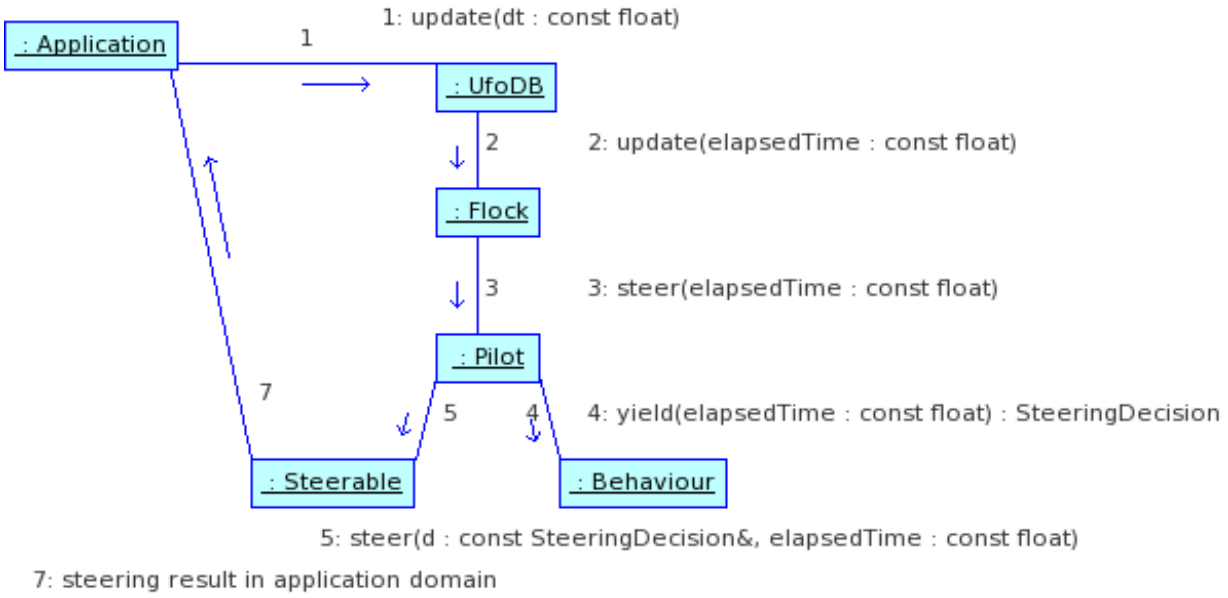

Figure 2: Collaboration diagram of *ufo* components during an update

Although one can inherit from `Pilot`, the base implementation is sufficient for most use-cases and implements all functionality needed by the supplied `Behaviours` and `Steerables`.

3.2 Flock

Whenever two or more `Pilots` should behave as a group, depending on each others position, and speed, they have to belong to the same `Flock`. Membership in a `Flock` allows each `Pilot` to query the `Flock` for other `Pilots` without the need for an exhaustive search over all `Pilots` in the `UfoDB`.

The supplied base implementation of `Flock` is essentially a vector of Pilots, but can be extended to provide more advanced features, like querying by spatial neighborhood etc.

A `Pilot` can only belong to one `Flock` at a time.

3.3 Behaviour

In order to keep the `Pilot` simple, it does not compute the steering decision itself, but uses the class `Behaviour`.

The `Behaviour` class can access the `Pilot` and, if available the `Flock` (and by this means other `Pilots` of the same `Flock`). The typical `Behaviour` would thus base its steering decision upon its `Pilot`'s position and speed, on the position and speed of its neighbors, and on an internal target speed and position.

The class `SteeringDecision` combines a vector describing the desired speed, called *direction*, and a quaternion describing the desired *rotation*. An implementation is allowed to leave either or both components unset. For simplicity's sake the rest of this document will only discuss the desired speed vector, where the rotation is of no immediate importance.

The steering decision as computed by the `Behaviour` represents the desired speed (and rotation). Constraints like a maximum speed value or a maximum applyable force are added afterwards by the `Steerable`.

3.3.1 Stackable Behaviours

With more advanced behavioural patterns, the code complexity of the class implementing a `Behaviour` quickly increases to a level at which changing and debugging the code becomes prohibitively difficult. For this reason, the idea of stackable behaviours is proposed.

In contrast to the aforementioned typical `Behaviour`, a stackable `Behaviour` does not itself compute a steering decision. Instead, it consults one or more child-behaviours, and combines and modifies their output.

Using this approach complex behaviours can be developed by combining several simpler behaviours. These simple behaviours can be easier modified and tested.

Another advantage of stackable behaviours lies in the flexible blending between child-behaviours. The application developer can create a hierarchy of behaviour trees. Some of these may always contribute towards the final steering decision, while other behaviours or entire subtrees can be turned off [2].

3.4 The Steerable Interface

One design goal of *ufo* is to be usable as an add-on library. For this reason a clean split between the steered entity inside the application domain and the `Pilot` was chosen. Instead of forcing an application-writer to inherit the steered object from a generic one (as found in [3]) an adaptor-class (`Steerable`) is used as an interface between the application and the *ufo*-library.

The main implication of this approach is that the steered vehicle is updated independently of the `Pilot`, i.e. the update-rate of the steering decisions is independent of the update-rate of the simulation.

[2] In this manner, *action selection* could be implemented easily and without architectural changes to the *ufo* library.

3.4.1 Communication between Pilot and Steerable

To set the desired velocity (and orientation), the `Steerable` interface provides the method `Steerable::steer(SteeringDecision)`. What to do with this information is up to the class adhering to the `Steerable` interface. A simple implementation could just set the object speed to the given value, while more sophisticated implementations could apply physical restraints on the object.

In return the `Pilot` can ask for the current position, velocity and orientation of the `Steerable` using the method `Steerable::getPosition()`, `Steerable::getVelocity()`, and `Steerable::getOrientation()`, respectively.

3.4.2 Communication between Steerable and Vehicle

As mentioned before, a direct implication of the adaptor-approach is that updating the vehicle is decoupled from updates in libufo. This allows the communication between `Steerable` and vehicle to be relatively flexible.

If one's application already has a vehicle implementation, the `Steerable`-implementation only has to translate the function calls accordingly, converting data types as needed. The factory method then has to couple `Steerable` objects with vehicle objects. For an example, see section 4.1 on page 16.

3.5 From Configuration File to Flock

ufo uses a three step setup to initialise `Flock`s and `Pilot`s: first a `ConfigurationReader` is created for the configuration file; second the configuration file is parsed and a `Configurator` is created; at last, the `UfoDB` is populated by the `Configurator`. Figure 3 on the next page demonstrates these steps.

3.5.1 Creating a ConfigurationReader

A `ConfigurationReader` implements the parsing of a configuration file or other configuration resource. It provides a `ConfigurationReader::readConfig()` method, which returns a `Configurator` object.

If an application writer decides to configure *ufo* in-band with his application's configuration, he can extend the *ufo* library with his own `ConfigurationReader` in a clean way.

The default implementation for `ConfigurationReader`, `PlainConfigurationReader` uses plain text files containing configuration data in the format depicted in table 1 on page 12. Currently, `PlainConfigurationReader` is the only implementation for this interface.

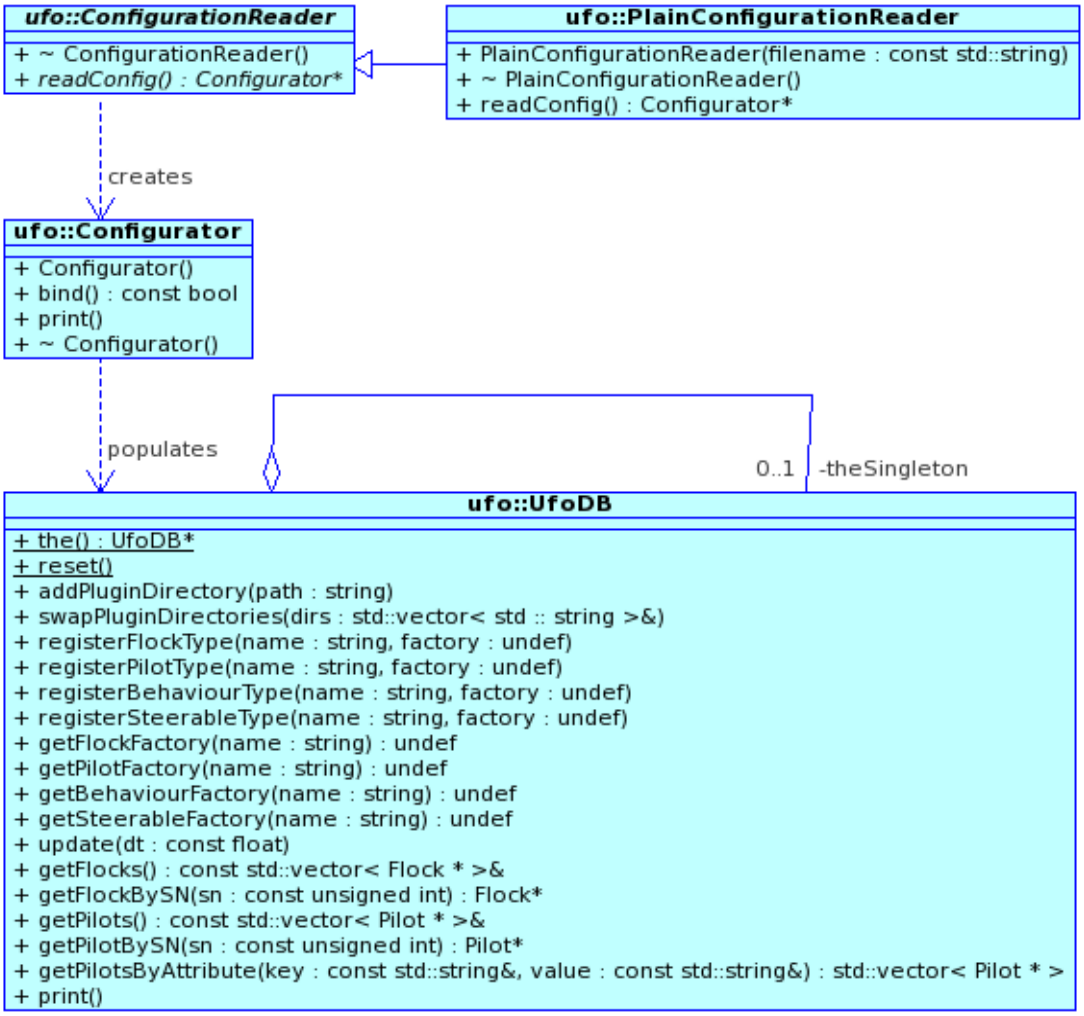

Figure 3: Classes used for configuration

```
UFOCFGPLAIN = "UFOCFGPLAIN" , { Flock | Pilot | Directive | Comment } ;

Flock = StorageModifier , "FLOCK" , Identifier , "{" ,
        { Parameters | Flock_Children }
        "}" ;
Pilot = StorageModifier , "PILOT", Identifier , "{" ,
        { Parameters | Pilot_Children }
        "}" ;
Directive = "pluginDirectory" , Path ;
Comment = "/*" , { Identifier } , "*/" ;

StorageModifier = "immediate" | "fromTemplate" | "template" ;
Identifier = letter , { letter | decimal digit | "_" | "-" } ;
Parameters = "parameters" , "{" ,
             { ( Identifier , "=" , String ) | Comment } ,
             "}" ;
Flock_Children = "children" , "{" , {Pilot} , "}" ;
Pilot_Children = "children" , "{" , { Steerable | Behaviour } , "}" ;
Path = [ [ "." ] , "/" ] , Identifier , { "/" , Identifier } ;

String = Identifier | ( '"' , Identifier , { " " Identifier } , '"' ) ;
Steerable = StorageModifier , "STEERABLE" , Identifier ,
            "{" , { Parameters } "}" ;
Behaviour = StorageModifier , "BEHAVIOUR" , Identifier ,
            "{" , { Parameters | Behaviour_Children } "}" ;

Behaviour_Children = "children" , "{" , { Behaviour } , "}" ;
```

Table 1: Extended Backus-Naur Form[7] grammar of *ufo* plain configuration format

```
static void addPluginDir(
  Configurator *cfg,
  std::string &path ) ;

static void addElement(
  Configurator *cfg,
  ConfigurationElement *elem ) ;

static void setTemplate(
  Configurator *cfg,
  std::string &name,
  ConfigurationElement *elem ) ;
```

Table 2: Functions in ConfigurationReader implementing the communication protocol with the Configurator

3.5.2 Going from ConfigurationReader to Configurator

An implementation of `ConfigurationReader` always creates a `Configurator` and populates it with `ConfigurationElement`s, templates and additional information (e.g. path information for plugins).

For this purpose, it has to access and modify private data of the `Configurator`, and therefore has to be a friend class of it. As friendship is not inherited, no subclasses of `ConfigurationReader` can directly access the `Configurator`.

For this reason the protocol between `ConfigurationReader` and `Configurator` is fully implemented as protected functions in the abstract base class `ConfigurationReader`. Any implementation has to use these functions (see table 2) to communicate with the `Configurator`.

By calling the method `Configurator::bind()`, the configuration is finalised and then used to populate the `UfoDB`.

3.6 UfoDB

The `UfoDB` is the central database for all objects created by *ufo*. Its interface and its relation to the other core classes of *ufo* can be seen in figure 1 on page 7.

All `Flock`, `Pilot`, `Behaviour`, and `Steerable` types must provide a factory function, which has to be registered with the `UfoDB` before they can be used. This registration can be done either manually, using the provided registration-methods, or automatically via the plugin mechanism outlined in section 3.7.

Once the `UfoDB` has been populated by the `Configurator`, methods are provided to access individual `Flock`s and `Pilot`s.

Once per simulation-step, one should call `UfoDB::update(dt)`, stepping the simulation

by the given amount of time for every `Flock` and `Pilot`.

3.7 Plugin System

Factory functions for every type of `Flock`, `Pilot`, `Behaviour` or `Steerable` are stored in the `UfoDB`. Whenever a factory function is requested (by the `Configurator::bind()` method) that is not already registered, plugin loading is triggered (see figure 4), and an appropriate plugin file is searched in the plugin directories specified in the configuration file.

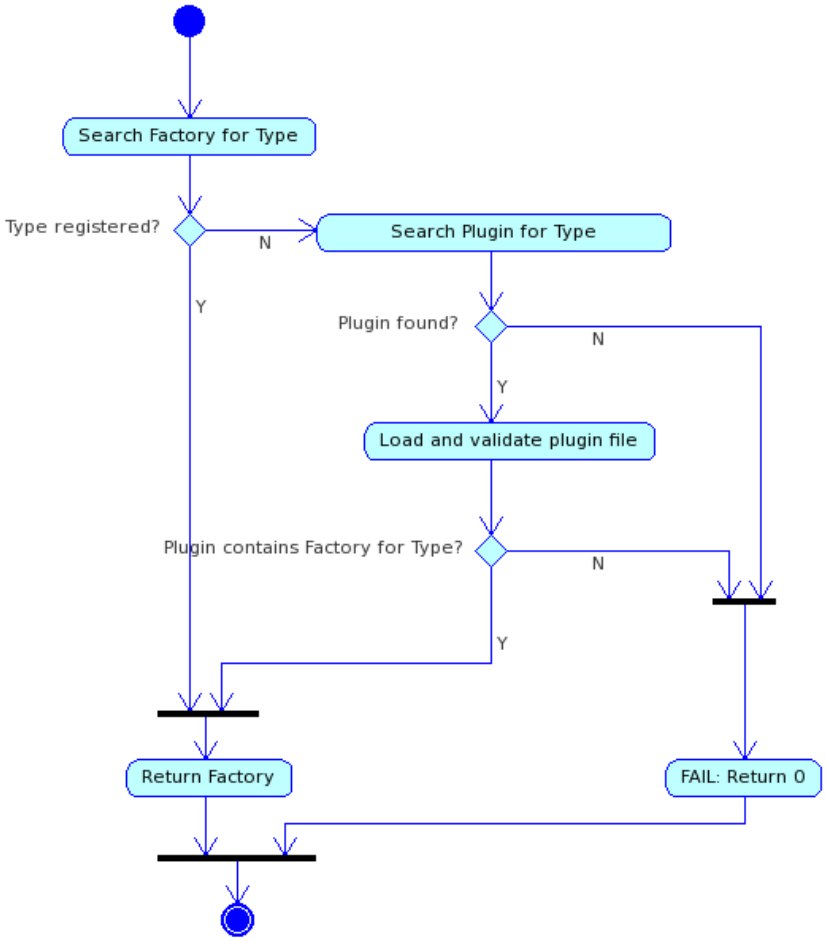

Figure 4: Factory function lookup triggers loading of a plugin

Plugins are used in *ufo* to extend functionality in a transparent way. Without the automatic plugin-loading, an application would need to know before loading its configuration which types are used. Thus, plugins are a necessary means to enable full runtime-configurability in applications.

A plugin file is a shared library that provides several `C` functions which are used to check compatibility with the plugin and register its factory function. A `C` preprocessor macro `MAKEPLUGIN` is provided for the developer, which automatically defines the neccessary functions. Any class deriving from one of the classes `Steerable`, `Behaviour`, `Pilot`, or `Flock` is expected to use the namespace `ufoplugin`, if it uses the `MAKEPLUGIN` macro.

Security implications: It is possible to inject untrusted code into an application using the plugin system. The confinement of plugin search to the given directories does reduce the possibility of local exploits using this attack vector.

Other implications: Because any class loaded via a plugin is not known to the application, it is possible that the operator `delete` of the plugin and the application doesn't match[8]. For this reason, a `destroy` method is suppllied for those classes which can be loaded via plugins. Should it be neccessary for an application to explicitly destruct an object created by the plugin system, this method has to be used.

4 Integration with Existing Projects – *inVRs*

In this section integration of the *ufo* library with an external project is demonstrated. For this, the *inVRs* framework[1] was chosen. *inVRs* is an application framework which aims to facilitate development of networked virtual environments.

4.1 Steering an Application Object

The *inVRs* framework organises objects in two databases: a *UserDatabase* for the local user object and information about remote users, and a *WorldDatabase* for all other in-world objects. Objects with which the user can interact are called *Entities*. In this chapter, a custom `Steerable` class called `InVRsSteerable` will be created, which applies steering decisions of the *ufo* library to such an Entity and translates position, orientation and velocity data of the Entity back into the *ufo* library.

It will be shown how parameters can be passed from the configuration file to this `InVRsSteerable`. Furthermore the example will show how additional restrictions can be enforced at the `Steerable`-level. This will be done by limiting the speed value of an object to a given maximum value.

Listing 1 shows the interface for the `InVRsSteerable` class. The class uses the namespace `ufoplugin`, so it can use the `MAKEPLUGIN` macro (see section 3.7).

In this example, the values for position, velocity and orientation are cached in private member variables of the `InVRsSteerable` class, and updated once during each simulation step inside the `steer` method. The associated getter functions simply return the cached value.

A factory function called `InVRsSteerable` is declared, which is supplied with a vector of string-pairs containing the key-value pairs supplied within the configuration file. The factory function is expected to use these values to create a suitable Steerable and return it to its caller. If a suitable Steerable can not be created (e.g. due to wrong or missing parameters), a *null-pointer* is to be returned.

```
1  namespace ufoplugin {
   class InVRsSteerable
     : public ufo::Steerable
   {
     public:
6      InVRsSteerable( Entity *ent, const float VMax );
       virtual ~InVRsSteerable();
       void steer( const ufo::SteeringDecision& d,
                   const float elapsedTime);
       gmtl::Point3f getPosition();
11     gmtl::Quatf getOrientation();
       gmtl::Vec3f getVelocity();
       void print() const;
```

```
    private:
      Entity *myEntity; // pointer to steered Entity
      float VMax; // maximum speed value
      gmtl::Point3f position;
      gmtl::Quatf orientation;
      gmtl::Vec3f velocity;
  };

  ufo::Steerable *InVRsSteerableFactory ( std::vector<std::pair
    <std::string,std::string> > *params);
}
```

Listing 1: Interface of the InVRsSteerable class

Listing 2 shows in detail, how this is done. Entities in *inVRs* can be uniquely identified via an unsigned integer called the *environmentBasedId*. Therefore, an accordingly named entry is searched in the configuration parameters and its value is converted from string and saved into a variable. The same is done with the speed limit parameter called *VMax*.

The id of the entity is then used in line 24 to query the WorldDatabase for the correct Entity, which in turn is used to initialise the `InVRsSteerable`.

```
ufo::Steerable *ufoplugin::InVRsSteerableFactory ( std::vector<
   std::pair<std::string,std::string> > *params)
{
  Entity *ent=0;
  float VMax = -1.0f;
  unsigned int environmentBasedId=0;
  for ( std::vector<std::pair<std::string,std::string> >::
     const_iterator it = params->begin();
       it != params->end(); ++it)
  {
    if ( it->first == "environmentBasedId" )
    {
      stringstream value;
      value.str(it->second);
      value >> environmentBasedId;
    }else if ( it->first == "VMax" )
    {
      stringstream value;
      value.str(it->second);
      value >> VMax;
    }else {
      cout << "WARNING: unknown parameter to InVRsSteerable: "
        << it->first <<endl;
```

```
    }
  }
  // get steered Entity from inVRs WorldDatabase:
  ent = WorldDatabase::getEntityWithEnvironmentId( environmentBasedId);
  if (ent)
  {
    // create \ufoclass{Steerable}
    return new InVRsSteerable( ent, VMax );
  }
  cout << "ERROR: WorldDatabase doesn't have an entity with environmentBasedId = " << environmentBasedId << " )!" << endl;
  return 0;
}
```

Listing 2: Factory function for `InVRsSteerable`

Once the `InVRsSteerable` has been created, all the work is done in the method `steer` (see listing 3). Because there is no notion of speed in *inVRs*, the velocity of an object has to be maintained inside the `InVRsSteerable`. For the sake simplicity, no further limits other than a maximum speed is enforced in this example.

In *inVRs* in a `struct` called `TransformationData` is used to hold the position and orientation of an object. Moving the `Entity` is simply a matter of getting its associated `TransformationData`, applying the position and orientation changes, and then setting the new `TransformationData` for the `Entity`.

```
void ufoplugin::InVRsSteerable::steer( const ufo::SteeringDecision &decision, const float elapsedTime)
{
  velocity = decision.direction;
  // are we faster than VMax?
  if ( VMax >= 0.0 && lengthSquared(velocity) > VMax*VMax )
  {
    // scale Velocity to VMax:
    normalize(velocity);
    velocity *= VMax;
  }
  // write transformationData:
  TransformationData td = myEntity->getWorldTransformation();
  td.position += velocity * elapsedTime;
  myEntity->setWorldTransformation(td);

  // update current Steerable position
```

```
    position = td.position;
18  orientation = td.orientation;
}
```

Listing 3: `InVRsSteerable::steer()` – Moving an *inVRs*Entity and updating position, speed and orientation

4.2 Following a User in the Application

In this section a `Behaviour` will be created, which is able to follow a user inside the *inVRs* framework. As previously mentioned, users are stored by *inVRs*inside the *UserDatabase*. In contrast to entities, users can connect and disconnect at any time. A callback mechanism is provided by *inVRs* as a means to react to these events. Combined with the multithreaded nature of *inVRs*applications this calls for inter-thread-synchronisation.

Listing 4 shows the interface of the `FollowInVRsUserBehaviour` class. For inter-thread synchronisation, the `User` object pointer, as well as two synchronisation variables are marked as `volatile`. One callback object of each type `AbstractUserConnectCB` and `AbstractUserDisconnectCB` are defined by *inVRs*.

```
1  namespace ufoplugin
   {
     class FollowinVRsUserBehaviour : public ufo::Behaviour
     {
       public:
6        FollowinVRsUserBehaviour(const std::string name, const
            std::string pilotLink, const bool verbose);
         virtual ~FollowinVRsUserBehaviour();

         ufo::SteeringDecision yield ( const float elapsedTime);
         void print () const;
11
         void registerUser ( User *u);
         void unregisterUser ( User *u);
       private:
         const std::string username;
16       volatile User *theUser;
         volatile bool crit_readUser;
         volatile bool crit_writeUser;
         // variables to store callback objects:
         AbstractUserConnectCB *ucCB;
21       AbstractUserDisconnectCB *udCB;

         User *getUserFromDB() const;
```

```
        // helper-function to switch theUser threadsafely:
        void setUser(User *u);
    };

    ufo::Behaviour *FollowinVRsUserBehaviourFactory(std::vector<
        ufo::Behaviour *> children, std::vector<std::pair<std::
        string,std::string> > *params);
}
```

Listing 4: Interface of the `FollowInVRsUserBehaviour` class

Listing 5 shows the factory function. A `FollowInVRsUserBehaviour` is *non-stackable*, therefore any child-`Behaviour`s are ignored. The list of key-value pairs is searched for the key *username*, and the `FollowInVRsUserBehaviour` is created.

```
ufo::Behaviour *ufoplugin::FollowinVRsUserBehaviourFactory(std
    ::vector<ufo::Behaviour *> children, std::vector<std::pair<
    std::string,std::string> > *params)
{
  string username;
  for ( std::vector<std::pair<std::string,std::string> >::
      const_iterator it = params->begin();
      it != params->end(); ++it)
  {
    if ( it->first == "username" )
    {
      cout << "InVRsSteerable parameter: username = " << it->
          second <<endl;
      username = it->second;
    }else {
      cout << "WARNING: unknown parameter to 
          FollowinVRsUserBehaviour: " << it->first <<endl;
    }
  }
  if ( username.empty() )
  {
    cout << "ERROR: FollowinVRsUserBehaviour: parameter '
        username' must be set!" <<endl;
    return 0;
  }
  return new FollowinVRsUserBehaviour(username);
}
```

Listing 5: Factory function for `FollowInVRsUserBehaviour`

The constructor of the `FollowInVRsUserBehaviour` class (listing 6) first initialises the two callback objects with the appropriate functions. Then, the `UserDatabase` is searched for a suitable user. It should be kept in mind that during the initialisation phase of *ufo*, remote users are unlikely to be connected yet. After this, the appropriate callback is registered. The `UserDatabase` serialises callback execution, although different callbacks may be executed concurrently. By only registering one of the two callback objects, the synchronisation code can be simplified.

```
ufoplugin::FollowinVRsUserBehaviour::FollowinVRsUserBehaviour
    (const std::string name)
: username(name), theUser(0), crit_readUser(false),
  crit_writeUser(false),
  ucCB( new UserConnectCB<FollowinVRsUserBehaviour>(this,&
    ufoplugin::FollowinVRsUserBehaviour::registerUser) ),
  udCB( new UserDisconnectCB<FollowinVRsUserBehaviour>(this,&
    ufoplugin::FollowinVRsUserBehaviour::unregisterUser) )
{
  theUser = getUserFromDB();
  if ( theUser )
  { // user found
    UserDatabase::registerUserDisconnectCallback( *udCB );
  } else { // user not connected (yet)
    UserDatabase::registerUserConnectCallback( *ucCB );
  }
}
```

Listing 6: Constructor for FollowInVRsUserBehaviour

Listing 7 shows how a user is queried from the `UserDatabase`. The username *localUser* is handled specially by always denoting the local user (even when there are remote users of the same name and the local user uses a different name).

```
User * ufoplugin::FollowinVRsUserBehaviour::getUserFromDB ()
  const
{
  // local or remote User?
  if ( username == "localUser" || username == UserDatabase::
    getLocalUser()->getName() )
  { // -> localUser:
    return UserDatabase::getLocalUser();
  }
  // -> remote User
  for ( int i=0; i<UserDatabase::getNumberOfRemoteUsers(); i++)
  { //search remoteUsers
```

```
    User *u = UserDatabase::getRemoteUserByIndex(i);
    if ( u->getName() == username )
      return u;
  }
  // no user found:
  return 0;
}
```

Listing 7: Method `FollowInVRsUserBehaviour::getUserFromDB()`

The two methods `registerUser()` and `unregisterUser()` are called from outside of the *ufo* simulation thread by means of the callback mechanism provided by `UserDatabase`. If no user is set, and a user connects, then the register method is called. If the user matches the preset *username*, it is set as *theUser* and the callbacks are swapped. The unregister method works analogously.

Within both methods, no members of the class may be accessed in a non-threadsafe way. For this purpose, the synchronisation steps to change the user have been grouped in a separate method `FollowInVRsUserBehaviour::setUser()` (listing 9).

```
void ufoplugin::FollowinVRsUserBehaviour::registerUser (User *
    u)
{
  if ( u->getName() == username ) // "our" User?
  {
    setUser(u);
    // switch callback:
    UserDatabase::unregisterUserConnectCallback( *ucCB );
    UserDatabase::registerUserDisconnectCallback( *udCB );
  }
}

void ufoplugin::FollowinVRsUserBehaviour::unregisterUser (User
    * u)
{
  if ( u == theUser ) // "our" User?
  {
    setUser(0);
    // switch callback:
    UserDatabase::unregisterUserDisconnectCallback( *udCB );
    UserDatabase::registerUserConnectCallback( *ucCB );
  }
}
```

Listing 8: Callbacks for (un)registering a User upon (dis)connect

The following scheme has been chosen for synchronisation between the the `setUser()` and `yield()` methods: before accessing the volatile *theUser* variable, `yield()` sets a read-lock an then checks if a write lock has been set. If this is the case, `yield()` exists its critical section without accessing the variable (listing 10). Similiarly, the `setUser()` method first sets a write-lock, and then waits until the read-lock has been released. After this, *theUser* is written and the write lock is released. The code works under the assumption that a compiler does not reorder reads and writes to `volatile` variables. While not applicable as a general-purpose mutex mechanism, this algorithm has the advantage of being portable and not being dependent on external libraries.

```
void ufoplugin::FollowinVRsUserBehaviour::setUser (User *u)
{
  crit_writeUser = true;
  // wait for yield to leave its critical section:
  while ( crit_readUser )
    ; // busy wait
  theUser = u;
  // leave our critical section:
  crit_writeUser = true;
}
```

Listing 9: Method `FollowInVRsUserBehaviour::setUser()`

The only thing missing now in the `FollowInVRsUserBehaviour` is the `yield()`, which is can be seen in listing 10. As discussed in the previous paragraph, access to the user object is restricted to the critical section of the method. Because the user object is `volatile`, a `const_cast` is needed to make the returned value of `getWorldUserTransformation` non-volatile. Once the `TransformationData` has been read, computing the correct heading is trivial.

```
ufo::SteeringDecision ufoplugin::FollowinVRsUserBehaviour::
   yield ( const float elapsedTime)
{
  // enter critical section:
  crit_readUser = true;
  if ( !crit_writeUser && theUser )
  {
    TransformationData td = const_cast<User *>(theUser)->
       getWorldUserTransformation();
    // leave critical section:
    crit_readUser = false;

    SteeringDecision retval (
        td.position - myPilot->getPosition(),
```

```
        td.orientation );
    return retval;
  } else {
    // leave critical section:
    crit_readUser = false;
    // no User available.
    return SteeringDecision();
  }
}
```

Listing 10: Method `FollowInVRsUserBehaviour::yield()`

4.3 Integration with an Application

In order to use *ufo* in an application, it is sufficient to make two modifications to the source code: first, the `UfoDB` needs to be initialised, and second, the `UfoDB::update()` method has to be called regularily (normally in the display-loop). Listing 11 shows the code necessary to initialise `UfoDB` wrapped in a simple function.

```
void initUfo(std::string configFile)
{
  ConfigurationReader *cfgReader=0;
  Configurator *cfg=0;

  cfgReader = new PlainConfigurationReader( configFile );
  cfg = cfgReader->readConfig();

  // we don't need the cfgReader any more:
  delete cfgReader;

  if ( cfg )
  {
    if ( cfg->bind() )
      std::cout << "initUfo(): initialisation succeeded" <<
          endl;
    // Configurator is no longer needed:
    delete cfg;
  }
}
```

Listing 11: Initialising the `UfoDB`

5 Results

In section 4 iS has been described that *ufo* can be integrated easily with an existing project. This section will look at possible performance bottlenecks and general performance considerations.

For example, figure 5 shows two flocks of the same size, but with a different set of behaviours. The complex swarm uses some boid-behaviours (alignment, cohesion, separation) twice with different scale-factor and flock-neighbourhood parameters. The result is a less uniform, more naturalistic looking flock behaviour compared to the same flock without the additional behaviours. The other, simplified flock does not use any boid-behaviour more than once. Instead, the randomized component of the behaviour has been given more weight. This has the effect of disturbing the uniform motion created by the boid-behaviours, thus also leading to a naturalistic looking flock.

Figure 5: A flock using 100 boids, using complex(left) or simplified behaviours (right)

Figure 6 on page 27 shows the decrease in performance with increasing flock size. All measurements have been done on an Intel® Core™ 2 Quad at 2.5 GHz with 6 GByte of RAM, running Debian linux. The measurements were taken without graphics output using a single thread (*ufo* currently has no threading capabilities). Not surprisingly, the simple flock consistently shows almost double the framerate compared with the complex flock. Still, as the boid-behaviours, as implemented in *ufo*, have an asymptotic complexity of $O(n^2)$, this doubled framerate only buys a few boids worth of performance, before the framerates collapses.

This problem can be mitigated using spatial databases supporting *neighbourhood query*, ideally leading to an asymptotic complexity of $O(n)$[11]. However, the (fixed) overhead of

a spatial database can make the problem *worse* for smaller flocks, and finding the correct setup parameters for the database needs to be done again for each new application. The spatial database approach is therefore best suited for deployment during towards the end of a project, when the approximate space distribution of the flock is known.

Another source for performance gains is the decoupling of boid-behaviours from the general update-rate[11]. This decoupling is achieved in the *ufo* behaviour `CachingBehaviour` by caching the steering decision until the aggregated time reaches the cache time limit *deltaTMin*. The third line in figure 6 on the following page shows the effects of this decoupled behaviour with a time limit of 40 ms, which corresponds to a framerate of 25 frames-per-second. In this case the caching does not perform better than the initial complex behaviour for flock sizes bigger than 200 boids.

This fact is easily explained by looking at the initial behaviour without caching: at 200 boids, the performance drops below 25 fps. Thus, the cached steering decision is invalidated before its calculation has finished.

Figure 7 on the next page shows the effect of different caching intervals on the framerate (shown as logarithmic scale). The underlying behaviour is the same complex behaviour as in the example shown above, with 200 boids in one flock. Without caching, the achievable framerate is around 14 fps. This means that one update cycle takes around 71 ms to complete. Unsurprisingly, this coincides with the performance drop apparent in the graph.

For this reason, the minimum time for caching δt_{min} must follow the formula

$$\delta t_{min} \geq \frac{1}{\texttt{fps}}$$

in order for caching to be effective. Reynolds reports simulation framerates as low as 10 fps to be sufficient for flocking to appear fluid[11], so in the example above a value of 80 ms for δt_{min} would be a suitable choice.

Caching is most effective when using multiple flocks: As long as each flock is small enough to benefit from caching at the desired framerate, a superlinear speedup can be observed. Figure 8 on page 28 shows the framerate of multiple independent flocks with 100 boids each. The scale is logarithmic, and the behaviours are the same as in the previous examples.

As we know from figure 6, the flock size of 100 boids is well below the "breakdown size" at the cache interval of 40 ms. For both behaviours, the framerate decreases linearly with the number of flocks, but the caching behaviour allows the system to run at framerates well beyond 1000 fps. It should be noted however, that in the observed dataset the speedup decreases linearly with an increased number of flocks (figure 9).

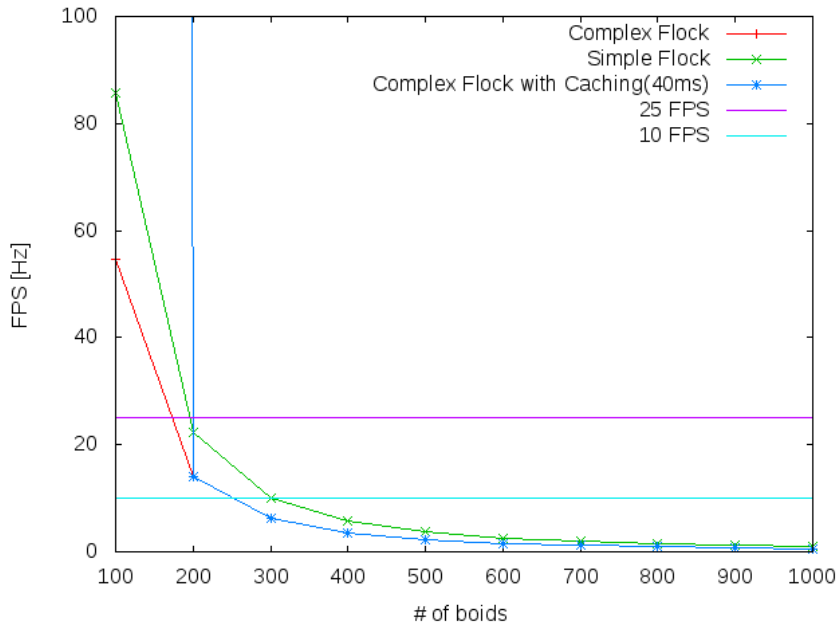

Figure 6: Framerates of different behaviours at varying flocksizes

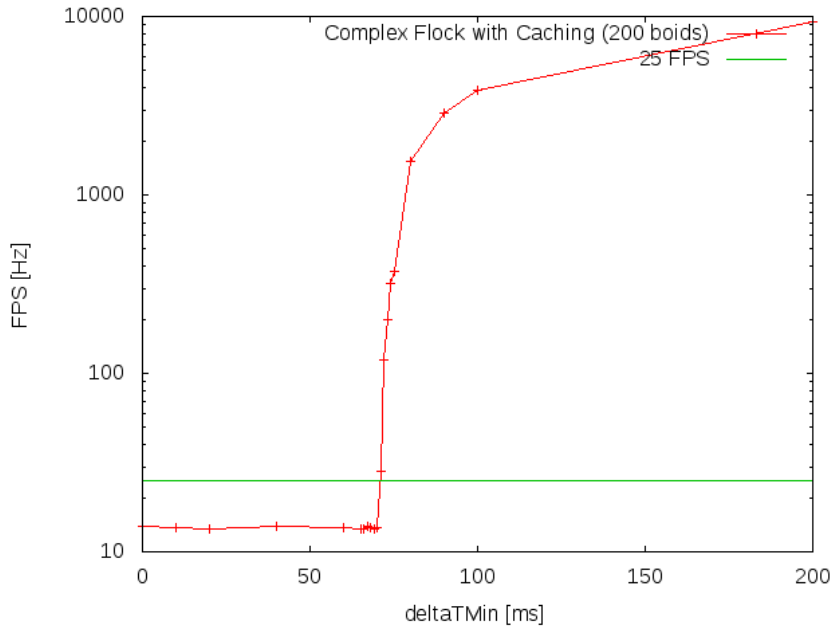

Figure 7: Effects of different caching intervals on the framerate

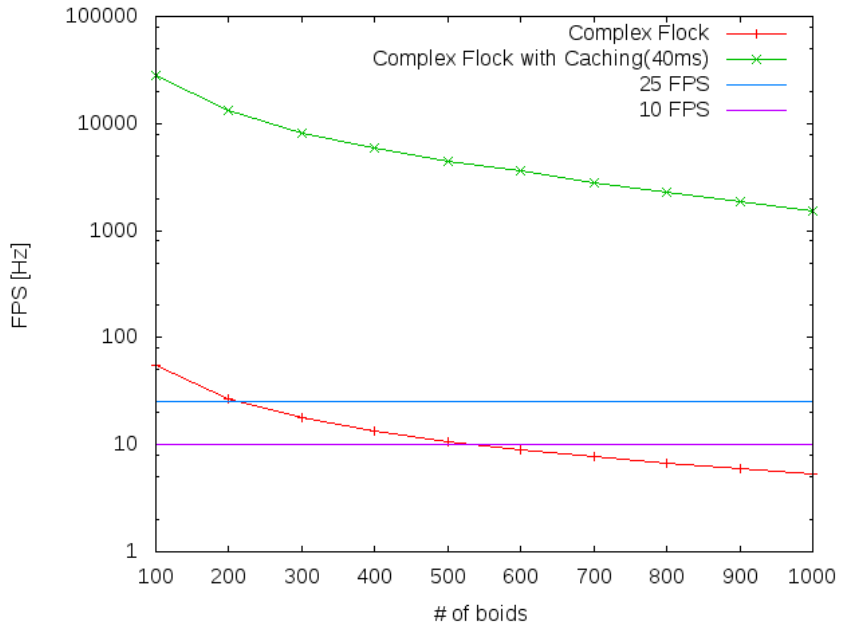

Figure 8: Performance gains of caching with multiple small flocks

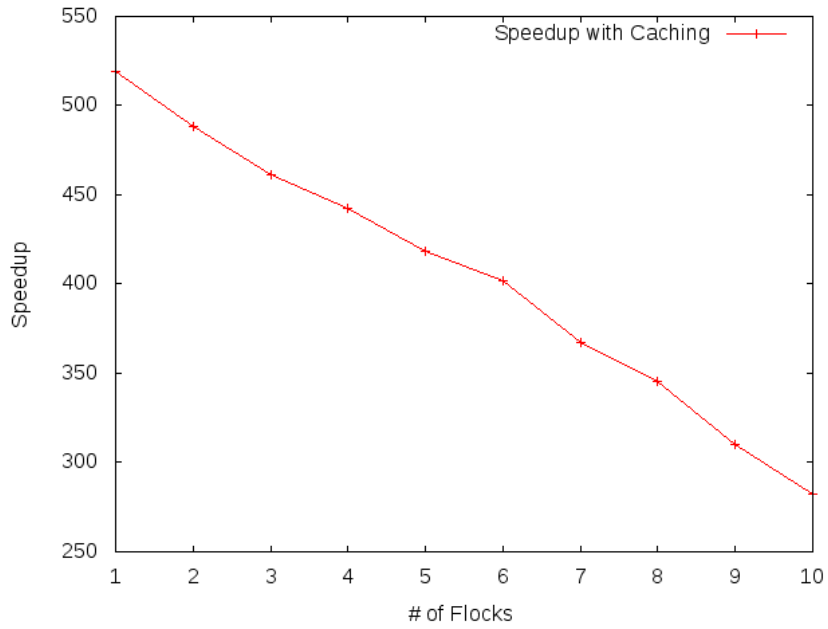

Figure 9: Speedup of caching over number of flocks

6 Conclusion and Future Work

This work has described a library that provides all facilities which are needed to add flocking and steering functionality to an existing application. The library is not bound to a predefined 3D graphics API and, using a plugin system, can be extended without the need for recompilation of either the *ufo* library or the application itself.

The basic approach of using simple steering behaviours for building more complex behaviours in a flexible and modular fashion has proved successful. No significant overhead could be observed, while the modular design allowed a low turnaround time during creation and testing of complex behaviours.

Further flexibility could be gained by creating full support for reconfiguring *ufo* during run-time. For this to work one would have to add facilities to `UfoDB`, allowing deletion of `Flock`s and `Pilot`s. Care has to be taken to gracefully deal with inter-flock and inter-pilot dependencies. These alterations would provide basic support for run-time reconfiguration, enough to create and delete pilots as steerable objects are created and deleted by an application.

Another possible field for improvement lies in leveraging hardware parallelism[12]. However, as animated swarms of animals are most often used solely as an ambient visual effect for increasing the general realism of a virtual environment, they normally must not interfere with general application performance. For this reason, even on modern multiprocessor systems, the single threaded approach taken by the *ufo* library can be sufficient and appealing for an application developer.

7 References

[1] Christoph Anthes. *A Collaborative Interaction Framework for Networked Virtual Environments*. PhD thesis, Institute of Graphics and Parallel Processing at JKU Linz, Austria, Institute of Graphics and Parallel Processing at JKU Linz, Austria, August 2009.

[2] Bruce M. Blumberg and Tinsley A. Galyean. Multi-level direction of autonomous creatures for real-time virtual environments. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 47–54, New York, NY, USA, 1995. ACM. doi:10.1145/218380.218405.

[3] Craig Reynolds et al. OpenSteer – steering behaviors for autonomous characters. Available from: http://opensteer.sourceforge.net/.

[4] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 29–38, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. doi:10.1145/311535.311538.

[5] Stephen Grand and Dave Cliff. Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems*, 1(1):39–57, 1998. doi:10.1023/A:1010042522104.

[6] Robin Green. Steering behaviours, 2000. SIGGRAPH 2000 Course on Games Research. Available from: http://www.red3d.com/siggraph/2000/course39/.

[7] ISO/IEC. Extended BNF, Dec 1996. Available from: http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[8] Aaron Isotton. C++ dlopen mini howto, March 2006. Revision 1.10. Available from: http://tldp.org/HOWTO/C++-dlopen/index.html.

[9] Andre Nathan and Valmir C. Barbosa. V-like formations in flocks of artificial birds. *Artif. Life*, 14(2):179–188, 2008. doi:10.1162/artl.2008.14.2.179.

[10] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983. doi:10.1145/357318.357320.

[11] C. Reynolds. Interaction with groups of autonomous characters. In *Proceedings of Game Developers Conference 2000*, pages 449–460, San Francisco, California, 2000. CMP Game Media Group (formerly: Miller Freeman Game Group).

[12] Craig Reynolds. Big fast crowds on ps3. In *Sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, New York, NY, USA, 2006. ACM. doi:10.1145/1183316.1183333.

[13] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, July 1987. doi:10.1145/37401.37406.

[14] Craig W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference 1999*, 1999. Available from: http://www.red3d.com/cwr/papers/1999/gdc99steer.html.