Synopsis

I will provide to jME **a clean implementation of an A.I.** in order **to control the locomotion** of "characters". Implementing this A.I will can reduce drastically the time needed to develop a game, especially a rpg or racing game, since for almost every game we need to set how these "characters", that can be a group of monsters or a car, will be moving around the scene.

I'm a passionate of video-games, 3d art and programming since I was a little boy and **I really enjoy spend all the day coding**. I'm specially motivated to this project because I would love to see the beauty of the finished algorithm, producing a 3d boids simulation.

Deliverables

When this project is finished users will be able to **provide to a vehicle** (We will call vehicle to any model that uses locomotion. It can be a character, a spaceship, etc.) the ability to:

1. **Seek** a point / Arrive to a point:
2. **Flee** away from a point or several points
3. **Pursuit** another vehicle
4. **Evade** another vehicle or a group of vehicles
5. **Avoid** obstacles
6. **Wander** around all the scene or a specific area
7. **Follow** a path
8. **Mix** several of the abilities mentioned before

After these four month of work, **the vehicle will not be able to**:

- **Directly implement more complex behaviors** (E.g: Queuing**),** You will **need** to mix correctly the abilities mentioned before.

Although you can not directly implement these complex behaviors, I will show few examples how to indirectly implement them making a proper configuration. In addition users will be able to see the given tutorials and examples.

The **GUI won't be provided.**

Project Details

With this project: characters, cars, spaceships, enemies and everything that can move will do it in a realistic way. For doing this every vehicle should be able to do a list of **basics behaviors** by separately. The codification will not require a lot of work since the algorithms are not complex but will require a lot of effort **checking and testing everything** correctly because these behaviors will be **the base of**

another features.

Furthermore, these vehicles will be able to **merge** several of the **basic behaviors** and obtain almost any wanted behavior. This **will require the most amount of work** because everything must work smoothly and not doing it well will produce bugs and unexpected results. Nevertheless we should **anticipate to possible unexpected situations** to **avoid the risk of an uncompleted project**, for this reason we have to **take in mind since the first moment that the basic behaviors should be compatible** in order to merge them without bugged stuff.

An in depth explanation how these behaviors work , with included example apps, can be found in these websites:

- http://www.red3d.com/cwr/steer/
- http://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors--gamedev-12732

An illustrative example of merging these basic behaviors can be found in this video:

- https://www.youtube.com/watch?v=86iQiV3-3IA

Finally, each behavior should include an amount of **configurable settings** in order to provide the user a way to adjust these vehicles to his own project, per example he maybe wants: only these behaviors working in the plane (2D), Add one or several lists of the obstacles, which list of obstacles should the vehicle avoid, brake speed, how fast a vehicles change his trajectory, distance to start braking... . Once the behaviors are working correctly; adding enough configurable settings should be easy and it will need the least amount of work.

Since the projects will be take in an open source environment, everything must be **clear** and be very **well documented**: This include the usage of javadoc, junit tests, documentation (In different formats) and a lot of examples showing to the developers how many options they have to combine these steer behaviors.

However, we don't have to reinvent the wheel because there is a massive amount of **information and code** about these behaviors available on the **internet**. So I will be constantly reviewing already coded algorithms and porting it to our java project.

In particular it is needed to **check the library already developed** that can be found here:

- https://code.google.com/p/jmonkeyplatform-contributions/source/browse/#svn/trunk/jme3-artificial-intelligence/release/libs

You can see that "seek", "flee", "pursuit" and "avoid" algorithms are already implemented so we need to **check**, search for possible bugs **and update all the code**. We will be reviewing the OpenSteer c++ library (http://opensteer.sourceforge.net/) and seeing what improvements we can include (making the needed adaptations) in the project.

Furthermore, **this java library** attach the agents with **JME Control class**, so there is **no reason to change that:**

It would be **possible** to make **native bindings with the c++** library, **but** one of reasons why **users use** jME is **Java** and if they want to personalize the library this will be harder. In addition, because of the simplicity of the basic steer behavior **transferring the code to Java worth it**.

We can check MetaAgent library if it is needed: http://sourceforge.net/projects/metaagent/

The algorithms remaining shall be developed **taking in mind the code already developed.** The "avoid" and "follow" behavior algorithms should be developed with specially attention to get an expected result. In particular the avoid behavior combine: the ability of brake if it is near of an obstacle and the ability to "predict the future" and the ability to change his trajectory (It Do not modify the speed).

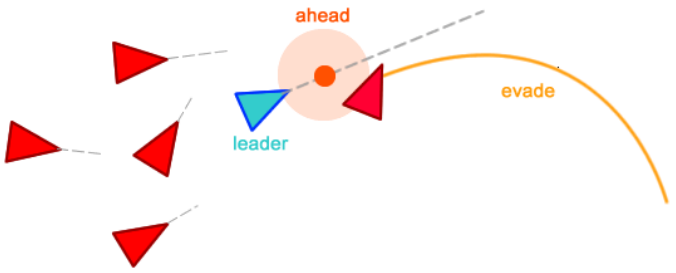
¿And what happen when the project is finished?

It would be a good idea *to provide a* ***GUI*** *that gives to the user the ability to easily include these behaviors to his own project. In order to use the GUI, they must install a plugin for the SDK. Additionally, the GUI* ***should not be complex*** *and only provide the basis of the steer behaviors, if the user want more control and power must use the library and maybe read a list of tutorials or the documentation. This feature can be developed* ***easily using*** *the advantages of "****NetBeans Plugin Development****".*

Groundwork:

These weeks I have been reading and seeing examples of jME. Then I started checking the java steer behaviors library:

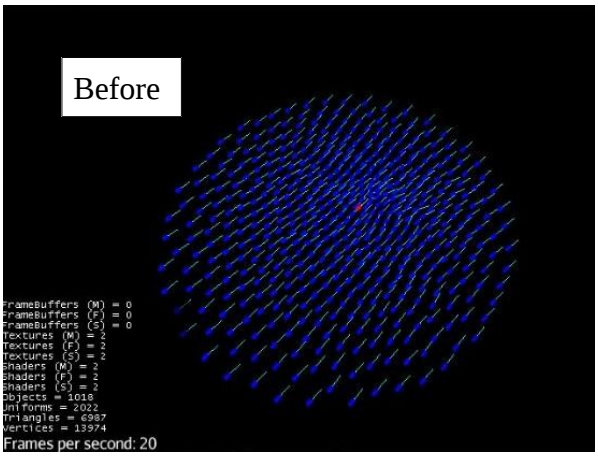
First of all you can see that t**he pursuit function do not work well,** some **pursuers stay in front of the pursued.** Instead**,** they should evade this situation like is shown on the picture.

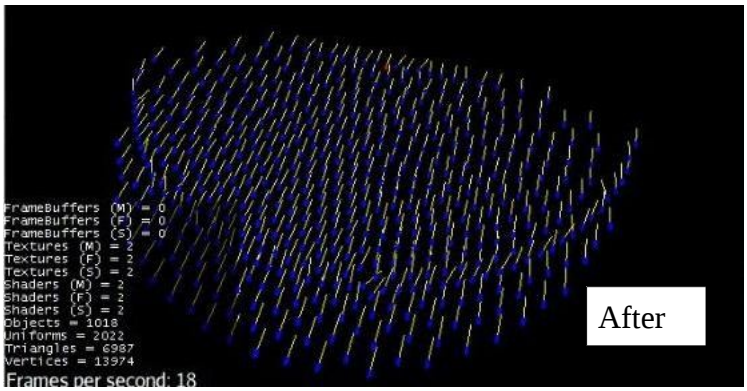


Furthermore, **the "avoid" behavior is really bugged**, the vehicles are trying to change the trajectory but "other forces" impede it. This should be further investigated in detail.

In order to fix the pursuit behavior it is needed to follow **two "rules"**:

- If the vehicle is near the objective **change his target** from the predicted location to the real location.
- If the vehicle is near the objective and is in front of the objective **push it away**.





Visual video: http://youtu.be/G2U0NRy5kNE

The ideal solution consist in implementing a blend of forces: Some forces will have more strength than other **depending of non finite circumstances;** and If you blend the forces, you have the final result.

- **Instead** of this:

```
//See if the vehicle is far away from the target
float distanceFromTrueLocation = location.distance(targetTrueLocation);
boolean isFarAwayFromTheTarget =  distanceFromTrueLocation > distanceToChangeFocus;

if(isFarAwayFromTheTarget) //Two (finite) solutions
           seekingLocation = targetPredictedLocation;
else //If its near of the target then focus the target and not the predicted location
           seekingLocation = targetTrueLocation;
```

- We can **do this:**

```
//Change the focus, non finite posible solutions
double focusFactor = this.changeFocusFactor(distanceFromTrueLocation);

seekingLocation = targetTrueLocation.add(targetPredictedLocation.subtract(
                           targetTrueLocation).mult((float) focusFactor));
```

Using the "changeFocusFactor" auxiliary function:

```
  private double changeFocusFactor(float distanceFromFocus){
        double factor;

        if(distanceFromFocus > this.distanceToChangeFocus)
            factor = 1;
        else
            factor = Math.pow((1 + distanceFromFocus/this.distanceToChangeFocus), 2); /* Real
number:infinite results */

        return factor;
    }
```

However, a **finite circumstances solution can be useful for some purposes**. Per example, We can add some **extra forces depending on a finite combination of** basic steer **behaviors**: Avoid-Pursuit, FollowPath-Avoid, etc. The extra forces will allow us to merge the behaviors nicely.

Independently, the user should be able to **change** the "distanceToChangeFocus" **variable** with a proper **API.**

In the previous video you could see some "Wiggling" and "Jittering" problems. It is very important to **identify the origin of the bug**, taking the time needed, and make a proper fix. In this case we will need to add another settable variable and improve the algorithm. In this **video** you can **see the difference** (Notable): http://youtu.be/F1yqDU86uSE

Download the source:
http://www.mediafire.com/download/2j3nsr0i3029iiz/pursuitBehaviorFixed_rev3_JMBerlanga.zip
(Zip password: **jmonkey**)

However, It still **can be improved** and is **need**ed to **be developed further**.

Finally, **custom models** can be used easily as you can see in this video: http://youtu.be/8ZboS5Mc8m8

Project Schedule

The project will require 14 weeks to be completed:

The **first working objectives may need a bit more of time** because I will have to complement my college studies with this project, as a result, **the five first weeks** (I will be able to spend 3 hours from Monday to Friday to the project, 5 hours the Saturday and 8 hours the Sunday) I will be able to work **28 hours per week**.

The remaining weeks I will be completely **free and focused on the project** so that I will be able to work 9 hours per day but the Sunday that I will work 5 hours because the mind may want to take a break. In short, I will **work 59 hours per week**.

2 weeks: May 19th – June 1st :

Gather information, what solutions can be implemented in our project **and how**.

3 weeks: June 2st – June 22th :

Check the java library **updating and fixing** "**seek**", "**flee**", "**pursuit**" **and** "**evade**" behaviors.

2 weeks: June 22th – July 4th :

Implement the **"avoid"** (including separation) behavior.

2 weeks: July 4th – July 11th :

Implement the **"wonder" and "path following"** behaviors.

3 weeks: July 11th – August 1st :

Implement the possibility of **mix several behaviors**. Identify **bugs and fix them**.

1 week: August 2nd – August 9th :

Make even more **tutorials** and **examples** with **different** deep **levels**.

1 week: August 10th – August 17th :

Review all the code, documentation, tests and examples.

When the summers ends it will be a great idea continue developing and **implement a GUI.**